AI Assessment I cases - Maicon Kevyn

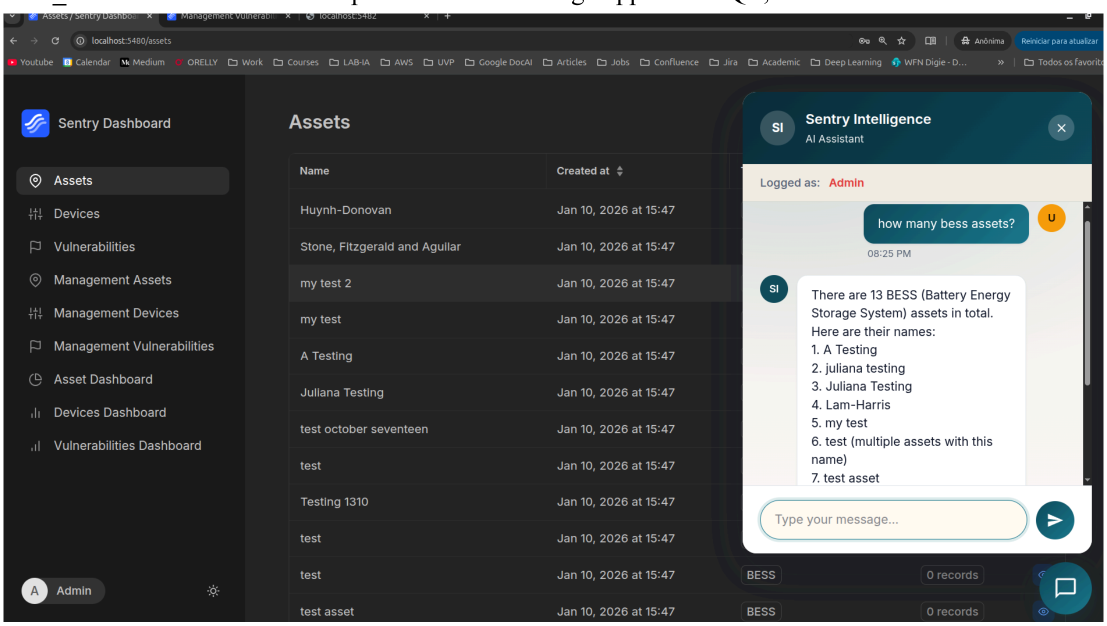
AI Assessment I - Chat AI Dashboard Application
Maicon Kevyn
January 2026

Case 1: RBAC (Operator vs Admin)

