

Enhancing Refracted Security Platform with AI Capabilities

Realization document

Alfiya Abdimutalipova
Student Bachelor Applied Computer Science

Table of Contents

1. INTRODUCTION	3
2. ANALYSIS	4
2.1. Development Environment	4
2.2. Artificial Intelligence Components	4
2.2.1. Embedding Model Selection	4
2.2.2. Database Architecture	5
2.3. Design and Prototyping	6
3. IMPLEMENTATION AND DEVELOPMENT	7
3.1. Sidebar Chat Helper	7
3.1.1. Design and User Experience	7
3.1.2. Technical Architecture	10
3.1.3. Challenges and Solutions	11
3.2. Observations Improvement Feature	11
3.2.1. Design and User Experience	11
3.2.2. Implementation Details	13
3.2.3. Challenges and Solutions	14
3.3. Automated Public Data Integration	14
3.3.1. Data Sources and Processing Strategy	15
3.3.2. Architecture and Workflow	15
3.3.3. Challenges and Solutions	16
4. CONCLUSION	17
REFERENCE LIST	19
ATTACHEMENTS	21

1. Introduction

This realization document presents the concrete execution and development process of my internship assignment titled “AI-Powered Enhancements for Vulnerability Management at Refracted Security”. Building on the project plan, this document details how the assignment was approached, analysed, and brought to life.

The internship took place at Refracted Security, a company specialized in providing organizations with tools to effectively manage their cybersecurity vulnerabilities. My assignment focused on integrating artificial intelligence (AI) solutions to enhance several key areas of their platform, including intelligent assessment helpers: an AI-powered sidebar chat assistant and automated improvement suggestions for vulnerability observations.

From the start, it was clear that confidentiality, performance, and the sensitivity of the data were central challenges. Part of my task was not only to develop and prototype the AI solutions but also to investigate the technical and ethical viability of deploying these solutions using local or self-hosted models. I had to consider questions such as: How can we use retrieval-augmented generation (RAG) to enrich local models? Which models are best suited to the specific use cases? And what resources would these models demand in a production environment?

In preparing and executing this project, I applied various technologies and methodologies I had explored during my Applied Computer Science studies, such as RAG architectures, as well as user interface and user experience design. The realization process involved iterative development, research, testing, and continuous feedback — all documented in the following chapters.

This document will cover:

Analysis: A breakdown of the problem, the tools and models evaluated, the challenges identified, and why specific approaches were chosen.

Execution: A detailed description of the development process, the implemented solutions, the design decisions made, and key examples illustrating the results.

Conclusion: A reflection on the outcomes, an evaluation of the objectives versus the results, and recommendations for future development or improvements.

By structuring the document this way, I aim to provide a clear and structured overview of how the project evolved from a conceptual plan to a realized prototype, highlighting both the technical and practical learning outcomes.

2. Analysis

This chapter presents a systematic evaluation of the technological components for the AI-powered vulnerability management platform. The selection process employed a structured decision-making to ensure alignment with the project's core objectives of performance, security, and maintainability while respecting existing company infrastructure. Before starting the actual development work, a thorough analysis was performed to select the most suitable tools, frameworks, and models for the assignment. This phase was essential to ensure that all technological decisions aligned with both the project's goals and the company's existing technology stack.

2.1. Development Environment

After evaluating several integrated development environments (IDEs), including WebStorm and Eclipse, Visual Studio Code (VS Code) was selected as the primary development environment for this project.

This decision was based on the following key factors:

- 1. Comprehensive Language and Framework Support**
VS Code provides excellent support for Node.js, React, and Next.js — the core technologies of the project — along with strong TypeScript and JSX handling, which was essential for frontend development.
- 2. Robust Extension Ecosystem**
The availability of high-quality extensions, such as those for GitHub integration, Docker management, and Azure DevOps pipelines, significantly streamlined the development and deployment workflows.
- 3. Lightweight and High Performance**
Compared to heavier IDEs like WebStorm and Eclipse, VS Code offered a lightweight, fast, and responsive experience, especially valuable when working on large projects and switching frequently between backend and frontend codebases.
- 4. Existing Familiarity**
I was already familiar with using Visual Studio Code from previous development work, which allowed me to start productively right away without the need for additional setup time or training. This prior experience also helped me navigate more advanced features and optimize my workflow effectively.
- 5. Integration with Azure DevOps**
The team used Azure DevOps for version control, CI/CD pipelines, and project management. VS Code's seamless integration with Azure services ensured smooth collaboration, code reviews, and automated deployments.

