Cairo University
Faculty of Computers and Artificial Intelligence
Network and Cyber Security Program

SAFE NET

THE SECURITY SHIELD

Safe Net: The Security Shield

Supervised by:
Dr. Nour Eldeen Mahmoud Khalifa

Implemented by

20216078	*Farida Mohammed Seleem*
20216038	*Radwa Khaled Fathy*
20216011	*Ahmed Mohamed Talaat*
20225085	*Mostafa Hesham Sherif*
20216138	*Karim Wael Marwan*
20215044	*Yousef Yasser El-Kady*

Graduation Project

Final Year Project (Final Discussion)

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

Acronym	Definition of Acronym	Page
AI	Artificial Intelligence	6
API	Application Programming Interface	6
URLs	Uniform Resource Locators	6
WHOIS	Domain registration lookup	6
EML	Email Message Format file	7
IPQS	IPQualityScore	9
HTML	Hyper Text Markup Language	11
CSS	Cascading Style Sheets	11
SQL	Structured Query Language	12
LLM	Large Language Model	15
HTTP	Hyper Text Transfer Protocol	16
npm	Node Package Manager	16
CORS	Cross-Origin Resource Sharing	16
.env	Environment file	17
ODM	Object Document Mapper	17
JWT	JSON Web Token	18
XSS	Cross-Site Scripting	18
CMS	Content Management System	18
TLS	Transport Layer Security	21
HTTPS	Hyper Text Transfer Protocol Secure	21
OWASP	Open Web Application Security Project	23
JSON	JavaScript Object Notation	23
SOC	Security Operations Center	35
GDPR	General Data Protection Regulation	36
HIPAA	Health Insurance Portability and Accountability Act	36
ISO	International Organization for Standardization	36

1. ABSTRACT

With the rise of cyber threats, phishing attacks have become a major security risk, leading to financial losses and data breaches. Attackers continuously refine their tactics, using malicious websites and phishing emails to deceive users. Traditional detection methods often struggle to keep up, highlighting the need for a real-time, efficient, and accessible phishing detection system.

SafeNet: The Security Shield addresses this challenge by providing a website-based platform that scans URLs and email files for potential threats. The Standard Plan utilizes the VirusTotal API for basic detection, while the Premium Plan integrates multiple APIs, such as IPQualityScore, Scamalytics and Hybrid analysis for enhanced security. SafeNet also employs heuristic analysis and WHOIS lookup to improve detection accuracy, along with an awareness module that educates users when phishing threats are detected. Additionally, the system features a chatbot for communication platforms, leveraging AI-API to provide real-time security guidance.

Built with JavaScript for the backend, React for the frontend, and MongoDB for database management, SafeNet seamlessly integrates external APIs for real-time detection. GitHub is used for version control, ensuring a structured development process. By combining API-driven detection, heuristic analysis, and real-time user awareness, SafeNet offers a scalable and user-friendly cybersecurity solution.

2. BACKGROUND

Phishing is one of the most common and dangerous threats in today's digital world. It tricks users into giving away sensitive information like passwords, bank details, or personal data through fake websites, misleading emails, or malicious attachments. These attacks are getting more advanced every day, and many users—regardless of technical skill—struggle to identify them. While several security tools exist, most are limited by platform, difficult to use, or fail to provide clear explanations, leaving users unprotected across devices. [4]

SafeNet was created to solve this problem by offering a smart, accessible, and user-friendly phishing detection platform. It protects users in real-time while helping them understand how to avoid threats in the future. The system is designed for a wide range of users, including individuals, businesses, and educational institutions, offering a simple way to stay safe online.

At the heart of SafeNet is a powerful multi-layered scanning engine that combines:

- **Heuristic analysis** to detect suspicious patterns,
- **WHOIS domain checks** for trust evaluation [13],
- And **API integrations** with platforms like VirusTotal, IPQualityScore, Scamalytics, and Hybrid Analysis to validate threats.

SafeNet supports three main features:

1. **Scan URL** – to check suspicious links in real-time.
2. **Scan Email File (.eml)** – to detect malicious headers, spoofed senders, phishing links, and harmful attachments.
3. **Report URL** – allowing users to submit known phishing links to a shared threat database.

These capabilities are delivered across platforms: through a **responsive web app**, **browser extension**, **Telegram bot**, and an **interactive chatbot**, ensuring users stay protected anywhere, anytime. Each scan result includes clear explanations to improve user awareness and digital literacy.

SafeNet goes beyond detection. It empowers users to learn, understand, and recognize threats themselves. This focus on **education and accessibility** makes SafeNet not just a security tool, but a long-term solution for building a safer internet.

In summary, SafeNet is a comprehensive phishing protection platform that combines real-time threat detection, user education, and cross-platform support. It transforms the way users interact with cybersecurity turning everyday people into proactive defenders of their own digital safety.

3. PROBLEM DEFINITION

Phishing attacks continue to be one of the most serious threats facing internet users today. These attacks rely on tricking users into clicking on malicious links, opening harmful attachments, or sharing private information through fake websites and emails. As phishing methods grow more sophisticated—using social engineering, spoofed domains, and carefully crafted emails—many users remain unaware of how to detect or avoid these threats.

To address this issue, there is a growing need for a reliable and user-friendly phishing detection system that works in real time and across multiple platforms. Many existing tools are either too technical, reactive instead of proactive, or inaccessible to everyday users. SafeNet was developed to fill this gap by providing a simple yet powerful tool that helps users identify, understand, and report phishing threats before any damage is done.

Current Security Challenges:

- **Phishing Websites [1][2]:**
  A new phishing site appears every 20 seconds, often mimicking trusted services to trick users. Many of these sites bypass traditional security filters and are active for short periods, making them hard to block in time.

- **Email-Based Phishing:**
  Malicious emails are the most common delivery method for phishing attacks. These emails can contain fake sender information, misleading links, and infected attachments that target users who lack the technical knowledge to spot red flags.

- **Lack of User Awareness:**
  Most users are not trained to recognize phishing techniques. Existing tools often block threats silently without explaining why, leaving users uneducated and vulnerable to future attacks.

- **Limited Accessibility:**
  Many advanced phishing detection tools are designed for enterprise environments, not individuals. They require setup, or training barriers that prevent wide adoption among students, small businesses, and the general public.

SafeNet responds to these challenges by offering a smart, multi-platform solution that scans links, analyzes emails, and supports threat reporting all while educating users with clear feedback. This combination of detection, explanation, and accessibility empowers users to take control of their online safety and builds a stronger, more resilient defense against phishing attacks.

4. RELATED WORK

Table 4.1:

Features	SafeNet	VirusTotal	IPQS	PhishTank	URLVoid
URL scanning	✓	✓	✓	✓	✓
Attachment Scanning	✓	✓	✓	✗	✗
Multiple APIs	✓	✓	✓	✗	✓
Header Scanning	✓	✗	✓	✗	✗
User awareness	✓	✗	✓	✗	✗
Whois lookup	✓	✗	✓	✗	✓
Chrome Extension	✓	✓	✗	✗	✗
Free version	✓	✓	✓	✓	✓
Paid version	✓	✓	✓	✗	✗
URL user report	✓	✗	✓	✓	✗
Email file scanning	✓	✗	✗	✗	✗
Heuristic analysis	✓	✗	✗	✗	✗
Chatbot integration	✓	✗	✗	✗	✗

With the rise of cyber threats, particularly phishing attacks, a wide range of phishing detection systems and cybersecurity platforms have emerged, each aiming to mitigate these evolving risks. These platforms often implement various techniques such as rule-based blacklists, threat intelligence integration, and third-party security APIs. While effective in isolation, these systems tend to operate independently, focusing on specific aspects of phishing prevention without offering a holistic or centralized framework.

For instance, platforms like **VirusTotal** excel in multi-engine antivirus scanning and URL checking, but they primarily rely on aggregated antivirus engine results and do not provide contextual awareness or educational support for users. Similarly, **IPQualityScore (IPQS)** delivers high-quality risk scoring and fraud detection through advanced algorithms, but it functions mainly as an API service and lacks user interaction features or awareness components. On the other hand, **PhishTank** leverages a crowdsourced approach to collect and share phishing URLs, which contributes to global threat intelligence but is limited by delayed reporting and the need for human verification.

The main limitation among these solutions lies in their fragmentation each tool or platform solves only part of the problem. Organizations and individuals often need to rely on multiple, disconnected tools to scan URLs, analyze email files, perform domain reputation checks, and educate users about potential threats. This decentralized approach results in inefficiencies, potential data silos, inconsistent user experiences, and increased complexity for non-technical users.In contrast, ***SafeNet: The Security Shield*** is designed to address these gaps through the following

Main Differences

- **Centralized Security Platform**
  Existing phishing detection solutions often operate in isolation each tool typically handles only one task, such as URL scanning (e.g., VirusTotal), risk scoring (e.g., IPQS), or crowdsourced reporting (e.g., PhishTank). [3]
  ➤ **SafeNet combines multiple capabilities into a centralized platform**, unifying URL and email file scanning, API-based detection, WHOIS lookups, heuristic analysis, and reporting into one seamless system.

- **Heuristic Analysis**
  Many existing platforms rely on static blacklists or predefined patterns, which can miss new or evolving phishing tactics.
  ➤ **SafeNet enhances detection accuracy through heuristic analysis**, identifying phishing indicators such as suspicious keywords and obfuscated links improving the system's ability to detect zero-day threats.

- **Email File Scanning**
  Most tools focus solely on scanning URLs and do not support the analysis of complete email files.
  ➤ **SafeNet allows users to upload and scan email files**, parsing headers, analyzing content, and inspecting attachments for phishing attempts or malware delivering a more comprehensive assessment.

- **AI Chatbot Integration**
  Traditional detection platforms provide little to no real-time interaction or guidance for users.
  ➤ **SafeNet includes an AI-powered chatbot**, enabling users to scan suspicious links or messages through communication platforms and receive instant, actionable feedback making cybersecurity more accessible and responsive.

- **Built-in Awareness Module**
  Competing platforms focus on detection but rarely educate users about the threats they encounter.
  ➤**SafeNet integrates an awareness module** that delivers phishing prevention tips and security insights immediately after a threat is detected bridging the gap between threat identification and user education.

5. USED TECHNOLOGIES

Frontend

- **React.js**
    - **Overview:** A JavaScript library for building dynamic user interfaces through reusable components. React enhances the responsiveness and interactivity of the SafeNet frontend.
    - **Use Cases:**
        - **Build reusable components:** Create modular UI elements for the web dashboard.
        - **Integrate real-time scan results:** Dynamically display data as it updates without reloading the page.
    - **Benefits:**
        - **Instant feedback:** Fast re-rendering allows quick display of scan results.
        - **Seamless navigation:** Enables smooth transitions between different dashboard sections.
        - **Efficient API integration:** Easily connect to VirusTotal, IPQS, and WHOIS APIs within components.
- **HTML5 & CSS**
    - **Overview:** Standard web technologies for building and styling the structure of SafeNet's frontend.
    - **Use Cases:**
        - **Page layout creation:** Structure scan forms, result displays, and user dashboards.
        - **Responsive design implementation:** Ensure cross-browser compatibility and mobile-friendly layout.
    - **Benefits:**
        - **Lightweight performance:** Fast loading times with minimal code overhead.
        - **Consistent design:** Uniform UI across pages and screen sizes.
- **JavaScript**
    - **Overview:** Core programming language for SafeNet's frontend logic and API interactions.