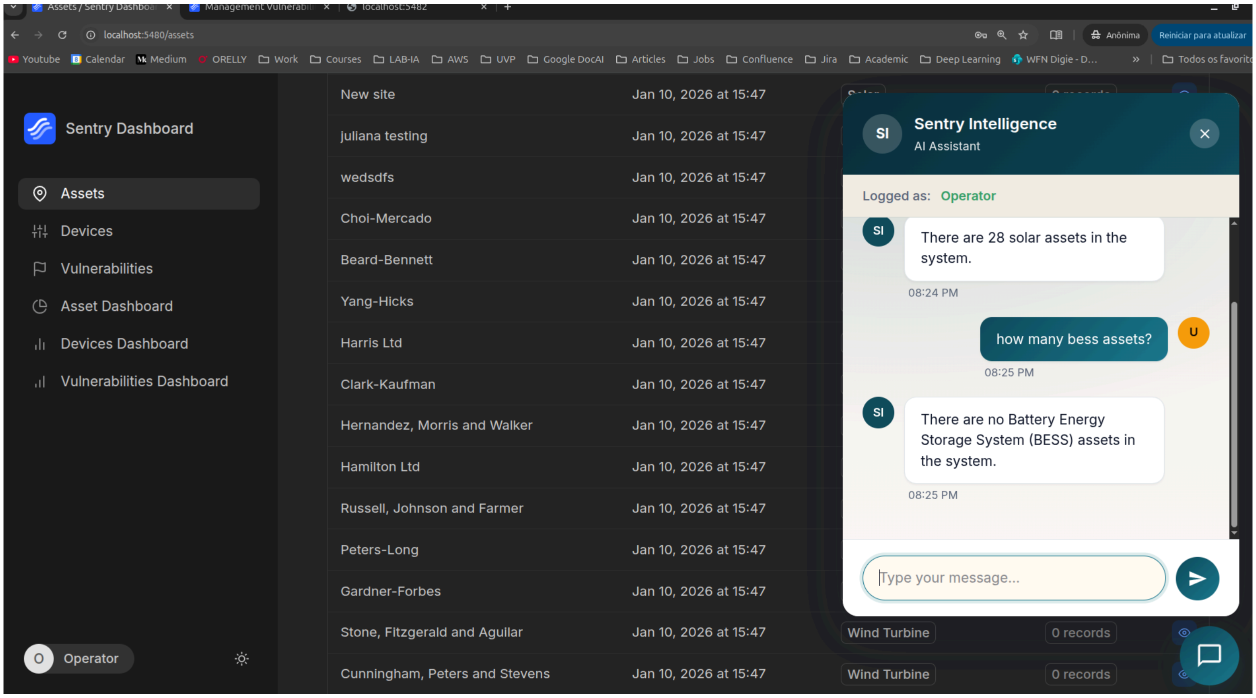
The chat answers structured questions by calling database tools and executing SQL queries against the dashboard PostgreSQL.

In this example ("How many bess assets do we have?"), the assistant returns the exact count and a partial list based strictly on the stored records. No content is fabricated; the response is assembled from tool results.

Role-based access control is enforced at the database query level. With the operator role, the same question returns no bess assets, while admin returns 13 assets. The difference is intentional: assets of type battery (BESS) and wind_offshore are hidden from operators. The filtering happens in SQL, so restricted data never reaches the mode

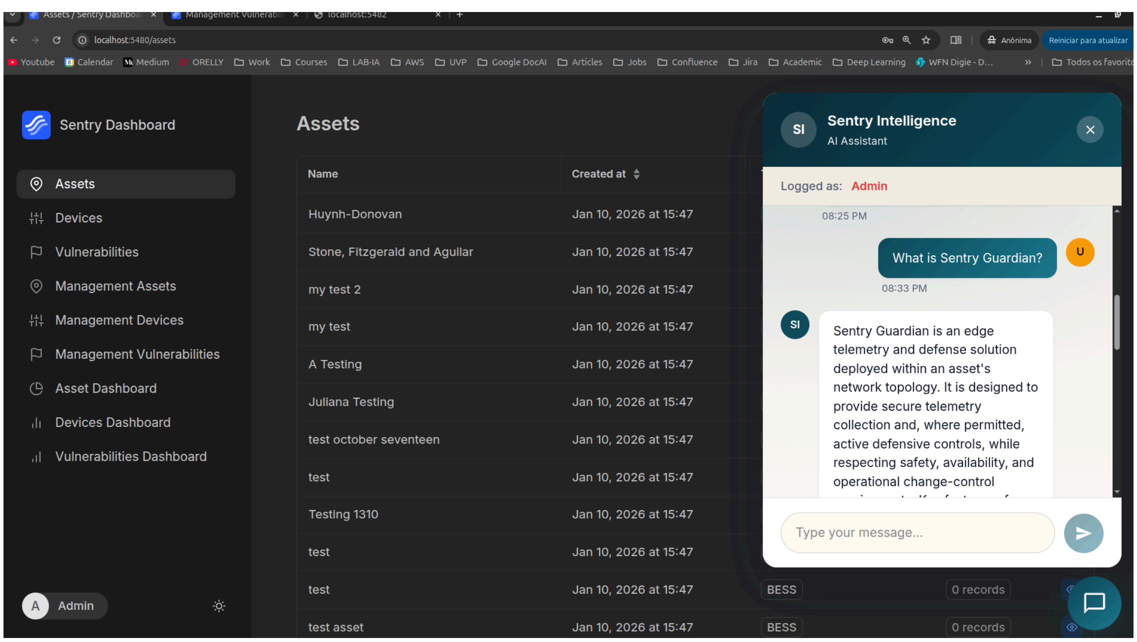


1.

