Product Requirements Document: Container Image Vulnerability Scanner

1. Introduction 📝

This document outlines the product requirements for a new security feature that scans container images for known vulnerabilities. The goal is to provide users with a clear understanding of their security posture related to container images and enable them to efficiently prioritize and remediate vulnerabilities.

2. Goals 🎯

- Enable users to identify vulnerabilities within their container images.
- Allow users to understand the severity of identified vulnerabilities.
- Help users prioritize which images require immediate attention and remediation.
- Provide actionable information for fixing vulnerabilities.
- Offer a scalable solution for users managing thousands of container images.

3. Target Users 👥

- **Security Operations (SecOps) Teams:** Responsible for the overall security posture of the organization, including containerized applications.
- **DevOps Engineers/Developers:** Responsible for building, deploying, and maintaining containerized applications. They need to understand and fix vulnerabilities in their images.

4. User Stories 📖

- **As a SecOps Engineer, I want to see a dashboard overview of all my container images and their vulnerability status (e.g., number of critical, high, medium, low vulnerabilities) so that I can quickly assess the overall risk.**
- **As a SecOps Engineer, I want to be able to filter and sort my container images based on vulnerability severity, image name, repository, or last scanned date so that I can focus on the most critical issues.**
- **As a SecOps Engineer, I want to receive notifications or alerts for newly discovered critical or high vulnerabilities in my images so that I can respond promptly.**
- **As a DevOps Engineer, I want to see a detailed list of vulnerabilities for a specific container image, including the vulnerable component, version, and the fixed version, so that I can understand how to remediate it.**
- **As a DevOps Engineer, I want to understand the severity of each vulnerability (Critical, High, Medium, Low) so that I can prioritize my remediation efforts.**
- **As a DevOps Engineer, I want to easily identify which base image or application dependency is causing a particular vulnerability so that I can update the correct component.**
- **As a user with thousands of images, I want the system to perform scans efficiently and display results in a manageable way so that I am not overwhelmed and can quickly find what I need.**
- **As a user, I want to be able to mark vulnerabilities as "false positives" or "accepted risk" with a justification so that they don't clutter my view of actionable items.**
- **As a user, I want to see a history of scans for an image so that I can track remediation progress over time.**

5. Functional Requirements ⚙️

- **Image Scanning:**
  - Integrate with common container registries (e.g., Docker Hub, AWS ECR, Azure CR, Google Artifact Registry) to discover and scan images.
  - Ability to manually trigger scans for specific images or repositories.

    - Scheduled scanning of configured repositories/images.
    - Scan images for known vulnerabilities in OS packages and application dependencies (e.g., Node.js, Python, Java libraries).
    - Utilize up-to-date vulnerability databases (e.g., CVE databases, NVD).
- **Vulnerability Display & Management:**
    - **Dashboard View:**
        - Summary statistics: Total images, total vulnerable images, count of vulnerabilities by severity (Critical, High, Medium, Low).
        - Trend charts showing vulnerability counts over time.
        - List of most vulnerable images.
    - **Image List View:**
        - Table displaying all scanned images.
        - Columns: Image Name, Repository, Tags, Last Scanned Date, Number of Critical Vulnerabilities, Number of High Vulnerabilities, Number of Medium Vulnerabilities, Number of Low Vulnerabilities.
        - Filtering options: by severity, repository, image name, vulnerability presence.
        - Sorting options: by any column.
        - Bulk actions (e.g., rescan selected images - future).
    - **Image Detail View:**
        - Image metadata (name, tag, digest, base image, layers).
        - Summary of vulnerabilities by severity for the selected image.
        - Detailed list of vulnerabilities:
            - Vulnerability ID (e.g., CVE-2023-XXXXX) with a link to the CVE details.
            - Severity (Critical, High, Medium, Low).
            - Vulnerable Component/Package Name.
            - Vulnerable Version.
            - Fixed Version(s) if available.
            - Brief description of the vulnerability.
            - CVSS score.
        - Ability to view scan history for the image.
        - Option to mark a vulnerability as a false positive or accepted risk (with justification).
- **Reporting & Notifications:**
    - Basic reporting on vulnerability status (exportable to CSV/PDF - future).
    - Email notifications for new critical/high vulnerabilities found in designated images or repositories.
- **User Management (Basic):**
    - Role-based access control (Admin, User - future).