2.2. Artificial Intelligence Components

2.2.1. Embedding Model Selection

As part of the AI system design, selecting the right embedding model was critical to ensure high-quality semantic search, efficient performance, and cost-effective scalability. To make an informed choice, several leading sentence embedding models were evaluated, including Hugging Face's all-mpnet-base-v2, Hugging Face's all-MiniLM-L6-v2, OpenAI's text-embed-3-large, and OpenAI's text-embed-3-small.

The evaluation was conducted across five key criteria, each weighted according to project priorities:

- Accuracy (30%) — how well the model captures semantic meaning, particularly in the security domain
- Speed (20%) — measured in milliseconds per query, reflecting system responsiveness
- Cost per 1 million tokens (20%) — local models vs. API pricing
- Model Size (15%) — influencing deployment footprint and hardware requirements
- Domain Adaptation (15%) — how effectively the model adapts to security-specific terminology

The comparative results are summarized in the *Appendix A*.

Based on the weighted total score, Hugging Face's all-mpnet-base-v2 emerged as the top candidate. Although all-MiniLM-L6-v2 offered faster inference and a smaller model size, all-mpnet-base-v2 delivered superior semantic accuracy and better domain adaptation, which were critical for the project's focus on security-related semantic search.

Additionally, choosing a local model like all-mpnet-base-v2 provided significant cost savings compared to API-based solutions like OpenAI's text-embed-3 series, eliminating per-token usage fees while maintaining full control over the model pipeline.

Overall, the selection of all-mpnet-base-v2 ensured the project met its goals for performance, scalability, cost-efficiency, and domain specificity, making it the optimal embedding foundation for the AI solution.

This structured evaluation ensured that the chosen model was not only technically sound but also maximally aligned with project requirements and company infrastructure.

2.2.2. Database Architecture

The selection of an appropriate vector database was a crucial step in the architecture of the AI solution, as it directly impacts the system's ability to perform fast and scalable semantic search operations on embedding vectors. To ensure an informed and well-justified decision, a comparative evaluation was conducted among three leading vector database solutions: Qdrant, Pinecone, and Weaviate.

The evaluation process was structured around five weighted criteria:

- Query Speed (30%) — The responsiveness of the system when handling similarity search queries is critical for ensuring a smooth user experience, particularly under real-time or near-real-time conditions.
- Scalability (25%) — The database must be capable of handling increasing data volumes and concurrent requests as the system evolves and scales.
- Security (20%) — Given the sensitive nature of some data, robust security mechanisms, including encryption, access control, and compliance features, were considered non-negotiable.
- Deployment Ease (15%) — A solution that integrates smoothly into the development environment and supports flexible deployment (on-premises, cloud, hybrid) reduces setup complexity and long-term maintenance effort.
- Cost Efficiency (10%) — While not the primary driver, operational cost remains an important consideration to ensure the solution remains financially sustainable.

For details refer to *Appendix B*.

Based on this weighted analysis, Qdrant emerged as the top choice. While Pinecone offered excellent scalability and ease of deployment, Qdrant excelled in query speed, security, and cost efficiency — all critical for the project's performance and budget requirements.

Additionally, Qdrant's open-source nature provided flexibility for customization and integration with existing infrastructure, while maintaining strong security features such as encrypted payloads and role-based access control.

Overall, Qdrant was selected as the optimal vector database, aligning with the project's needs for high-performance similarity search, secure data handling, and scalable, cost-effective deployment.

2.3. Design and Prototyping

For the design phase, I selected Figma primarily because I was already familiar with its features and workflows. Compared to alternatives like Adobe XD or Sketch, Figma offered a familiar environment with strong cloud-based collaboration, reusable design components, and efficient developer handoff.

Using Figma, I developed wireframes, high-fidelity mock-ups, and interactive prototypes, allowing for rapid validation of design ideas and smooth feedback loops with stakeholders. Its built-in developer tools helped ensure consistent, high-quality implementation while speeding up delivery.

3. Implementation and Development

This chapter describes the result of the developed system: the implemented functionalities, the application's design, the architectural principles applied, and the key patterns used throughout the build. The purpose here is to give a clear picture of what has been made — focusing on the most significant features, design decisions, and technical choices. Where relevant, small screen descriptions or high-level technical explanations are provided to illustrate the functioning parts of the system.

Importantly, due to the intellectual property policy of the company where this thesis project was conducted, no direct source code may be shared in this document. While general design approaches and technical explanations are provided, all proprietary code, scripts, and configurations are considered confidential and cannot be reproduced here. This ensures full compliance with the company's confidentiality agreements and policies.