- **Use Cases:**
  - **Handle user input:** Process form data and scan submissions.
  - **Call backend APIs:** Send data to the Node.js server and receive scan results.

- **Benefits:**
  - **Interactive UI:** Enables dynamic elements and real-time updates.
  - **Asynchronous operations:** Supports background API calls without blocking the UI.

- **Chrome Extension API**

  - **Overview:** Provides SafeNet's Chrome extension with access to browser context menus and event listeners.

  - **Use Cases:**
    - **Right-click scanning:** Instantly scan URLs from the browser context menu.
    - **Trigger scans remotely:** Launch scans without navigating to the main web interface.

  - **Benefits:**
    - **Seamless user flow:** Users can interact with SafeNet while browsing.
    - **Lightweight integration:** Minimal performance overhead on the browser.

Backend

- **Node.js**

  - **Overview:** JavaScript runtime for building RESTful APIs and backend logic in SafeNet.

  - **Use Cases:**
    - **User session handling:** Manage login, plan selection, and logout.
    - **File and URL scan processing:** Forward requests to scanners and return verdicts.

  - **Benefits:**
    - **Asynchronous design:** Handles multiple requests efficiently.
    - **Scalable microservices:** Supports modular development and deployment.

- **MongoDB Atlas [11]**

  - **Overview:** A cloud-hosted NoSQL database used to manage user data, scan logs, and subscription plans.

- **Use Cases:**
  - **Store user accounts and roles:** Manage login credentials, plan type (Standard or Premium), and access control in the users collection.
  - **Log scan activity:** Track each scan action with timestamp and associated user via the scan logs collection.
  - **Track login sessions:** Store session tokens and expiration timestamps in the sessions collection for session-based authentication.
  - **Maintain threat database:** Persist manually reported or auto-detected phishing URLs in the blocked URLs collection for real-time threat blocking.

- **Benefits:**
  - **Cloud scalability:** Handles varying workloads without manual scaling.
  - **Flexible schema:** Easily store complex scan-related data.

Security & Scanning APIs

- ### VirusTotal API [6]

  - **Overview:** Industry-standard threat detection service for URL and file scanning.

  - **Access Level:** Available for both Standard and Premium users.

  - **Use Cases:**
    - **Integrated scanning engine:** Primary scanner available to all user plans.

  - **Benefits:**
    - **Broad detection coverage:** Aggregates results from over 70 engines.
    - **Trusted results:** Widely recognized and respected in the cybersecurity industry.

- ### IPQualityScore (IPQS) [7]

  - **Overview:** Threat scoring API used for premium-level email header and URL scans.

  - **Access Level:** Premium-only feature.

  - **Use Cases:**
    - **Evaluate phishing risks:** Assess email headers, links, and sender domains.

  - **Benefits:**
    - **Granular scoring:** Provides detailed threat classification.
    - **Low latency:** Fast risk assessment in real time.

- **Scamalytics API [9]**
    - **Overview:** Premium IP and domain intelligence used to detect fraud and abuse patterns.
    - **Access Level:** Premium-only feature.
    - **Use Cases:**
        - **rich scan results:** Add fraud indicators from IP and domain intelligence.
    - **Benefits:**
        - **Real-time updates:** Constantly refreshed threat intelligence.
        - **Enhanced accuracy:** Improves detection of scam-related content.
- **Hybrid Analysis API [8]**
    - **Overview:** Cloud sandbox analysis tool for file attachments.
    - **Access Level:** Premium-only feature.
    - **Use Cases:**
        - **Deep malware analysis:** Simulate file behavior in a secure environment.
    - **Benefits:**
        - **Behavior-based detection:** Detects zero-day and evasive threats.
        - **Premium-grade analysis:** Suitable for file attachments from .eml emails.
- **WHOIS Lookup**
    - **Overview:** Extracts domain registration data for heuristic threat scoring.
    - **Use Cases:**
        - **Detect suspicious domains:** Identify newly registered or anonymous domains.
    - **Benefits:**
        - **Heuristic context:** Helps in phishing classification.
        - **Complements scanners:** Adds non-signature-based insight.

AI & Chatbot

- **Gemini Flash 1.5 [17]**
    - **Overview:** LLM used to analyze scan results and generate threat summaries.
    - **Use Cases:**
        - **Explain scan results:** Provide understandable summaries to users.
    - **Benefits:**
        - **Improved awareness:** Users learn about threats in plain language.
        - **Fast inference:** Lightweight enough for near-instant response.
- **Telegram Bot API [10]**
    - **Overview:** Enables SafeNet's bot to communicate scan results and alerts via Telegram.
    - **Use Cases:**
        - **Allow mobile scans:** Submit links from Telegram chats.
        - **Push alerts:** Notify users of scan completions or threats.
    - **Benefits:**
        - **Mobile integration:** Extends SafeNet's reach beyond the browser.
        - **Simple interface:** Easy to use for non-technical users.

Dev Tools

- **Visual Studio Code [21]**
    - **Overview:** Main development environment for both frontend and backend code.
    - **Benefits:**
        - **Extension support:** Integrates with MongoDB, REST Client, and ESLint.
- **Postman [20]**
    - **Overview:** API testing tool for debugging REST endpoints.
    - **Benefits:**
        - **Simulate requests:** Test scan routes with payloads.
        - **Automate testing:** Save test collections for reuse.

- **Git + GitHub [18]**

    - **Overview:** Version control system for managing SafeNet's source code and collaboration.

    - **Benefits:**
        - **Track development:** Monitor changes and rollbacks.
        - **Enable collaboration:** Allow multiple contributors.

- **Npm**

    - **Overview:** Package manager for JavaScript used to manage SafeNet's dependencies.

    - **Benefits:**
        - **Extensive ecosystem:** Access to thousands of Node packages.
        - **Script automation:** Run build, test, and deployment scripts.

Additional Libraries Used

- **Express.js [19]**

    - **Overview:** Minimalist web framework for Node.js used for building REST APIs.

    - **Benefits:**
        - **Simplified routing:** Easily define endpoints for scanning, authentication, and reporting.
        - **Middleware support:** Enables integration with tools like cors, helmet, and jsonwebtoken.

- **Multer**

    - **Overview:** Middleware for handling multipart/form-data, used for .eml file uploads.

    - **Benefits:**
        - **Efficient file handling:** Parses uploaded files into buffers or local storage.
        - **Security-focused:** Limits file size and type to reduce risk.

- **Axios**

    - **Overview:** Promise-based HTTP client for the browser and Node.js.

- **Benefits:**
  - **Clean syntax:** Simplifies making async requests to scanning APIs.
  - **Built-in error handling:** Catches and logs issues with external services.

- **Dotenv**

  - **Overview:** Loads environment variables from .env into process.env.

  - **Benefits:**
    - **Secure credentials:** Keeps API keys and DB URLs out of the source code.
    - **Environment management:** Allows different configs for dev, test, and production.

- **Cors**

  - **Overview:** Express middleware for enabling Cross-Origin Resource Sharing.

  - **Benefits:**
    - **Frontend-backend integration:** Allows React app to communicate with the Node.js server.
    - **Flexible policy control:** Restricts or opens origins, methods, and headers.

- **html2canvas**

  - **Overview:** Frontend library to export DOM content as images.

  - **Benefits:**
    - **User export support:** Allows users to download scan results as screenshots.
    - **Client-side rendering:** No server processing needed.

- **Mongoose [22]**

  - **Overview:** ODM for MongoDB used in Node.js environments.

  - **Benefits:**
    - **Schema enforcement:** Defines consistent document structures.
    - **Query abstraction:** Simplifies complex MongoDB operations.

- **Bcryptjs**

  - **Overview:** Password hashing library for Node.js.

- **Benefits:**
  - **Strong encryption:** Safely hashes and compares passwords.
  - **Asynchronous performance:** Doesn't block event loop during login.

Why These Technologies Are Better for Building SafeNet

1. Customization and Flexibility

- **Custom-built interface:** Using HTML5, CSS3, JavaScript, and Bootstrap 5 gave us complete control over the UI/UX. We could precisely tailor the layout, components, and API interactions to suit the phishing detection use case, including dynamic scan result rendering and real-time feedback.
- **Avoiding CMS limitations:** A traditional CMS would have imposed rigid templates and required complex third-party plugins to replicate similar functionality especially for real-time scanning and asynchronous API behavior.

2. Performance

- **Fast page load & responsiveness:** Our approach used lightweight components, modular JavaScript, and asynchronous API calls (e.g., via Axios). This ensured faster interaction, minimized initial load times, and allowed us to display scan results and risk scores almost instantly.
- **No CMS overhead:** Pre-built CMS platforms often include bloated themes and unused resources, which slow down page load and increase memory usage—leading to a degraded user experience, especially on low-end devices or mobile browsers.

3. Security

- **Direct control over security mechanisms:** By coding the backend and frontend from scratch, we had full authority over HTTP headers, CORS policies, rate limiting, and session-based authentication. This ensured we could enforce secure defaults and adapt security rules to changing requirements.
- **Avoiding plugin-related risks:** CMS-based solutions heavily rely on third-party plugins, many of which are not regularly updated or audited. This could introduce vulnerabilities such as XSS, SQL Injection, or privilege escalation none of which we wanted in a phishing detection platform.

4. Learning and Skill Development

- **Hands-on full-stack experience:** Building SafeNet without CMS shortcuts meant directly writing frontend logic, managing REST API requests, and implementing backend session handling. This significantly deepened our team's knowledge of HTTP, REST, CORS, JWT/session tokens, and frontend-backend coordination.
- **No drag-and-drop shortcuts:** Unlike CMS builders that offer WYSIWYG editors, our custom build required us to manually debug issues, optimize performance, and structure the codebase efficiently enhancing long-term software engineering skills.

5. Scalability and Maintenance

- **Microservice-ready architecture:** Our backend design—using Express.js with MongoDB—was structured into modular services like URL scanning, email analysis, user management, and chatbot integration. This separation makes future scaling, containerization (e.g., Docker), and CI/CD deployments much easier.
- **CMS-based scalability issues:** In contrast, CMS plugins often conflict during updates, rely on shared resources, and are tightly coupled making automated deployments, scaling via microservices, or performance tuning significantly harder and riskier.

6. SECURE CODE PRACTICES

These standards implement essential security requirements considered necessary to provide a secure baseline for the SafeNet platform. Writing secure code reduces the risk of introducing software defects that cause vulnerabilities and improves the overall quality, reliability, and maintainability of the application.

- **Authentication and Access Control [12]**

    - **Authentication:** Users' identities are verified before accessing protected resources through session-based authentication with secure cookies.
    - **Access Control:** Role-Based Access Control (RBAC) ensures users (Admin, Standard, Premium) only access functionalities assigned to their roles via route-level middleware.

    Implemented in:
    ACCESS_CONTROL.md, auth.js, security.js

    Practices:
    - Role enforcement through middleware
    - Segregation of access for Standard and Premium plans

- **Input Validation and Sanitization**

    - **Input Validation:** User-submitted data (query parameters, request bodies) is validated against predefined formats and criteria.
    - **Input Sanitization:** Recursively removes malicious patterns like NoSQL injections ($gt,, etc.) to safeguard database operations.