6. Non-Functional Requirements 🚀

- **Performance:**
    - Dashboard and image lists should load within 3-5 seconds, even with thousands of images.
    - Image scan times should be reasonable (specific benchmarks to be defined based on image size and complexity).
- **Scalability:**
    - The system should be able to handle scanning and data management for tens of thousands of container images.
- **Usability:**
    - Intuitive and easy-to-navigate user interface.
    - Clear and concise presentation of vulnerability information.
- **Security:**
    - Secure storage of API keys and credentials for accessing container registries.
    - Protection against common web application vulnerabilities (e.g., XSS, CSRF).
- **Reliability:**
    - Scanning processes should be robust and fault-tolerant.

- Accurate vulnerability data from trusted sources.

7. Success Metrics 📊

- Time to identify critical/high vulnerabilities in a new image.
- Percentage of critical/high vulnerabilities remediated within X days of detection.
- User engagement: Daily/weekly active users.
- User satisfaction (measured via surveys or feedback).
- Reduction in the average number of vulnerabilities per image over time.
- Number of images actively being scanned.

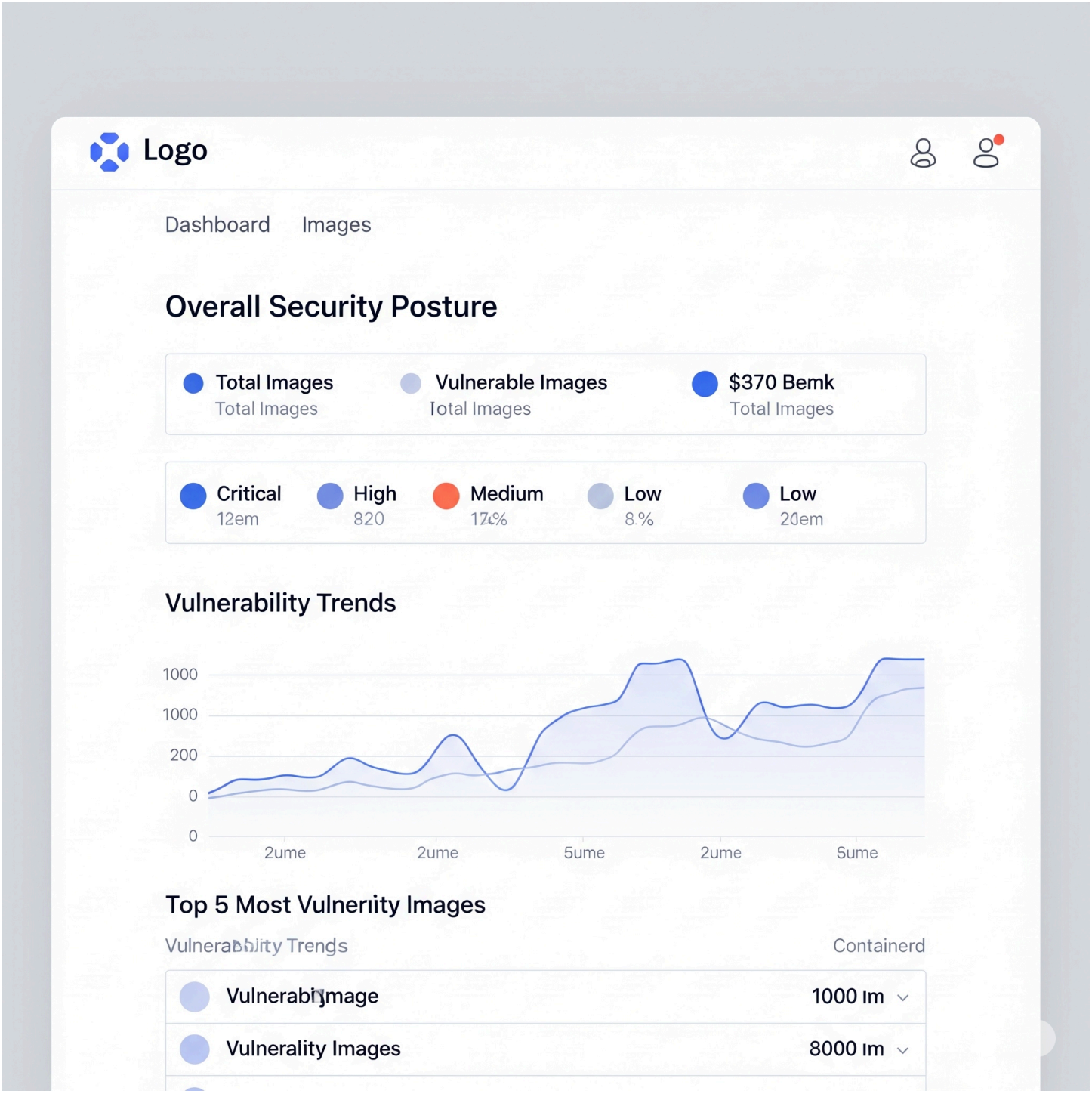
8. Future Considerations (Out of Scope for MVP) 🔮