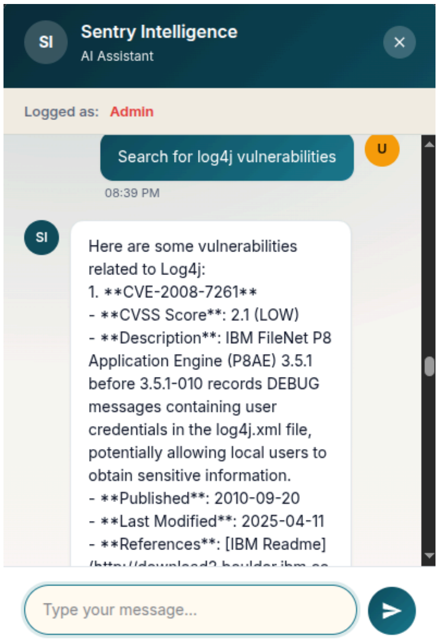


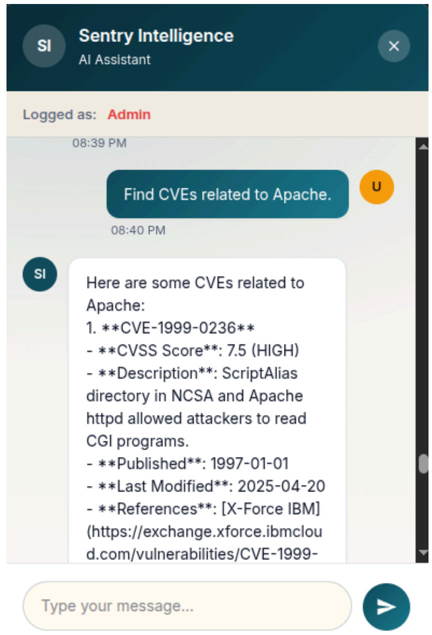
Case 2: Documentation Query

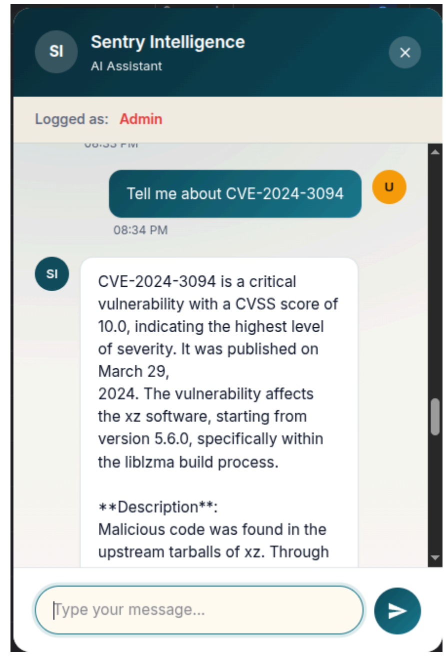
Documentation questions are handled through RAG (Retrieval-Augmented Generation). The assistant retrieves relevant chunks from the indexed PDFs—Business Overview and Dashboard User Manual—and answers using those sources only. The examples shown ("What is Sentry Guardian?" and "How do I filter assets in the dashboard?") come directly from the indexed documents.