    Implemented in:
    NOSQL_INJECTION_PROTECTION.md, secureDatabase.js

    Practices:
    - Avoiding direct use of user input in queries
    - Middleware-based input sanitization to prevent NoSQL injection

- **Error Handling and Logging**

    Errors are caught using try/catch to avoid exposing internal details. Stack traces and sensitive database errors are never shown to the user. Logs record suspicious input patterns, unauthorized access attempts, and scanning activities.

    Implemented in:
    secureDatabase.js, server.js, logRoutes.js

Practices:

- Centralized error logging
- Use of try/catch blocks to avoid unhandled exceptions
- Monitoring scan requests and system behavior

- **Password Management**

  SafeNet implements password policies to ensure users create strong, secure passwords. These policies enforce minimum password length, complexity (including a mix of uppercase, lowercase, numbers, and special characters).

  Implemented in:
  models/User.js, auth.js

  Practices:

  - Enforcement of strong password complexity and minimum length

- **Secure Data Storage and Transmission**

  All communication between the frontend and backend is secured with HTTPS and TLS. Secrets such as database credentials or API tokens are securely stored in .env files.

  Implemented in:
  server.js, .env, auth.js

  Practices:

  - Enforced HTTPS for secure communication
  - Use of environment variables for sensitive configurations
  - Secure cookie settings (HttpOnly, Secure, SameSite)

- **Security Misconfiguration Prevention (Directory traversal protection)**

  Backend framework details are hidden by disabling the `x-powered-by` header. Security middleware is initiated early to ensure coverage of all routes. Directory traversal attempts (`../`) are detected and logged to prevent unauthorized file access.

  Implemented in:
  security.js, server.js

  Practices:

  - Disabled unnecessary headers
  - Early middleware initialization
  - Detection and logging of directory traversal attempts (../)

- **Cryptographic Practices**

  Integral to protecting sensitive user data and ensuring secure session management. Cryptographic practices include secure hashing (bcrypt), cryptographically secure random generation for verification codes, and encrypted data transmission.

  Implemented in:
  auth.js, User.js, server.js

  Practices:
  - Secure password hashing using bcrypt
  - Secure session management tokens
  - Cryptographically secure random generation for verification codes

- **Rate Limiting and Brute Force Protection**

  Protects against automated abuse by implementing IP-based rate limiting (e.g., 100 requests/minute). Suspicious activities are logged for further security measures.

  Implemented in:
  security.js, server.js

  Practices:
  - Rate-limiting middleware configuration
  - Logging of suspicious activities

- **CORS Policy Enforcement**

  Explicitly restricts cross-origin requests to trusted frontend origins, controlling preflight requests to prevent cross-origin abuse.

  Implemented in:
  security.js, server.js

  Practices:
  - Explicit CORS configuration

- **Session Management**

  SafeNet uses secure, server-side session management to maintain authenticated user state. Sessions are stored in MongoDB using connect-mongo and are protected by secure cookies with short expiration periods. On logout, session data is explicitly destroyed both in memory and in the database to prevent reuse. All sessions are tied to authenticated users and validated on every protected request to prevent unauthorized access.

Implemented in:
auth.js, server.js

Practices:

- Server-side session storage using MongoDB
- Secure cookie flags: HttpOnly, Secure, SameSite=Lax
- Session expiration and auto-removal for inactive users
- Session validation before accessing protected resources
- Explicit session ID termination and removal from MongoDB on logout

- **OWASP Secure Coding Practices Checklist [5] SafeNet Implementation Summary:**

Table 6.1:

	Input Validation
1.	**Conduct all input validation on a trusted system (server side not client side)** — This is fully enforced in SafeNet using Flask server-side validation for login, scan, and all input fields including file uploads.
2.	**Identify all data sources and classify them into trusted and untrusted** — All user inputs, file uploads, and URLs are treated as untrusted and sanitized before further processing.
3.	**Validate all data from untrusted sources (databases, file streams, etc)** — SafeNet validates emails and URLs from untrusted uploads and external scan results.
4.	**Use a centralized input validation routine for the whole application** — Input validation is implemented in most routes but not centralized into a shared module (partially implemented).
5.	**Specify character sets, such as UTF-8, for all input sources (canonicalization)** — SafeNet enforces UTF-8 decoding especially during email header extraction and WHOIS checks.
6.	**Encode input to a common character set before validating** — Applied during email decoding, but not consistently across all endpoints (partially implemented).
7.	**All validation failures should result in input rejection** — Any invalid email file, malformed URL, or bad JSON input is rejected with a proper error.
8.	**Validate all client provided data before processing** — This includes email headers, uploaded files, URL query params, and login inputs.
9.	**Validate for expected data types using an "allow" list rather than a "deny" list** — For example, .eml file type, email format, and rating numbers are validated using positive criteria.
10.	**Validate data range** — Rating scores (1–5), URL length, and user input lengths are checked.
11.	**Validate data length** — Max lengths for email fields, usernames, and passwords are enforced in the backend.

12.	**If any potentially hazardous input must be allowed, then implement additional controls** — Email attachments are scanned using VirusTotal or Hybrid Analysis to mitigate risky file contents.
13.	**If the standard validation routine cannot address some inputs, then use extra discrete checks** — Used in email header scanning and raw email parsing for edge cases.
14.	**Utilize canonicalization to address obfuscation attacks** — Applied in decoding techniques during heuristic email URL and header analysis (partially).
	Output Encoding
15.	**Conduct all output encoding on a trusted system (server side not client side)** — All HTML and JSON responses are built server-side.
16.	**Utilize a standard, tested routine for each type of outbound encoding** — Encoding is partially applied during JSON logging and template rendering.
17.	**Specify character sets, such as UTF-8, for all outputs** — Flask defaults and custom headers ensure UTF-8 in HTML, JSON, and CSV responses.
18.	**Contextually output encode all data returned to the client from untrusted sources** — Untrusted data like scan results, email subjects, and ratings are encoded.
19.	**Ensure the output encoding is safe for all target systems** — Logs, scan summaries, and downloadable content avoid unsafe characters.
20.	**Contextually sanitize all output of un-trusted data to queries for SQL, XML, and LDAP** — Not directly applicable due to MongoDB and no raw queries.
21.	**Sanitize all output of untrusted data to operating system commands** — SafeNet does not invoke OS commands from user data (compliant).
	Authentication and Password Management
22.	**Require authentications for all pages and resources, except those specifically intended to be public** — All scan, dashboard, and profile features require login.
23.	**All authentication controls must be enforced on a trusted system** — Enforced with session and bcrypt password checks.
24.	**Establish and utilize standard, tested, authentication services whenever possible** —Login and server-side hashed sessions are used.
25.	**Use a centralized implementation for all authentication controls** — All login and session management are handled in the main auth module.
26.	**All authentication controls should fail securely** — Errors are handled without revealing whether the username or password is wrong.

27.	**All administrative and account management functions must be at least as secure as the primary authentication mechanism** — Admin panel uses same bcrypt and session checks.
28.	**If your application manages a credential store, use cryptographically strong one-way salted hashes** — Passwords are stored using bcrypt hashing.
29.	**Password hashing must be implemented on a trusted system (server side not client side)** — Implemented entirely in backend.
30.	**Validate the authentication data only on completion of all data input** — Input is processed after full JSON parsing.
31.	**Authentication failure responses should not indicate which part of the authentication data was incorrect** — Generic error messages are used for failed login.
32.	**Utilize authentication for connections to external systems that involve sensitive information or functions** — Authenticated user info is used when calling scan APIs.
33.	**Authentication credentials for accessing services external to the application should be stored in a secure store** — Some API keys are still hardcoded during dev (partially implemented).
34.	**Use only HTTP POST requests to transmit authentication credentials** — Login uses POST requests exclusively.
35.	**Only send non-temporary passwords over an encrypted connection or as encrypted data** — HTTPS enforced for login and registration.
36.	**Enforce password complexity requirements established by policy or regulation** — Basic enforcement (min 8 chars, letters & numbers).
37.	**Enforce password length requirements established by policy or regulation** — 8-character minimum required.
38.	**Password entry should be obscured on the user's screen** — Input field uses type="password".
39.	**If using email-based resets, only send email to a pre-registered address with a temporary link/password** — Implemented using email verification.
40.	**Temporary passwords and links should have a short expiration time** — Tokens expire within 15 minutes.
41.	**Change all vendor-supplied default passwords and user IDs or disable the associated accounts** — No default accounts used.
42.	**Re-authenticate users prior to performing critical operations** — Re-authentication prompt shown during plan upgrade (partially implemented).
	Session Management
43.	**Use the server or framework's session management controls** — SafeNet uses sessions to manage authenticated state securely.
44.	**Session identifier creation must always be done on a trusted system (server side not client side)** — Session IDs are generated securely on the backend.
45.	**Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers** — Default session implementation uses signed cookies.

46.	**Logout functionality should fully terminate the associated session or connection** — Logout route clears session data.
47.	**Logout functionality should be available from all pages protected by authorization** — Provided consistently in the dashboard UI.
48.	**Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements** — Implemented with a 30-minute timeout.
49.	**If a session was established before login, close that session and establish a new session after a successful login** — Session is re-issued upon successful login.
50.	**Generate a new session identifier on any re-authentication** — Sessions are reset when logging in again.
51.	**Do not expose session identifiers in URLs, error messages or logs** — Session ID never appears in URL or logs.
52.	**Implement appropriate access controls to protect server-side session data from unauthorized access** — Session data is protected by secure cookie implementation.
53.	**Consistently utilize HTTPS rather than switching between HTTP to HTTPS** — All production instances use HTTPS exclusively.
54.	**Set the "secure" attribute for cookies transmitted over an TLS connection** — Secure cookie flag is enabled.
55.	**Set cookies with the HttpOnly attribute** — Cookies are flagged as HttpOnly for added protection.
	Access Control
56.	**Use only trusted system objects, e.g. server-side session objects, for making access authorization decisions** — SafeNet uses sessions to determine logged-in user identity and privileges.
57.	**Use a single site-wide component to check access authorization** — Decorators and middleware verify session roles on all protected routes.
58.	**Access controls should fail securely** — If session is invalid or missing, access is denied with appropriate error message.
59.	**Enforce authorization controls on every request, including those made by server side scripts** — Each protected endpoint checks role and session validity.
60.	**Segregate privileged logic from other application code** — Admin routes are isolated and require elevated session roles.
61.	**Restrict access to files or other resources, including those outside the application's direct control** — Users cannot access or download internal files outside their scope.
62.	**Restrict access to protected URLs to only authorized users** — Admin dashboard and scan history pages require proper session roles.
63.	**Restrict access to protected functions to only authorized users** — Role-based logic prevents unauthorized access to scan features.
64.	**Restrict direct object references to only authorized users** — ID checks and role restrictions are enforced on data fetching.