The following major components were developed during the internship:

1. **Sidebar Chat AI-Helper:** A real-time AI assistant integrated into the platform, allowing users to interact with AI models for support in tasks such as querying documentation or generating structured content.
2. **Observations Improvement Feature :** Enhances user-written findings using AI-generated suggestions, including text improvement, translation, mitigation suggestions, and tagging.
3. **Automated Public Data Integration :** An automated pipeline that fetches and processes publicly available cybersecurity datasets (e.g., CVEs, CWEs, CAPECs, MITRE ATT&CK, EPSS scores) to enrich the AI's knowledge base via RAG.
4. **Security and Scalability Measures :** Ensures secure handling of data across multiple tenants and maintains system performance under load.

Below, each component is described in terms of its purpose, how it was designed, which technologies were selected, and how it contributes to the overall system.

3.1. Sidebar Chat Helper

The Sidebar Chat Helper serves as a lightweight, always-accessible AI assistant embedded within the application interface. Its primary purpose is to provide real-time assistance to users without requiring them to leave their current context. Users can ask questions, request summaries, or generate structured responses based on prompts they provide.

This feature was designed with flexibility in mind: it allows users to select different AI models (e.g., GPT-based models hosted on Azure), supports chat history persistence, and integrates securely with the backend systems. Importantly, the platform ensures that analysis of the provided data aligns with the company's strict data privacy requirements.

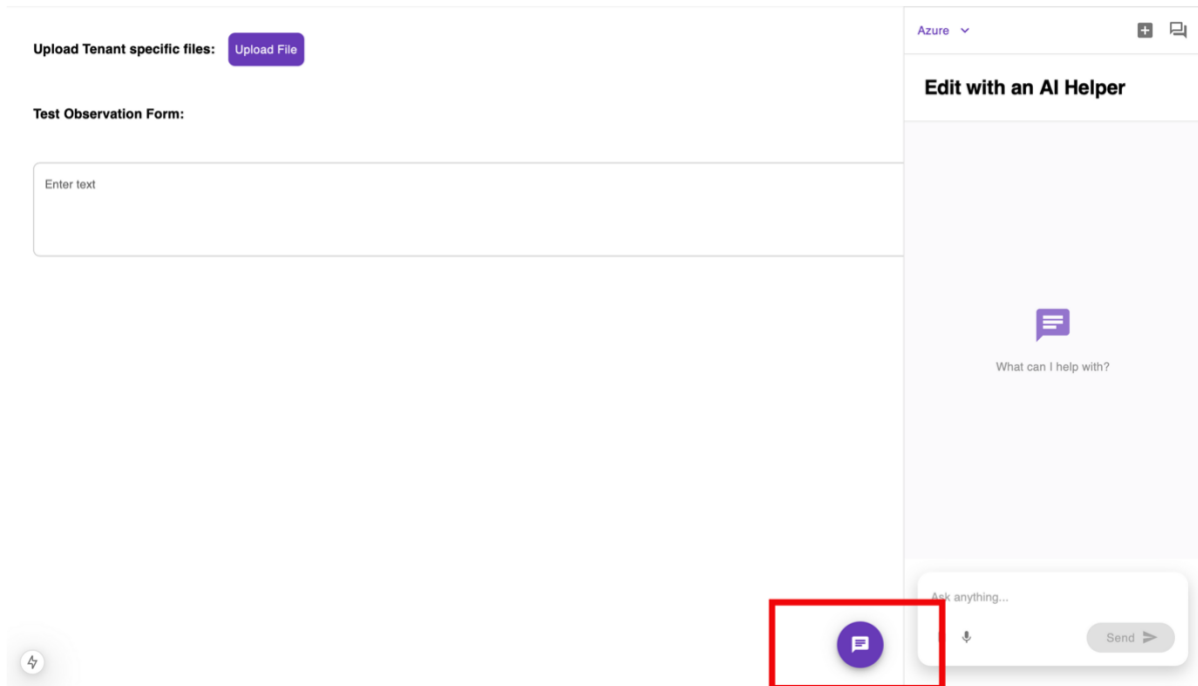
To prevent unnecessary database bloat and ensure efficient resource usage, any documents or contextual files uploaded by the user during a session are stored temporarily in memory and discarded once the browser session ends. This approach ensures that user-specific content does not accumulate in the database over time, preserving storage space and maintaining system performance.