- Automated policy enforcement (e.g., block deployments of images with critical vulnerabilities).
- Integration with CI/CD pipelines for automated scanning during the build process.
- Advanced reporting and analytics.
- Remediation advice beyond just listing fixed versions (e.g., suggested Dockerfile changes).
- SBOM (Software Bill of Materials) generation and management.

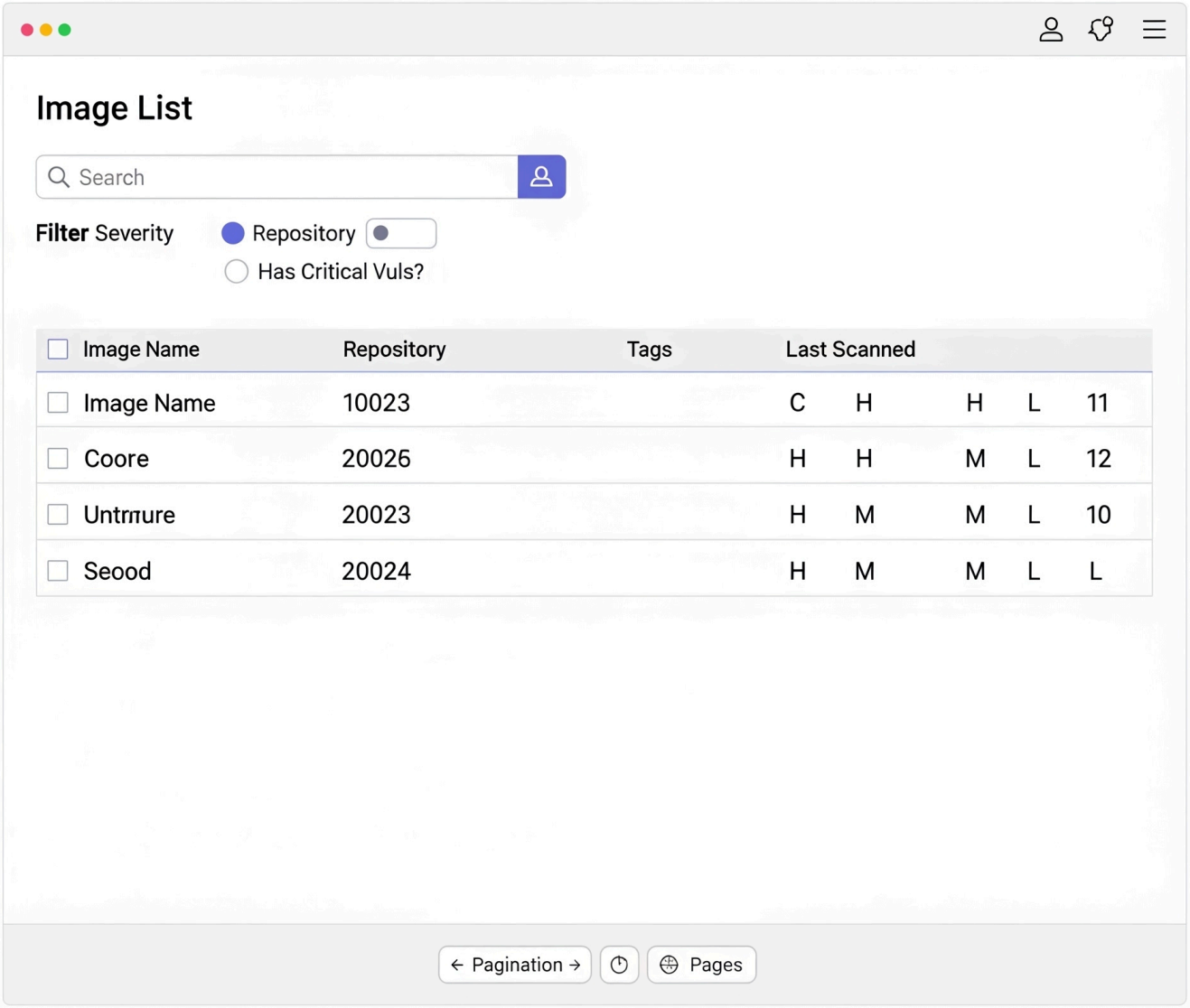
Low-Fidelity Wireframes 🖼️

Here are some low-fidelity wireframes for the key screens. These are basic sketches to illustrate the layout and core elements.

1. Dashboard View

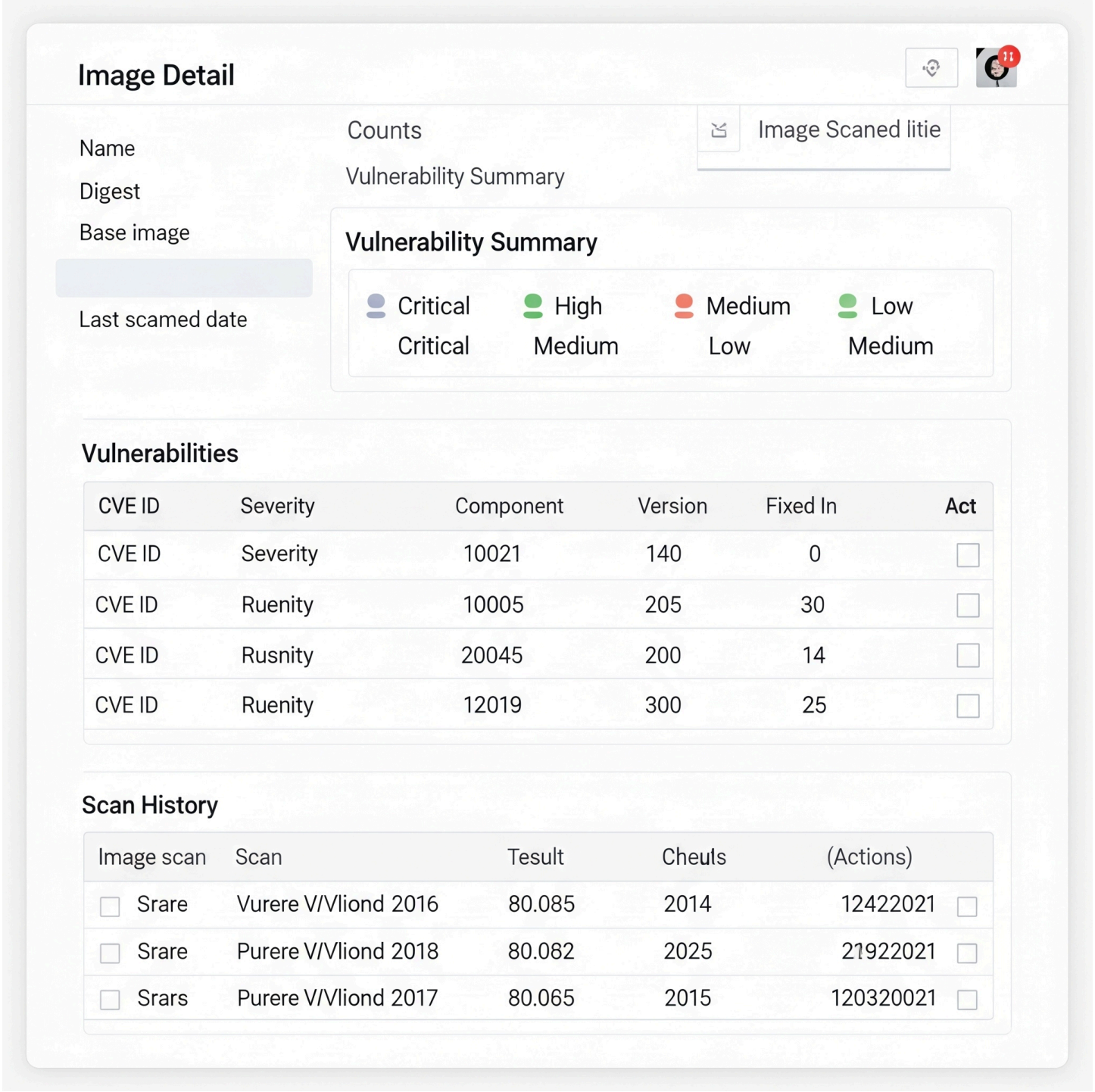


2. Image List View



- *C, H, M, L columns show counts of Critical, High, Medium, Low vulnerabilities.*