65.	**Restrict access to services to only authorized users** — API endpoints are protected with session and role guards.
66.	**Restrict access to application data to only authorized users** — Mongo queries are filtered based on session user's ID and role.
67.	**Restrict access security-relevant configuration information to only authorized users** — API keys, plans, and logs are only visible to admins.
68.	**Server side implementation and presentation layer representations of access control rules must match** — Frontend hides buttons and backend enforces permission checks.
69.	**If state data must be stored on the client, use encryption and integrity checking on the server side to detect state tampering** — Minimal client state is stored, and session integrity is checked via cookies.
70.	**Enforce application logic flows to comply with business rules** — Access flows such as upgrade, scan, view history follow defined user roles and payment status.
71.	**Limit the number of transactions a single user or device can perform in a given period of time** — Rate limiting under development (partially implemented).
72.	**Service accounts or accounts supporting connections to or from external systems should have the least privilege possible** — External API credentials are limited to scan-only permissions.
	Cryptographic Practices
73.	**All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system** — Password hashing and token generation are done in backend.
74.	**Protect secrets from unauthorized access** — Credentials and keys are stored in environment variables and not exposed to clients.
75.	**Cryptographic modules should fail securely** — If encryption/hashing fails, fallback is disabled and errors are handled.
76.	**All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator** — Secure tokens and filenames generated using os.urandom and secrets module.
	Error Handling and Logging
77.	**Do not disclose sensitive information in error responses** — All public errors are generic and do not leak details.
78.	**Use error handlers that do not display debugging or stack trace information** —error handlers catch and mask server errors.
79.	**Implement generic error messages and use custom error pages** — Invalid session, scan errors, and access issues return clean JSON messages.
80.	**The application should handle application errors and not rely on the server configuration** — Try-except blocks handle major logic exceptions.
81.	**Error handling logic associated with security controls should deny access by default** — Fail-closed behavior applied on auth and scan checks.

82.	**All logging controls should be implemented on a trusted system** — Logs written by backend only.
83.	**Logging controls should support both success and failure of specified security events** — Successful logins, failed scans, and validation errors are recorded.
84.	**Ensure logs contain important log event data** — Logs include timestamp, user ID and action.
85.	**Restrict access to logs to only authorized individuals** — Only admins can access scan logs.
86.	**Utilize a central routine for all logging operations** — Logging is routed through a shared module.
87.	**Do not store sensitive information in logs** — Passwords, session tokens, and user secrets are never logged.
88.	**Ensure that a mechanism exists to conduct log analysis** — Logs are quarriable by admin.
89.	**Log all input validation failures** — Malformed input, missing fields, and invalid data types are logged.
90.	**Log all authentication attempts, especially failures** — Login failures with timestamp are stored.
91.	**Log all access control failures** — Unauthorized access attempts are logged.
92.	**Log attempts to connect with invalid or expired session tokens** — Expired session actions trigger log entries.
93.	**Log all system exceptions** — Backend logs record exception types and location.
94.	**Log all administrative functions, including changes to the security configuration settings** — Admin login, scan limits, and plan upgrades are logged.
	Data Protection
95.	**Implement least privilege, restrict users to only the functionality, data and system information that is required to perform their tasks** — User roles (standard, premium, admin) determine accessible features.
96.	**Protect all cached or temporary copies of sensitive data stored on the server from unauthorized access and purge those temporary working files as soon as they are no longer required** — Uploaded files are deleted immediately after scan.
97.	**Encrypt highly sensitive stored information, such as authentication verification data, even if on the server side** — Passwords hashed with bcrypt.
98.	**Protect server-side source-code from being downloaded by a user** — Server prevents public file listing or source code exposure.
99.	**Do not store passwords, connection strings or other sensitive information in clear text or in any non-cryptographically secure manner on the client side** — All sensitive data is backend-only and stored using env variables.
100.	**Remove comments in user accessible production code that may reveal backend system or other sensitive information** — React frontend is minified before deployment.

101.	**Remove unnecessary application and system documentation as this can reveal useful information to attackers** — No dev routes or docs exposed.
102.	**Do not include sensitive information in HTTP GET request parameters** — All tokens and credentials are passed via headers or POST.
103.	**Disable auto complete features on forms expected to contain sensitive information, including authentication** — Login and registration fields use autocomplete=off.
104.	**Disable client-side caching on pages containing sensitive information** — No-store and cache-control headers are set.
105.	**The application should support the removal of sensitive data when that data is no longer required** — File uploads, tokens, and expired plans are cleared regularly.
106.	**Implement appropriate access controls for sensitive data stored on the server** — Logs, passwords, and plans are only accessible based on role.
	Communication Security
107.	**Implement encryption for the transmission of all sensitive information** — HTTPS is enforced for all endpoints.
108.	**TLS certificates should be valid and have the correct domain name, not be expired, and be installed with intermediate certificates when required** — SafeNet domains are protected with valid certs.
109.	**Failed TLS connections should not fall back to an insecure connection** — No fallback to HTTP.
110.	**Utilize TLS connections for all content requiring authenticated access and for all other sensitive information** — User data, logs, and scans are all protected over HTTPS.
111.	**Utilize TLS for connections to external systems that involve sensitive information or functions** — All API calls to VirusTotal, IPQS, etc. use HTTPS.
112.	**Utilize a single standard TLS implementation that is configured appropriately** —server + client APIs use standard TLS stack.
113.	**Specify character encodings for all connections** — UTF-8 set in HTTP headers.
	System Configuration
114.	**Ensure servers, frameworks and system components are running the latest approved version** —Node, and Mongo are updated regularly.
115.	**Ensure servers, frameworks and system components have all patches issued for the version in use** — Production environment is patched.
116.	**Turn off directory listings** — Public folders do not expose indexes.
117.	**Restrict the web server, process and service accounts to the least privileges possible** — Backend runs with restricted permissions.
118.	**When exceptions occur, fail securely** — Unexpected errors return generic messages.
119.	**Remove all unnecessary functionality and files** — Dev-only files excluded in production.

120.	**Remove test code or any functionality not intended for production, prior to deployment** — Test routes disabled on deployment.
121.	**Define which HTTP methods, Get or Post, the application will support and whether it will be handled differently in different pages in the application** — Only GET and POST allowed.
122.	**Disable unnecessary HTTP methods** — PUT, DELETE, TRACE are disabled.
123.	**Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks** — Server header is hidden.
124.	**Implement a software change control system to manage and record changes to the code both in development and production** — GitHub used for version control and auditing.
125.	**Isolate development environments from the production network and provide access only to authorized development and test groups** — Dev and prod are separate.
	Database Security
126.	**Use strongly typed parameterized queries** — SafeNet uses MongoDB with secure query structures that avoid raw input injections.
127.	**Utilize input validation and output encoding and be sure to address meta characters** — All user input is validated and escaped before any DB operation.
128.	**Ensure that variables are strongly typed** — Python backend handles typed access to fields, and JSON schema is enforced in some modules.
129.	**The application should use the lowest possible level of privilege when accessing the database** — The database user only has permissions required to perform needed operations.
130.	**Use secure credentials for database access** — Credentials are stored in environment variables.
131.	**Connection strings should not be hard coded within the application. Connection strings should be stored in a separate configuration file on a trusted system and they should be encrypted** — SafeNet uses .env files and keeps them out of version control.
132.	**Close the connection as soon as possible** — Connection pooling and scoped access are used.
133.	**Remove or change all default database administrative passwords** — No default accounts are used.
134.	**Turn off all unnecessary database functionality** — Only essential collections and indexes are enabled.
135.	**Disable any default accounts that are not required to support business requirements** — Only one application account exists per environment.
	File Management
136.	**Do not pass user supplied data directly to any dynamic include function** — SafeNet never evaluates uploaded data.
137.	**Require authentication before allowing a file to be uploaded** — Only logged-in users can upload .eml files.

138.	**Limit the type of files that can be uploaded to only those types that are needed for business purposes** — Only .eml email files are accepted.
139.	**Validate uploaded files are the expected type by checking file headers rather than by file extension** — SafeNet inspects content using libraries.
140.	**Do not save files in the same web context as the application** — Temporary upload directory is isolated from the web application directory.
141.	**Prevent or restrict the uploading of any file that may be interpreted by the web server** — Upload restrictions and type validation prevent executable formats.
142.	**Turn off execution privileges on file upload directories** — No execute permissions are granted to the upload folder.
143.	**When referencing existing files, use an allow-list of allowed file names and types** — Filenames are sanitized, and access is restricted.
144.	**Never send the absolute file path to the client** — All download links and file logs avoid showing internal paths.
145.	**Ensure application files and resources are read-only** — Deployment scripts enforce read-only file access.
146.	**Scan user uploaded files for viruses and malware** — VirusTotal and Hybrid Analysis APIs are used to analyze uploaded files.
	Memory Management
147.	**Utilize input and output controls for untrusted data** — Input validation and output encoding are enforced across all input paths.
148.	**Double check that the buffer is as large as specified** — dynamic typing avoids buffer overflows; boundary checks are added for clarity.
149.	**Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions** — All string input (email, name, URLs) are limited by max length.
150.	**Avoid the use of known vulnerable functions (e.g., eval, exec)** — No use of unsafe built-ins like eval, exec, or raw subprocess calls.
151.	**Properly free allocated memory upon the completion of functions and at all exit points** — garbage collector handles memory safely, and file handles are closed explicitly.
	General Coding Practices
152.	**Use tested and approved managed code rather than creating new unmanaged code for common tasks** — SafeNet uses MongoDB drivers, bcrypt, and other well-maintained libraries.
153.	**Utilize task specific built-in APIs to conduct operating system tasks** — No unsafe system calls are used.
154.	**Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage** — conventions followed throughout the codebase.
155.	**Do not pass user supplied data to any dynamic execution function** — Inputs are validated and not used in dynamic eval, exec, or shell code.
156.	**Restrict users from generating new code or altering existing code** — No form of user script or plugin injection is supported.

157.	**Review all secondary applications, third party code and libraries to determine business necessity and validate safe functionality** — Dependencies are regularly reviewed, and versions are locked.
158.	**Implement safe updating using encrypted channels** — All updates are fetched over HTTPS (GitHub, API vendors).