Case 3: CVE Search via NVD Only

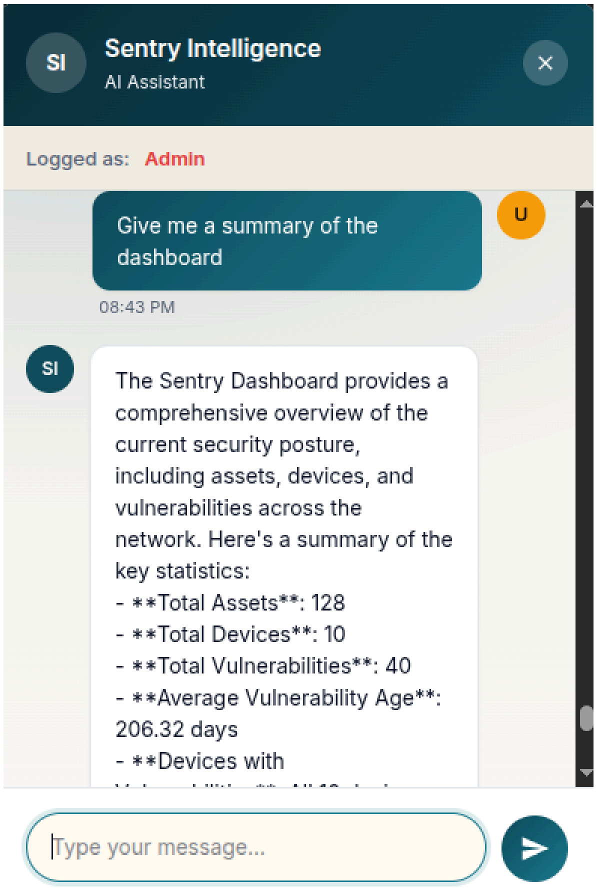


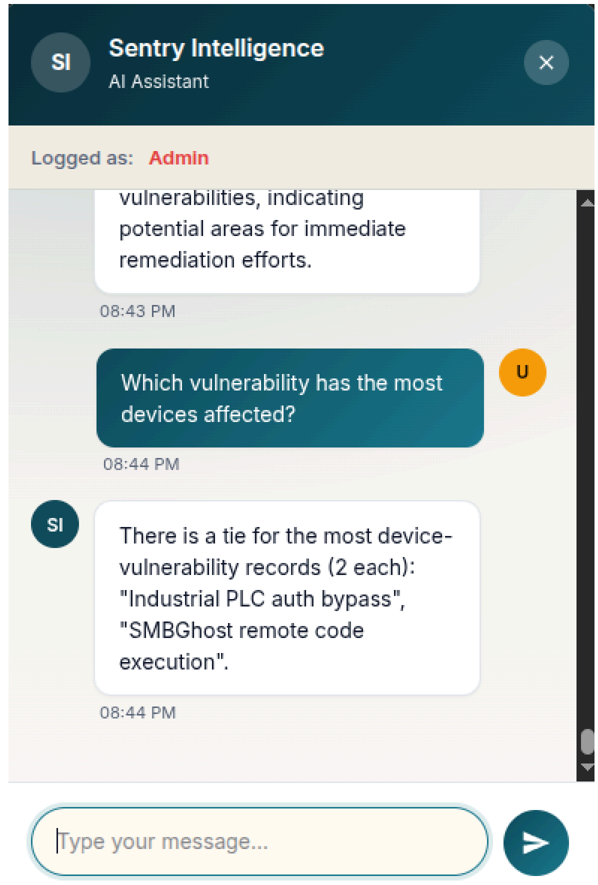




CVE lookups are performed exclusively through the National Vulnerability Database (NVD) API, as required. The assistant does not use any other external source. Results are cached in PostgreSQL for 7 days to reduce latency and respect NVD rate limits. The screenshot “Tell me about CVE-2024-3094” demonstrates a direct CVE ID lookup; an optional keyword search (e.g., “log4j”) shows the same NVD-only path.