3.1.1. Design and User Experience

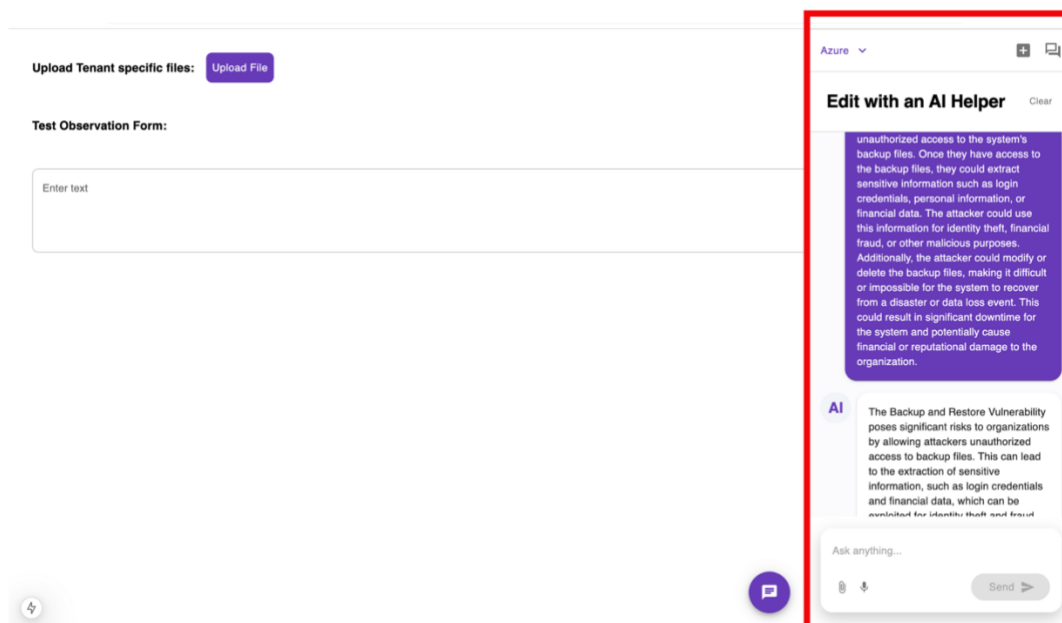
The UI was developed using React and Material UI (MUI), ensuring consistency with the existing platform design language. The chat interface appears as a floating button that expands into a sidebar when clicked.

All styling was aligned with the shared 'theme.ts' file provided by the company, ensuring visual cohesion with other platform components.
Key features of the UI include:

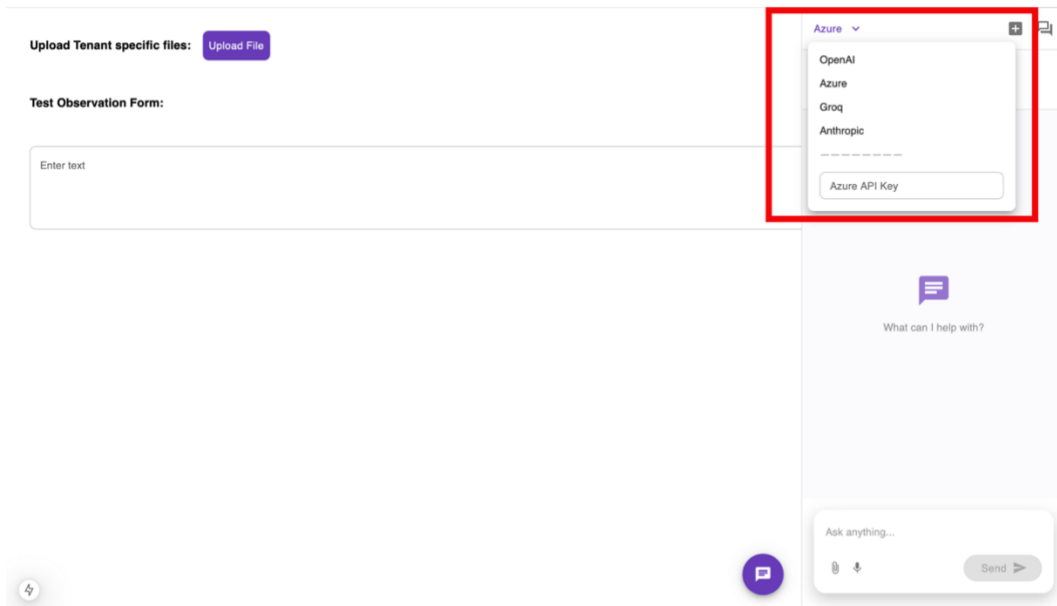
- Floating button that expands into a sidebar chat interface.