7. PROJECT SPECIFICATIONS

7.1. System Architecture:

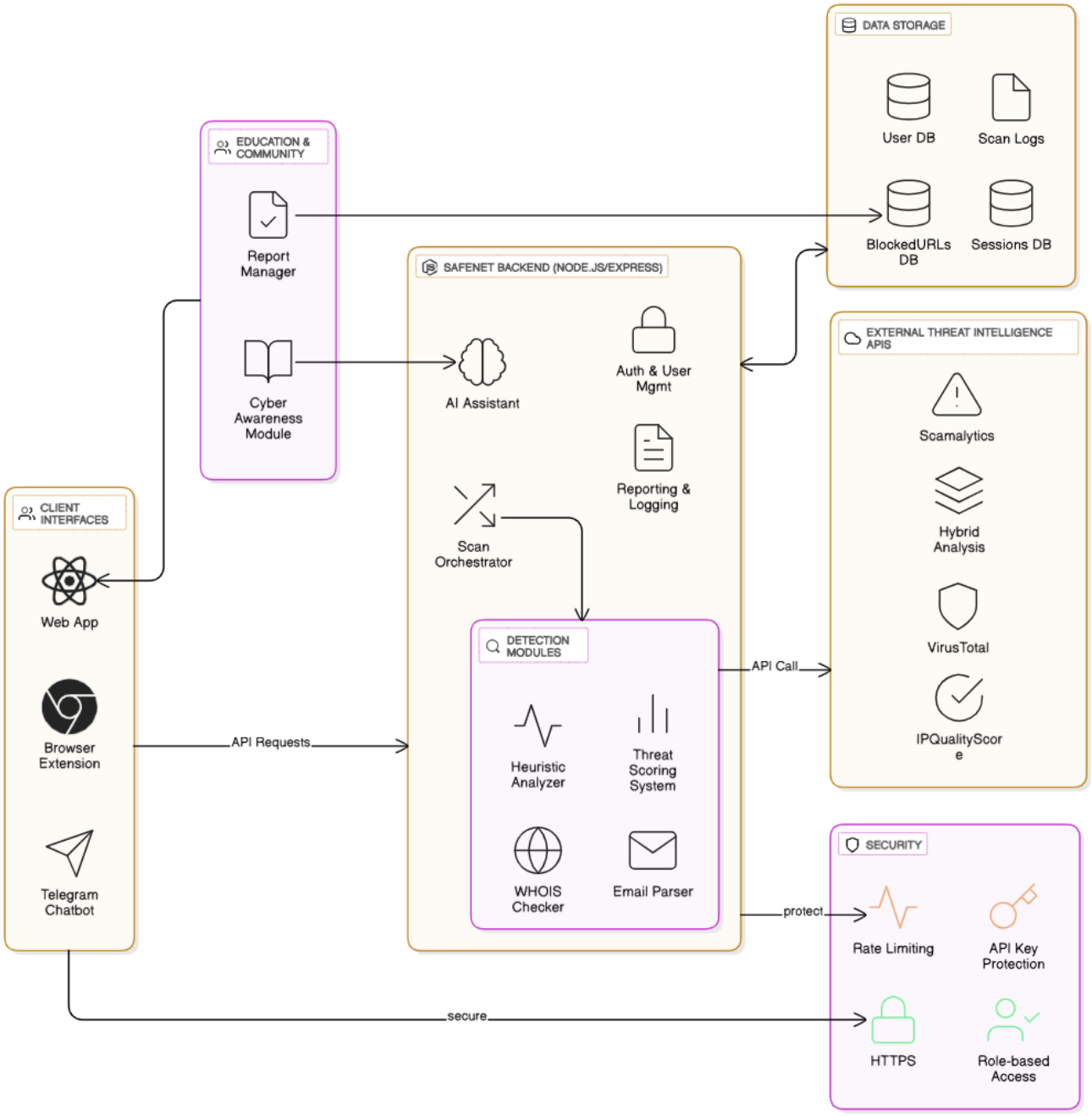


Figure 7.1: System Architecture Diagram

7.2. Stakeholders:

The SafeNet platform involves a diverse range of stakeholders, each contributing to or benefiting from the system in unique ways. Understanding these stakeholders helps ensure the platform is developed, maintained, and used effectively to combat phishing threats.

7.2.1 Internal Stakeholders

- **Development Team:**
  This includes all team members who actively contributed to the planning, design, implementation, and testing of SafeNet. They are responsible for writing secure code, integrating third-party APIs, building the frontend and backend, and ensuring the platform functions as intended. The team also ensures adherence to best practices in cybersecurity and usability.

- **Project Supervisor / Academic Advisor:**
  The academic supervisor oversees the project's progress, provides guidance, ensures academic integrity, and evaluates whether the solution meets the objectives set out at the beginning. They also offer feedback on documentation, technical implementation, and innovation.

- **UI/UX Designer:**
  Responsible for ensuring that the user interface is intuitive, responsive, and user-friendly. They ensure that SafeNet is accessible to both technical and non-technical users, improving the adoption rate and usability of the tool.

- **QA/Testers:**
  Team members who conduct testing and validation to identify bugs, ensure feature completeness, and verify that the platform meets security standards before deployment.

- **Cybersecurity Experts:**
  These professionals are responsible for analyzing phishing patterns detected by the platform and refining detection logic. They play a key role in enhancing the accuracy of threat identification by staying updated with the latest phishing techniques and tactics. Their feedback is vital in adapting SafeNet's heuristic analysis and improving the threat scoring mechanisms, ensuring the platform remains effective against evolving attack methods.

- **Developers & IT Support:**
  The core technical team that builds and maintains the SafeNet platform. They implement new features, optimize performance, and resolve bugs or security vulnerabilities. In addition to maintaining frontend and backend codebases, they also handle integrations with third-party APIs and ensure a seamless user experience across web, browser extension, and Telegram bot interfaces. They also respond to user feedback, rolling out timely updates to meet emerging needs.

- **Database Administrators (DBAs):**
  Responsible for managing and securing the MongoDB Atlas database, DBAs ensure reliable and efficient storage of critical data, including user accounts, scan histories, threat intelligence reports, and flagged content. They handle performance tuning, backup strategies, and data integrity, playing a crucial role in maintaining platform scalability and compliance with data protection best practices.

- **API Providers & Integration Engineers:**
  While technically external, API providers such as VirusTotal, Hybrid Analysis, IPQualityScore, Scamalytics, and WHOIS services function as key data sources. Internally, engineers responsible for API integration ensure smooth communication between SafeNet and these services. They monitor API usage, handle rate limits, and manage premium plans to ensure uninterrupted access to threat intelligence.

7.2.2 External Stakeholders

- **End Users (Individuals and Organizations):**
  These are the primary consumers of SafeNet's services. They use the platform to scan suspicious URLs or upload email files and receive real-time phishing verdicts. End users benefit from clear risk assessments, educational guidance, and chatbot support. Additionally, by submitting phishing reports, they contribute valuable data that enhances the platform's threat detection capabilities.

- **Educational Institutions and Students:**
  Universities, cybersecurity programs, and academic researchers use SafeNet as a practical learning tool. It enables students to explore phishing detection, email analysis, and security awareness in controlled environments. Institutions may also provide academic support, feedback, or collaborative development efforts.

- **Cybersecurity Researchers and Analysts:**
  This group includes ethical hackers, SOC analysts, and independent researchers who benefit from access to scan data, phishing patterns, and threat indicators. They can use SafeNet as a source of real-world intelligence for tracking trends, evaluating techniques, and developing better detection strategies.

- **Open-Source Contributors and Future Maintainers:**
  Developers interested in extending or maintaining the platform rely on scalable architecture and clean documentation. They may enhance SafeNet by adding features (e.g., SMS phishing detection), fixing bugs, or deploying it in new environments. Community contributions help ensure the project's sustainability and adaptability.

- **Companies and Enterprise Security Teams:**
  Businesses use SafeNet to protect employees from phishing attacks by integrating it into their email verification workflows. The platform helps organizations reduce breach risks, conduct internal phishing simulations, and build awareness. It also provides reporting features that can be used for internal audits and training programs.

- **Regulatory and Compliance Officers:**
  Compliance teams use SafeNet to support adherence to frameworks like GDPR, HIPAA, and ISO 27001. By providing features like scan logging, phishing education, and incident reporting, the platform helps demonstrate proactive mitigation of phishing risks and supports audit readiness.

7.3. Functional Requirements

1. User Authentication & Access Control

- Users can create accounts and log in securely.
- Subscription-based access (Standard & Premium plans).

2. URL & Email Scanning

- Users can submit URLs and email files for phishing analysis.
- The system scans URLs using the VirusTotal API (Standard Plan).
- The Premium Plan integrates IPQualityScore API, etc., for advanced analysis.

3. Heuristic Analysis & Threat Detection

- The system analyzes URL structures, domain age, and other phishing indicators.
- WHOIS lookup fetches domain registration details and expiration dates.

4. Chatbot Integration

- Users can scan URLs, messages, or emails via a chatbot in communication platforms.
- The chatbot provides real-time feedback on potential phishing threats.

5. Security Awareness Module

- When a URL is flagged as malicious, users receive educational tips on phishing prevention.

6. Reporting & Threat Intelligence

- Users can report phishing URLs, contributing to an internal database.
- The system logs scan results for historical reference.

7. API Integration & Data Processing

- The system connects with third-party APIs for phishing detection.
- A secure API gateway ensures efficient and safe data exchange.

8. Web-Based Dashboard

- Users can view scan history, reports, and flagged URLs through a dashboard.

9. Browser Extension

- A lightweight browser extension allows users to scan links directly from their websites.
- Results are displayed instantly within the browsing environment for maximum convenience.

10. Email Header Analysis: Parses uploaded email headers to identify anomalies such as:

- Spoofed sender addresses
- Forged "Reply-To" domains
- Unusual "Return-Path" values or inconsistencies in message routing

7.4. Non-Functional Requirements

1. Performance: The system should provide fast phishing detection results, ensuring minimal delay for users when checking links or uploading emails.

2. Scalability: The platform should be capable of handling multiple simultaneous user requests without performance degradation, ensuring smooth operation as the user base grows.

3. Security: The system should be secure, with robust measures in place to protect against unauthorized access, data breaches, and tampering.

4. Usability: The user interface should be intuitive, easy to navigate, and accessible to both technical and non-technical users, ensuring a seamless experience.

5. Reliability: The platform must operate consistently with minimal downtime, ensuring users can access phishing detection and reporting features whenever needed.

6. Compatibility: The application should be functional across different web browsers and devices, including desktops, tablets, and smartphones, to maximize accessibility.

7. Maintainability: The system should be designed for easy updates and improvements, allowing developers to add new features or fix issues without causing disruptions.

8. Extensibility: The system should be designed to support the future addition of new features such as SMS/phishing detection, multi-language support, or integration with corporate systems via APIs.

9. Auditability & Logging: All key actions (e.g., login attempts, scans, admin changes) must be logged securely to support monitoring, debugging, and incident investigation, especially in enterprise use cases.

10. Responsiveness: The user interface should remain responsive under all conditions, including during high-load scenarios. Loading animations, progress indicators, and user feedback should be present to inform users of ongoing processes.

7.5. Class Diagram:

SafeNet's class diagram in Figure 7.2.