Case 4: Dashboard summary (multi-tool query)



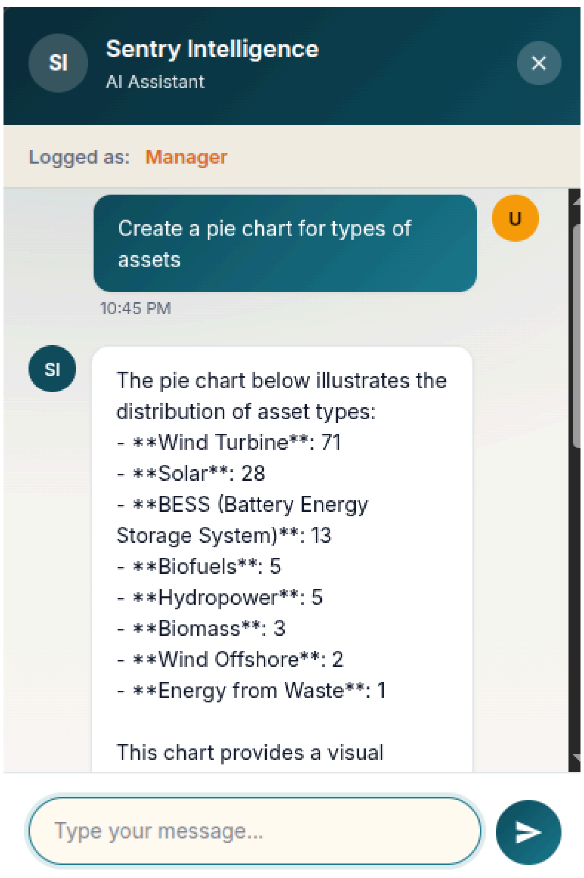


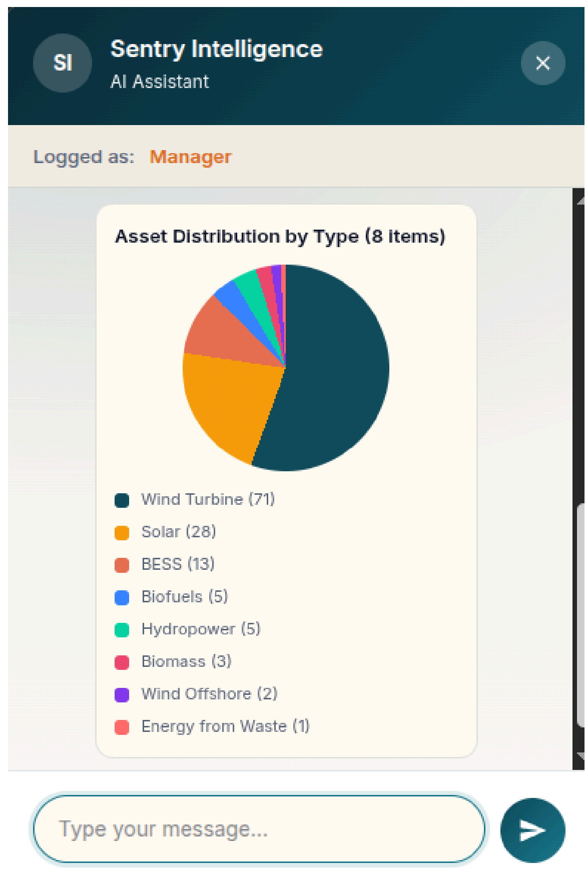
Case 5: Bonus generation (Bonus)

When a user requests data visualization (e.g., "show me a pie chart of assets by type"), the system detects chart keywords and routes the request through a two-phase workflow.

First, the assistant calls a specialized aggregation function like get_asset_distribution_by_type(), which executes SQL GROUP BY to return complete, accurate counts for all categories without pagination. Second, the LLM calls generate_chart() with the aggregated data, specifying chart type (bar, pie, or line), title, and formatted label-value pairs.

The backend validates the configuration and returns it to the frontend, where the chat widget renders the chart inline using CSS and SVG. RBAC filters are applied at the SQL level, so operators and admins see different distributions based on their access rights. This approach avoids pagination artifacts and ensures 100% accuracy in visualizations.





Technical Implementation Summary

Architecture Overview

The Chat AI is implemented as a microservice using FastAPI (Python), integrated with the existing dashboard via Docker Compose. It employs a hybrid architecture combining OpenAI Function Calling for structured database queries, RAG (Retrieval-Augmented Generation) for document search, and external NVD API integration for CVE lookups.