3. Image Detail View



Development Action Items 🛠️

Here's a breakdown of development action items that can be discussed with the development team:

Phase 1: Core Scanning & Viewing (MVP)

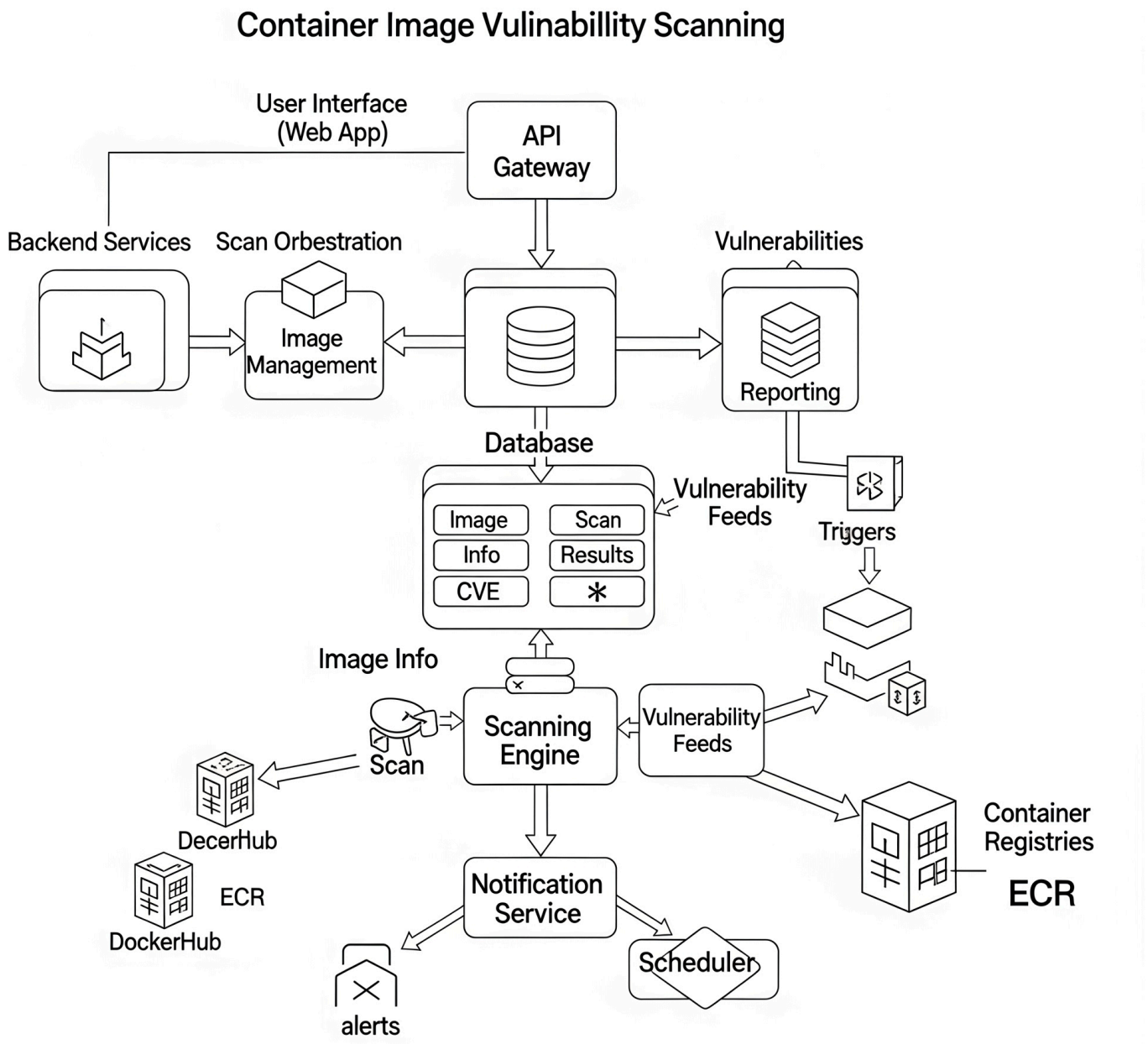
1. **Backend Development:**
   - **Image Metadata Ingestion:**
     - Develop service to connect to specified container registries (e.g., start with one like Docker Hub or ECR).
     - Implement logic to list images and tags.
     - Store image metadata (name, tag, digest, layers, creation date).
   - **Scanning Engine Integration:**