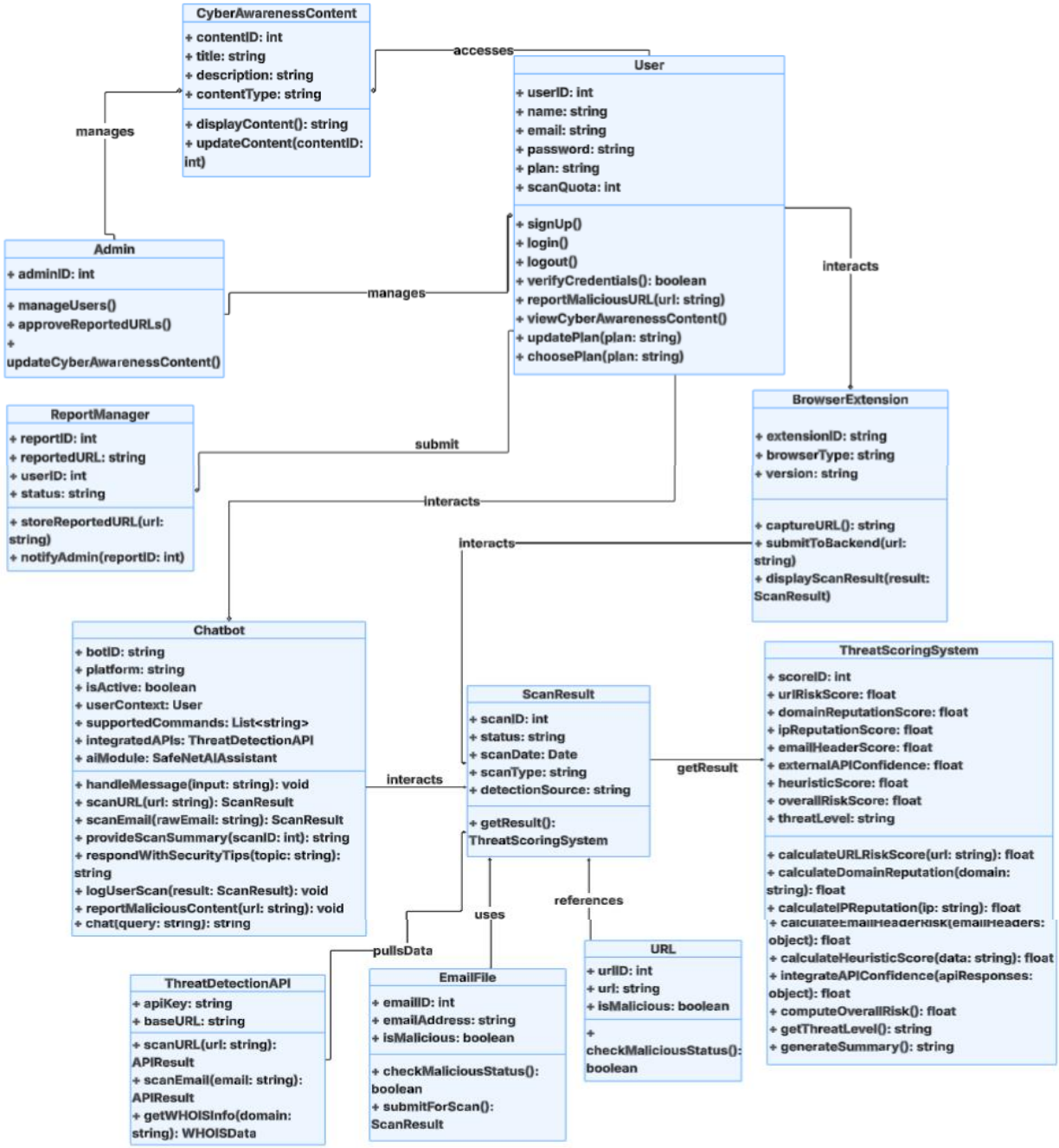


Figure 7.2: Class Diagram

7.6. Use-case Diagram:

SafeNet's use-case diagram in Figure 7.3.