Core Technologies

- **LLM:** OpenAI GPT-4-turbo with function calling for intelligent query routing and multi-tool orchestration
- **Embeddings**: text-embedding-3-small (1536 dimensions) for cost-effective semantic search
- **Vector Database**: pgvector extension on PostgreSQL with HNSW indexing for sub-50ms retrieval
- **Backend**: FastAPI with async request handling and CORS middleware
- **Database**: PostgreSQL with role-based query filtering at the SQL level
- **Frontend**: Vanilla JavaScript chat widget with Chart.js-style rendering (injected via script tag)

Function Calling vs RAG: Database queries use OpenAI function calling (10 custom tools) for precise SQL execution, while unstructured documentation uses RAG. This hybrid approach ensures accuracy for structured data and semantic relevance for documents.

RBAC Enforcement: Role-based access control is implemented at the database query level using SQL WHERE clauses that filter based on JWT-decoded user roles. Operator users are restricted from viewing BESS and Wind Offshore asset types. This backend-enforced approach prevents client-side bypass attempts.

Chart Generation Workflow: Implements a two-phase approach: (1) specialized SQL GROUP BY aggregation functions (get_asset_distribution_by_type, get_device_distribution_by_category) return complete, accurate counts without pagination limits, (2) generate_chart function formats data for visualization. The LLM orchestrates both phases when chart keywords are detected.

CVE Data Source: Uses NVD API exclusively (no local CVE database) as specified in requirements. Implements 7-day caching in PostgreSQL to reduce API calls and respect rate limits (5 requests per 30 seconds without API key).

Token Optimization: Smart query detection bypasses GPT-4 for pattern-matched queries (e.g., "top vulnerabilities by device count"). Temperature set to 0.1 for consistency. Conversation history limited to last 10 messages to control context size.

Security Features

- JWT authentication with role extraction from dashboard tokens
- SQL injection prevention via parameterized queries (no raw SQL from LLM
- CORS configured for specific dashboard origins only
- Rate limiting on chat endpoint (10 requests per minute per user)
- Input sanitization on all function parameters

Data Management

- Document corpus: 48 chunks from 2 PDF files (Business Overview, User Manual)

- Chunking strategy: 1000 characters with 200-character overlap
- Embedding storage: 48 vectors × 1536 dimensions in pgvector
- CVE cache: JSON storage in PostgreSQL with automatic expiration

Limitations and Future Improvements

This implementation was developed as a POC to demonstrate the feasibility of building a chat interface for the Sentry Dashboard. While it successfully fulfills the core assessment requirements, I acknowledge several limitations that would need to be addressed before production deployment.

Most notably: the absence of audit logging for RBAC decisions (critical for cybersecurity compliance and incident response), lack of response streaming (resulting in higher perceived latency and token costs), limited prompt injection protection (a security risk in production environments), no structured error handling (impacting user experience and troubleshooting), and the absence of evaluation metrics to measure system quality.

Specifically, I did not implement metrics to evaluate tool selection accuracy (whether the LLM chooses the right function for each query), RAG chunk relevance (whether retrieved document chunks actually answer the question), or end-to-end response quality (factual accuracy, completeness, and user satisfaction). While I understand how to implement these metrics, using techniques like LLM-as-judge for response evaluation, cosine similarity thresholds for chunk relevance, and confusion matrices for tool selection, time constraints of this POC prevented their implementation. In a production system, these metrics would be essential for continuous improvement and identifying failure modes.

Despite these constraints, I was able to create a functional chatbot using OpenAI's native function calling API without relying on abstraction frameworks like LangChain or LangGraph. This architectural decision provided several advantages: full control over the function execution flow, reduced dependencies and attack surface, lower latency (no framework overhead), and easier debugging of the LLM orchestration logic. However, it also meant implementing features like conversation memory management, retry logic, and tool chaining manually.