        - Choose and integrate an open-source (e.g., Trivy, Grype) or commercial vulnerability scanner.
        - Develop a service to trigger scans for images (pull image, run scanner).
        - Parse scanner output (JSON/structured format).
    - **Vulnerability Data Storage:**
        - Design database schema to store vulnerability information (CVE ID, severity, component, version, fixed version, CVSS, description) linked to images and scan results.
        - Store scan history.
    - **API Development:**
        - API endpoint for dashboard summary (total images, vulnerable counts).
        - API endpoint for image list with pagination, filtering (severity, name, repo), and sorting.
        - API endpoint for detailed vulnerability information for a specific image.
        - API endpoint to trigger an on-demand scan for an image/repository.
    - **Vulnerability Database Management:**
        - Mechanism to regularly update the vulnerability database used by the scanner.
2. **Frontend Development:**
    - **UI Framework Setup:**
        - Choose and set up a frontend framework (e.g., React, Vue, Angular).
    - **Dashboard Implementation:**
        - Develop components to display overall stats and charts as per wireframe.
        - Integrate with backend API for dashboard data.
    - **Image List Page Implementation:**
        - Develop table component to display images.
        - Implement client-side or server-side pagination, filtering, and sorting.
        - Integrate with backend API for image list data.
    - **Image Detail Page Implementation:**
        - Develop components to display image metadata, vulnerability summary, and detailed vulnerability list.
        - Implement filtering for vulnerabilities by severity.
        - Integrate with backend API for image detail data.
    - **Basic Navigation & Layout:**
        - Implement the main navigation (Dashboard, Images).
3. **Infrastructure & Deployment:**
    - **CI/CD Pipeline Setup:** Basic pipeline for building and deploying backend and frontend.
    - **Database Setup & Configuration.**
    - **Scanner Deployment & Configuration.**
    - **Initial cloud infrastructure setup (e.g., compute instances, object storage for image layers if needed during scan).**

Phase 2: Enhancements & Usability

1. **Backend Development:**
    - **Scheduled Scanning:** Implement a scheduler (e.g., cron job) to trigger scans automatically based on user configuration.
    - **Notification Service:**
        - Develop a service to send email notifications for new critical/high vulnerabilities.
        - User preferences for notifications.
    - **False Positive/Risk Acceptance:**
        - API endpoints to mark vulnerabilities as false positives or accepted risks with justification.
        - Modify vulnerability fetching logic to exclude these as per user settings.
    - **Support for More Registries:** Add connectors for other popular container registries.
2. **Frontend Development:**
    - **Notification UI:** Interface for configuring notification preferences.
    - **False Positive/Risk Acceptance UI:**

    - Add controls on the Image Detail View to mark vulnerabilities.
    - Visual indicators for marked vulnerabilities.
  - **Improved Filtering/Searching:** More advanced search capabilities (e.g., search by CVE ID).
  - **UI for Scan History:** More detailed view or interaction with scan history.
3. **Testing & QA:**
  - Develop comprehensive test cases (unit, integration, E2E).
  - Performance testing with a large number of images and vulnerabilities.
  - Security testing of the application itself.



Discussion Points with Development Team:

- **Choice of Vulnerability Scanner:** Pros and cons of different open-source/commercial scanners. Integration effort.
- **Database Choice:** Scalability and query performance considerations.

- **API Design:** RESTful principles, authentication/authorization strategy.
- **Scalability Approach:** How to design for thousands of images and frequent scans (e.g., message queues for scan jobs, read replicas for database).
- **Definition of "Efficiently Display Results":** Performance targets for UI rendering, API response times.
- **Prioritization of Registries:** Which container registries to support first.
- **Complexity of "Fix" Information:** How detailed should the remediation guidance be initially? (e.g., just fixed version vs. suggesting commands).
- **Effort Estimation for each item.**
- **Technology Stack preferences/constraints.**