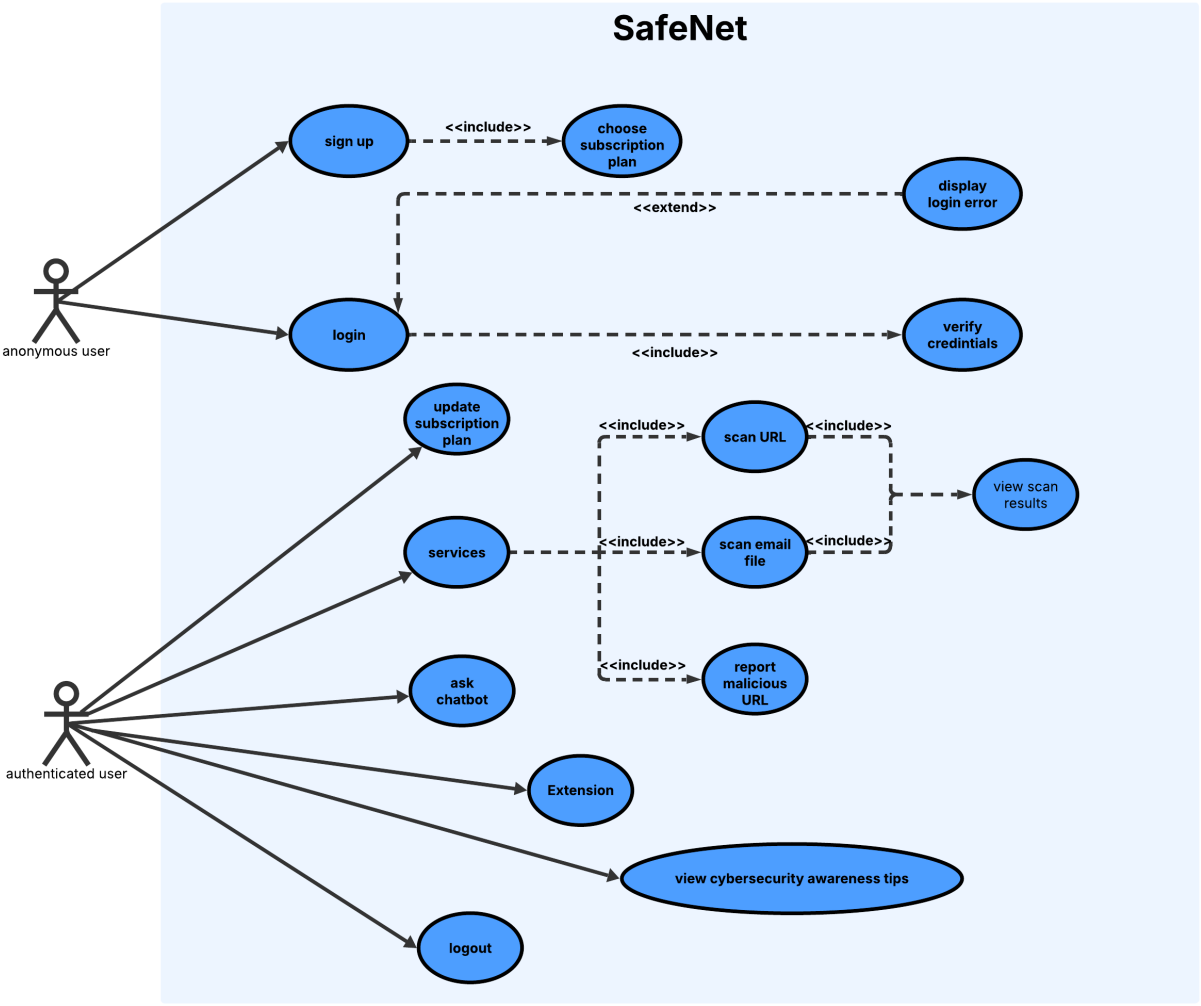


Figure 7.3: Use-case Diagram

7.7. Sequence Diagram:

1. Sign up sequence diagram

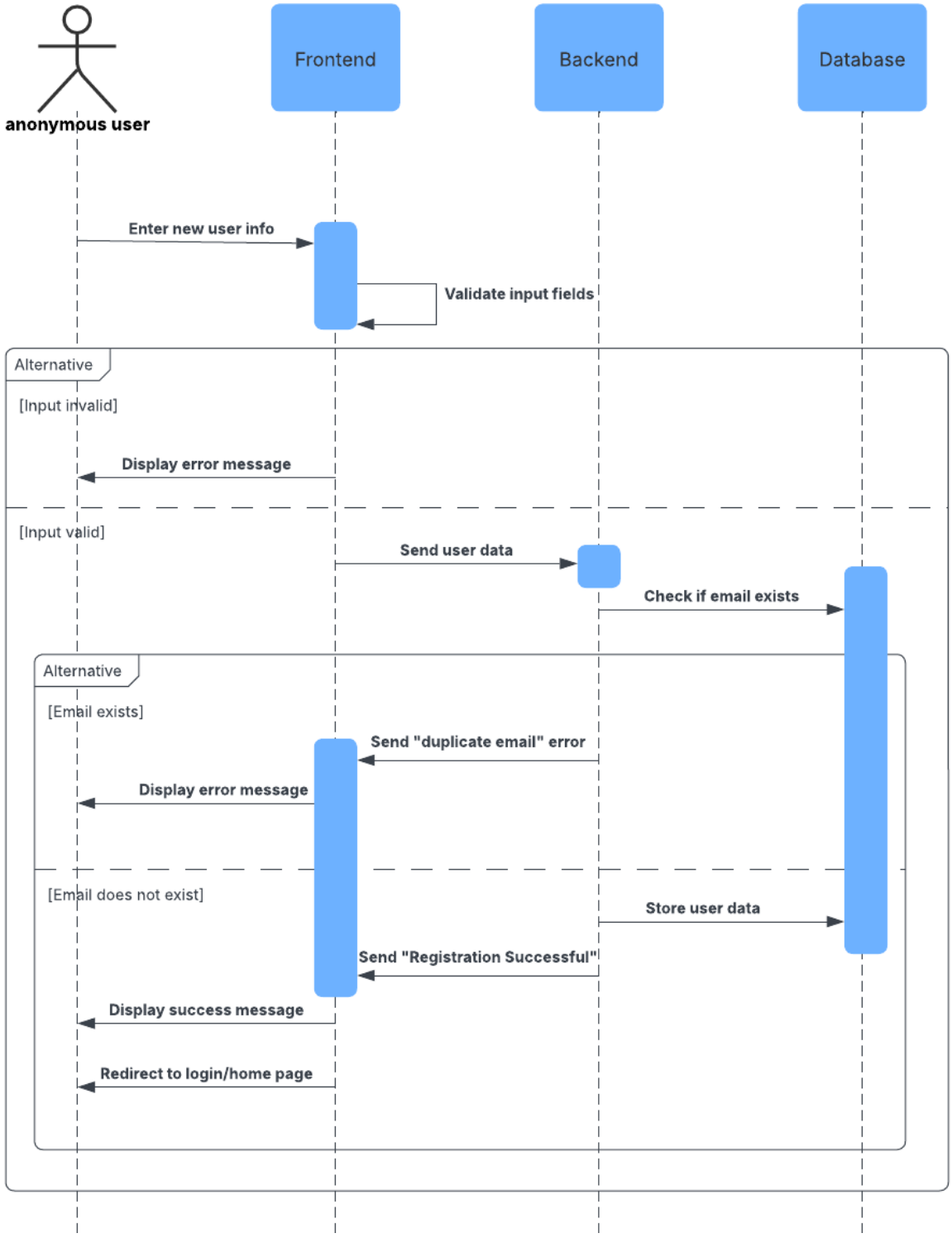


Figure 7.4: Sign Up Sequence Diagram

2. Login sequence diagram:

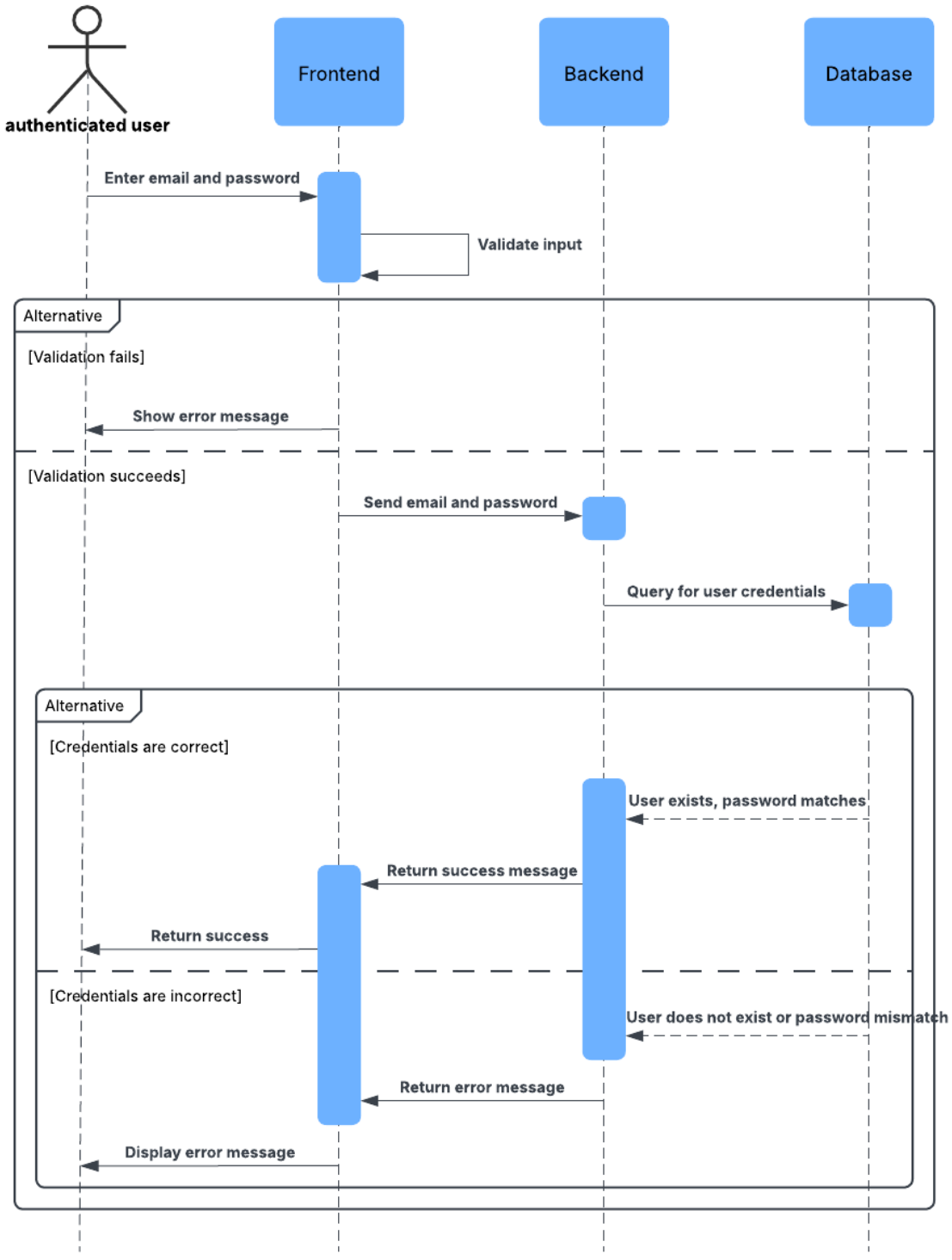


Figure 7.5: Login Sequence Diagram

3. Scan sequence diagram:

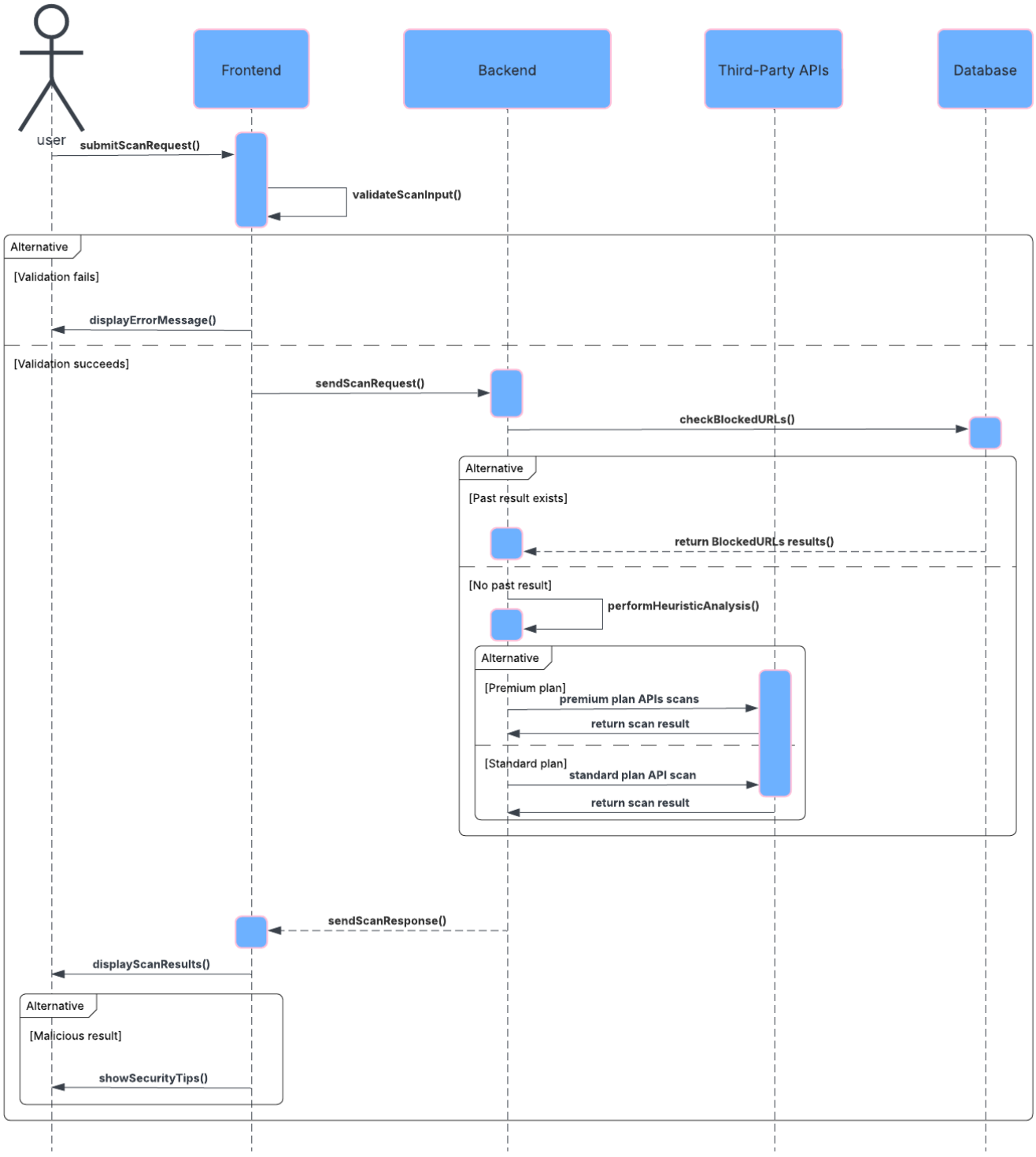


Figure 7.6: Scan Sequence Diagram

4. Report sequence diagram:

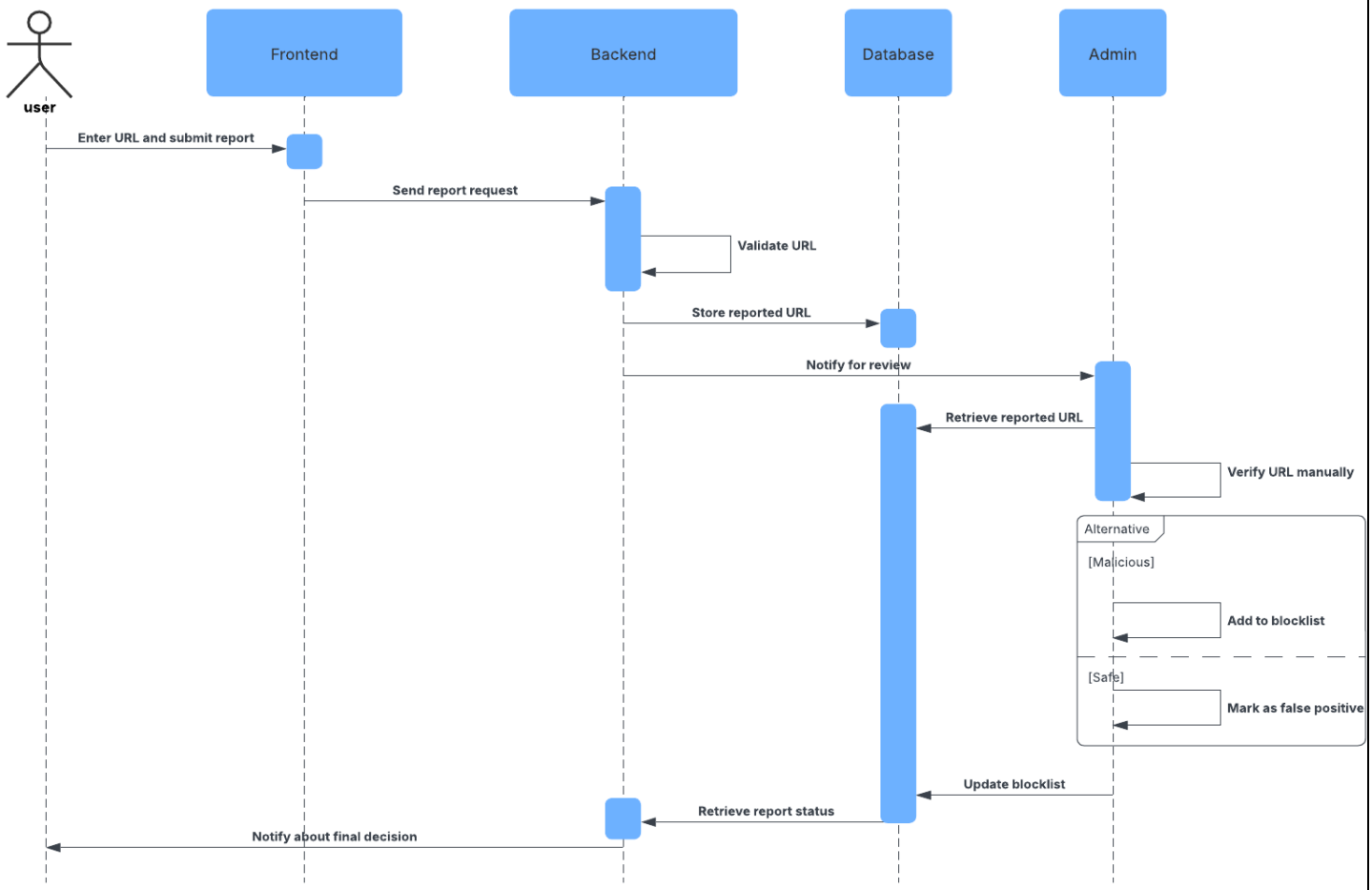
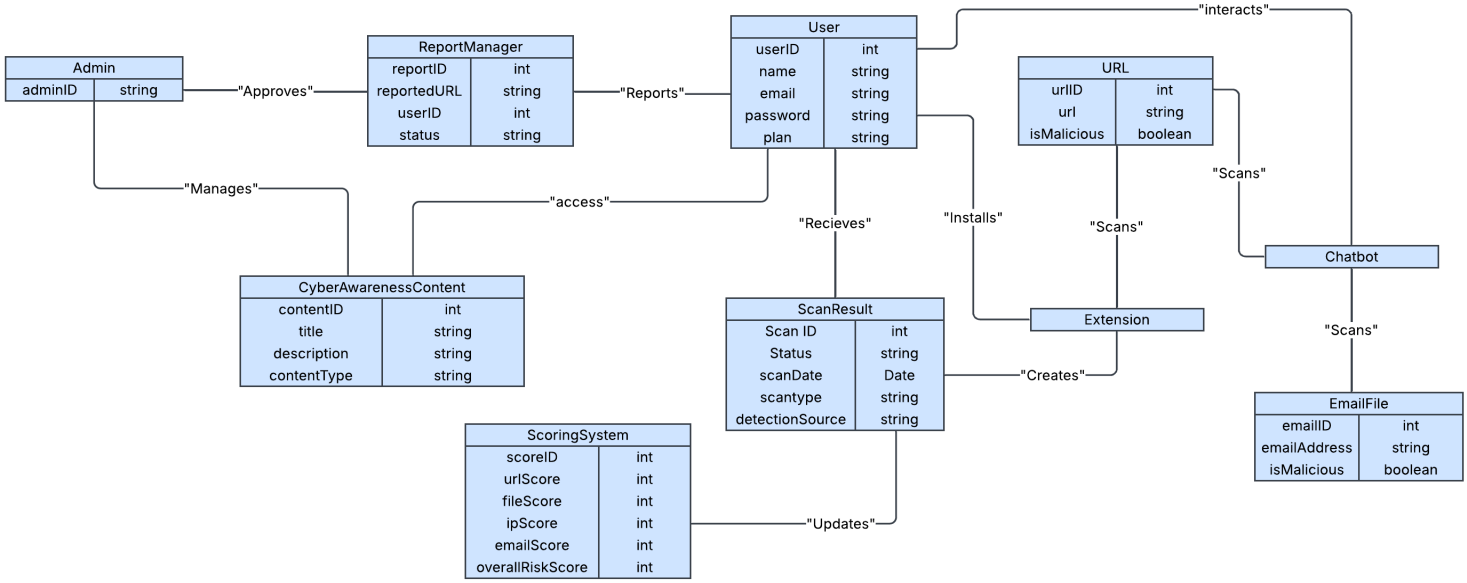