The following sections detail each limitation, its impact, and a concrete improvement proposal. This analysis demonstrates not only awareness of the implementation's boundaries but also a clear understanding of the engineering work required to evolve from POC to production-ready system.

Limitation 1: Response Formatting
Current State: Chat responses are plain text without rich formatting. Code snippets, tables, and structured data lack visual hierarchy.

Proposed Improvement: Implement markdown rendering with syntax highlighting (using marked.js and highlight.js). Add support for collapsible sections, data tables, and inline code formatting. Example: vulnerability lists could render as expandable cards with severity badges.

Limitation 2: Limited Conversation History
Current State: Chat history is stored only in browser localStorage and limited to 10 messages. No persistence across sessions or devices.

Proposed Improvement: Create a chat_sessions table in PostgreSQL to persist conversations with user associations. Implement conversation summarization for context window management (e.g., summarize messages 1-5 when reaching token limits). Add UI for browsing past conversations.

Limitation 3: Generic Error Messages
Current State: Errors like "I encountered an error while querying" lack actionable details for users.

Proposed Improvement: Implement user-friendly error classification (e.g., "No assets found matching your criteria" vs "Database temporarily unavailable"). Add retry buttons for transient failures. Log detailed error traces server-side while showing sanitized messages to users.

Limitation 4: Limited Chart Types
Current State: Supports only bar, line, and pie charts. Cannot generate scatter plots, heatmaps, stacked bars, or multi-axis charts.

Proposed Improvement: Extend generate_chart function to support:
- Scatter plots: For correlation analysis (e.g., "Show vulnerability age vs severity")
- Stacked bar charts: For multi-dimensional comparisons (e.g., "Vulnerabilities by severity per asset type")
- Heatmaps: For matrix data (e.g., "Device categories across asset locations")
- Treemaps: For hierarchical data (e.g., "Asset hierarchy with vulnerability counts")

Implementation would require adding Chart.js or Recharts library and extending the data schema.

Limitation 5: Chart Data Export
Current State: Charts are rendered as SVG/Canvas with no export functionality.

Proposed Improvement: Add download buttons for PNG export and CSV data export. Implement via html2canvas for images and dynamic CSV generation for data.

Limitation 6: Read-Only Operations
Current State: Chat cannot perform write operations (create assets, update device status, assign vulnerabilities, delete records).

Proposed Improvement: Implement write functions with multi-step confirmation:
1. Function design: create_asset, update_device_status, assign_vulnerability_to_device
2. Safety mechanism: Require explicit user confirmation before execution ("Are you sure you want to mark Device X as offline?")
3. Audit logging: Log all write operations with user, timestamp, and before/after states
4. RBAC enforcement: Restrict write operations by role (e.g., operators can only update device status, managers can create assets)
5. Rollback capability: Store change history to enable undo operations

Example flow:
User: "Mark device Solar-Panel-001 as offline"
AI: "I found Solar-Panel-001 (ID: abc-123). Marking it offline will:
- Update status from 'online' to 'offline'
- Trigger alert notifications
Confirm? [Yes] [No]"

Limitation 7: No Multi-Asset Operations
Current State: Cannot perform bulk operations (e.g., "Show me vulnerabilities for all Solar assets").

Proposed Improvement: Enhance query functions to accept array parameters and implement batch processing. Add aggregation capabilities for cross-asset analysis (e.g., "Compare vulnerability counts across all Wind farms").

Limitation 8: Limited CVE Intelligence
Current State: CVE lookups return raw NVD data without contextual analysis or recommendations.

Proposed Improvement:
- Cross-reference NVD CVEs with affected devices in the database
- Generate risk assessments based on asset criticality and exposure
- Suggest mitigation actions (e.g., "CVE-2024-1234 affects 3 of your devices. Vendor patch available.")
- Integrate CISA KEV (Known Exploited Vulnerabilities) catalog for prioritization

Limitation 9: Single-Tenant Architecture
Current State: System assumes single organization. No multi-tenancy support.