Figure 7.7: Report Sequence Diagram

7.8. Entity Relationship Diagram (ERD):

Figure 7.8: ERD



8. WORK PLAN

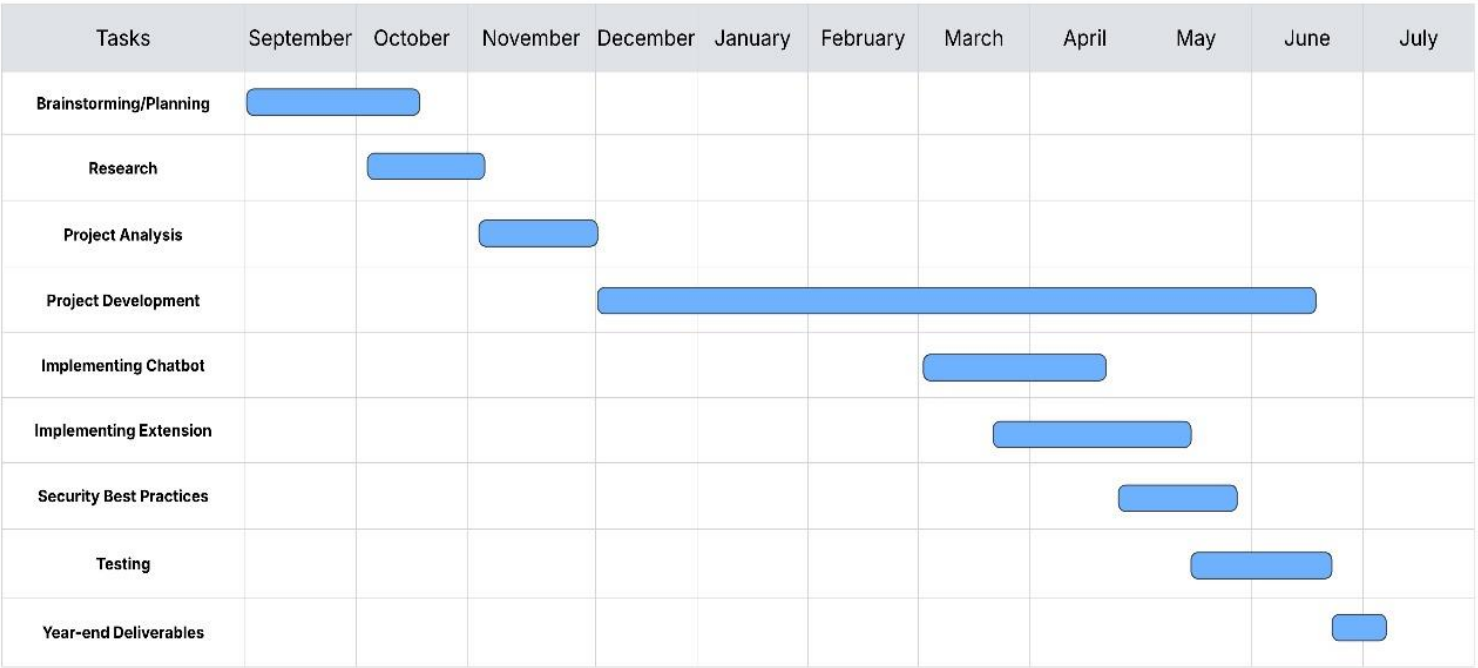


Figure 8.1:SafeNet's Work Plan

1)Brainstorming/Planning

- Brainstorming sessions to discuss potential ideas and select the best idea
- Defining Project scope and objectives
- Setting Work Breakdown Structure
- Assigning roles and responsibilities

2)Research

- Searching for related works
- Study new technologies
- Research on Anti-phishing services, scanning algorithms, security best practices

3)Project analysis

- Analyze the specific phishing threats the project will address
- Evaluate and select third-party security APIs, planning for their integration, data aggregation, and the mitigation of risks such as API downtime or rate limit
- Implement the logic for the custom heuristic analysis

4)Project Development

- Develop the core backend infrastructure using Node.js
- Build the dynamic and responsive client-side application using React
- Develop a backend service to connect with external APIs
- Implementing awareness content

- Ensure the website is responsive and user-friendly.

5)Implementing chatbot

- Defining chatbot functions and scope
- Creating scripts to handle various user intents and queries.
- Building the chatbot interface and integrating it with SafeNet services
- Set up and configure the chatbot integration with the Telegram

6)Implementing Extension

- Defining the extension functions and scope
- Designing extension interface
- Develop the core background scripts to capture URLs

7)Security best Practices

- Implement security measures to protect user data
- Integrate security best practices in website
- Monitor and update security protocols as needed

8)Testing

- Reviewing and testing all services and security measures of the project

9)Year-end Deliverables

- Document all processes, code, and configurations.
- Prepare final reports and presentations.
- Ensure all deliverables are met and hand over the project.

9. FUTURE WORK

While SafeNet meets its current objectives of detecting phishing threats and educating users, there is considerable potential for future enhancements. These improvements aim to increase coverage, accuracy, usability, and scalability of the platform:

- **SMS Phishing (Smishing) Detection:**
  Extend the system's capabilities to analyze suspicious SMS messages. This would involve parsing and evaluating embedded URLs and social engineering patterns often used in mobile phishing attacks.

- **Mobile Application Development:**
  Build dedicated Android and iOS applications to allow users to scan links, upload emails, receive alerts, and interact with the chatbot directly from their smartphones, improving usability and real-time protection.

- **AI-Based Threat Prediction & Auto-Learning Engine:**
  Incorporate machine learning models trained on phishing patterns, user reports, and historical scan data. This would enable proactive threat detection, even for zero-day phishing attacks not yet registered in external APIs. [4]

- **Browser Extension Enhancements:**
  Add features such as auto-alerts on risky websites, passive scanning in real time, and integration with browser bookmarks/history to identify suspicious browsing behavior.

- **Automated Incident Response Actions:**
  Integrate automated workflows for organizations—such as alerting security teams, blacklisting domains, or generating incident tickets—when a phishing attempt is confirmed.

- **Gamified Cybersecurity Training:**
  Develop an engaging training module that uses gamification (quizzes, badges, phishing simulations) to reinforce user knowledge and improve cybersecurity habits in a fun and interactive way.

10. CITATION AND REFERENCING

[1] CSO Online. (2020). *Over 80% of phishing sites now target mobile devices*. Retrieved from: https://www.csoonline.com/article/3545654/over-80-of-phishing-sites-now-target- mobile-devices.html

[2] DataProt. (2023). *Phishing statistics*. Retrieved from: https://dataprot.net/statistics/phishing-statistics/

[3] Frontiers in Computer Science. (2021). *Phishing Website Detection using Machine Learning*. Retrieved from: https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2021.563060/full

[4] Arsen.co. (n.d.). *Phishing Detection – A Comprehensive Guide*. Retrieved from: https://arsen.co/en/blog/phishing-detection

[5] OWASP Foundation. (2023). *Secure Coding Practices – Quick Reference Checklist*. Retrieved from https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/02-checklist/05-checklist

[6] VirusTotal. (n.d.). *VirusTotal API v3 Documentation*. Retrieved from https://docs.virustotal.com/reference/overview

[7] IPQualityScore. (n.d.). *IPQS API Documentation*. Retrieved from https://www.ipqualityscore.com/documentation

[8] Hybrid Analysis. (n.d.). *API Documentation*. Retrieved from https://hybrid-analysis.com/docs/api/v2

[9] Scamalytics. (n.d.). *IP and Domain Risk Intelligence*. Retrieved from https://scamalytics.com/

[10] Telegram. (n.d.). *Telegram Bot API*. Retrieved from https://core.telegram.org/bots/api

[11] MongoDB. (n.d.). *MongoDB Atlas Documentation*. Retrieved from https://www.mongodb.com/docs/atlas/

[12] JWT.io. (n.d.). *JSON Web Token Introduction*. Retrieved from https://jwt.io/introduction

[13] IP-API. (n.d.). *Free Geolocation API*. Retrieved from http://ip-api.com/docs

[14] Google. (n.d.). *Safe Browsing – Protecting users from phishing and malware*. Retrieved from https://safebrowsing.google.com

[15] Microsoft. (n.d.). *Microsoft Defender SmartScreen*. Retrieved from https://learn.microsoft.com/en-us/windows/security/operating-system-security/virus-and-threat-protection/microsoft-defender-smartscreen/

[16] Mozilla. (n.d.). *Phishing*. Retrieved from https://infosec.mozilla.org/guidelines/phishing.html

[17] Meta AI. (2024). *Gemini Flash 1.5 Technical Overview*. Retrieved from https://ai.google.dev/models/gemini

[18] GitHub Docs. (n.d.). *GitHub Version Control*. Retrieved from https://docs.github.com/en/get-started

[19] Express.js. (n.d.). *Express Web Framework*. Retrieved from https://expressjs.com

[20] Postman. (n.d.). *Postman API Platform*. Retrieved from https://www.postman.com/product/api-client/

[21] Visual Studio Code. (n.d.). *VS Code Docs*. Retrieved from https://code.visualstudio.com/docs

[22] Mongoose. (n.d.). *Mongoose ODM Documentation*. Retrieved from https://mongoosejs.com/docs/guide.html