Proposed Improvement: Add organization_id to all tables and implement tenant isolation at the database level. Use JWT claims to enforce tenant boundaries. This enables SaaS deployment.

Limitation 10: No Caching Layer for Frequent Queries
Current State: Every query hits PostgreSQL, even for common requests like "How many total assets?"

Proposed Improvement: Implement Redis caching layer:
- Cache dashboard stats with 5-minute TTL
- Cache asset/device counts with 1-minute TTL
- Invalidate cache on write operations
- Reduce database load by 60-80% for read-heavy workloads

Limitation 11: Synchronous Processing
Current State: Document indexing blocks API during PDF updates. NVD lookups can timeout for large result sets.

Proposed Improvement: Implement Celery task queue for background jobs:
- Asynchronous document indexing with progress tracking
- Batch CVE enrichment during off-peak hours
- Email notifications for completed long-running task

Limitation 12: No Structured Logging
Current State: Logs are unstructured print statements. Difficult to query or analyze.

Proposed Improvement: Implement structured logging with structlog:
```
log.info("function_called",
     function_name="query_assets",
     user_role="operator",
     execution_time_ms=234,
     result_count=113)
```
Aggregate logs in ELK stack or Datadog for analysis.

Limitation 13: No Performance Metrics
Current State: No visibility into response times, token usage, or cache hit rates.

Proposed Improvement: Implement Prometheus metrics:
- chat_response_time_seconds (histogram)
- openai_tokens_used_total (counter)
- cache_hit_rate (gauge)
- function_call_count (counter by function name)

Create Grafana dashboards for real-time monitoring.

Limitation 14: No User Feedback Loop
Current State: Cannot track response quality or collect user satisfaction data.

Proposed Improvement: Add thumbs up/down buttons to each response. Store feedback in chat_feedback table linked to conversation turns. Use feedback to:
- Identify problematic queries for prompt engineering
- Train fine-tuned models (future enhancement)
- Generate quality metrics for evaluation

Limitation 15: Limited Prompt Injection Protection
Current State: Basic input sanitization. Advanced prompt injection attacks (e.g., "Ignore previous instructions") might bypass filters.

Proposed Improvement: Implement defense-in-depth:
- Input validation with regex patterns for malicious tokens
- Output filtering to detect instruction leakage
- Separate system prompts from user context using OpenAI's message roles strictly
- Implement "guardrails" library for automated prompt injection detection

Limitation 16: No Audit Trail for RBAC Decisions
Current State: When operators are denied access to restricted assets, no audit log is created.

Proposed Improvement: Log all RBAC decisions to access_audit table:
```
CREATE TABLE access_audit (
  id UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  user_role TEXT NOT NULL,
  requested_resource TEXT NOT NULL,
  access_granted BOOLEAN NOT NULL,
  reason TEXT,
  timestamp TIMESTAMPTZ DEFAULT now()
);
```
Essential for compliance and security incident investigation.

Limitation 20: API Key Management
Current State: OpenAI API key stored in environment variables. Single key for all users.

Proposed Improvement:
- Support user-provided API keys (BYOK model) for cost attribution
- Implement key rotation without downtime using secret management (HashiCorp Vault)
- Set per-user token budgets to prevent abuse

Limitation 17: No Automated Testing
Current State: Manual testing only. No unit tests, integration tests, or end-to-end tests.

Proposed Improvement: Implement comprehensive test suite:
- Unit tests: Test each function handler with mocked database (pytest)
- Integration tests: Test OpenAI function calling with recorded responses (VCR.py)
- End-to-end tests: Selenium tests for chat widget functionality
- RBAC tests: Verify operator restrictions across all query types
- Target: 80%+ code coverage

Limitation 18: No Load Testing
Current State: Unknown behavior under concurrent load (e.g., 100 simultaneous users).

Proposed Improvement: Conduct load testing with Locust or k6:
- Simulate 100 concurrent users with realistic query patterns
- Identify bottlenecks (database connections, OpenAI rate limits)
- Establish baseline performance SLIs (e.g., p95 response time < 5s)