- Responsive message bubbles for user and AI messages

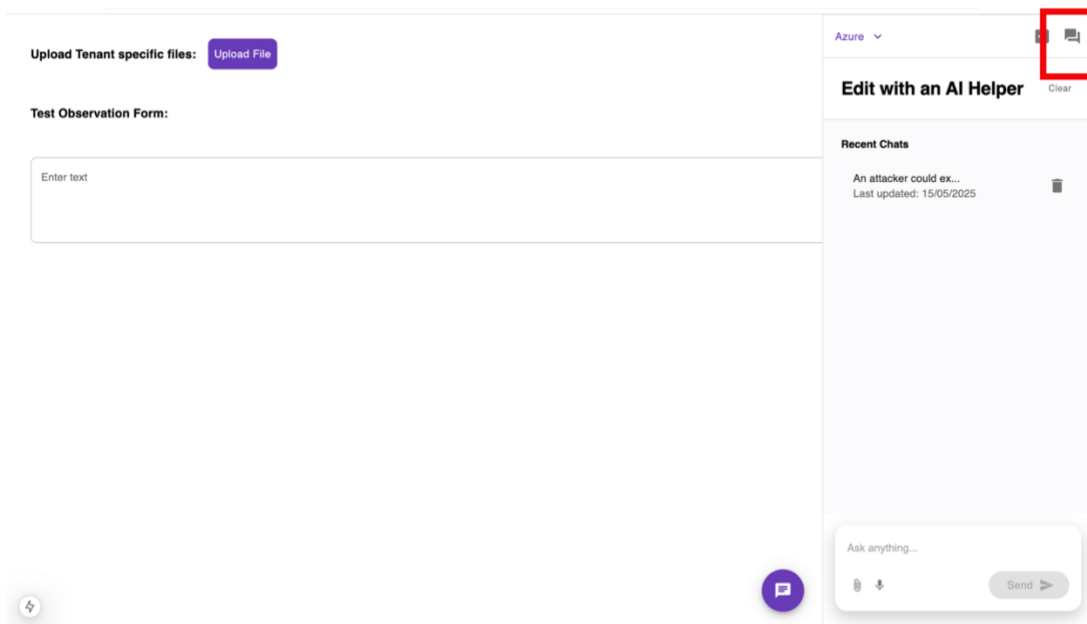


- Dropdown menu for selecting AI models selection (e.g., GPT variants via Azure OpenAI)



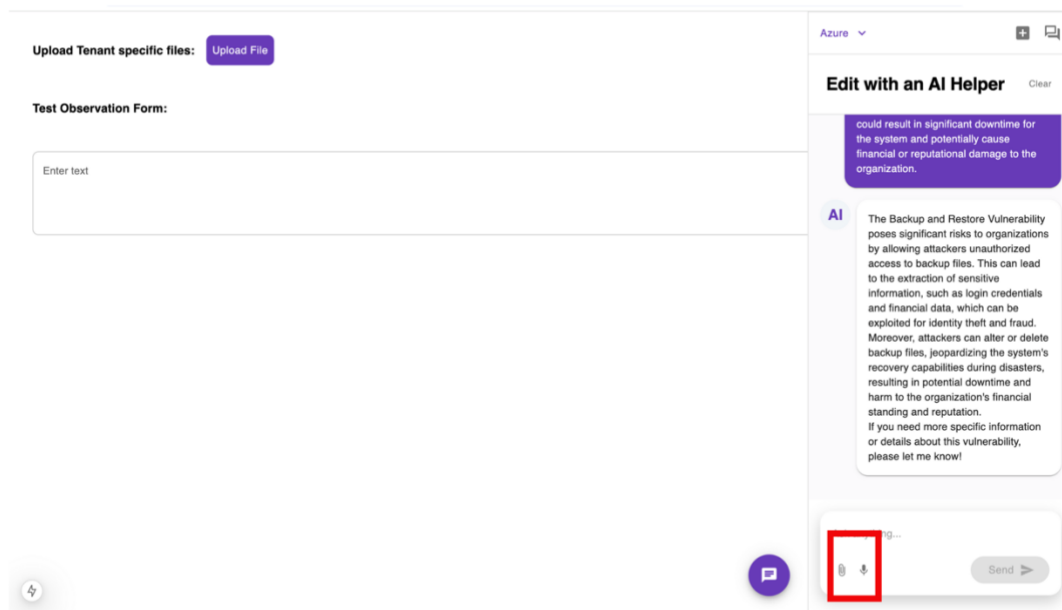
The screenshot shows a web interface for an AI chatbot. On the left, there is a section titled "Upload Tenant specific files:" with an "Upload File" button. Below this is a "Test Observation Form:" with a text input field labeled "Enter text". On the right, there is a chat window. At the top of the chat window, a dropdown menu is open, showing options: "OpenAI", "Azure", "Groq", and "Anthropic". Below these options is a text input field labeled "Azure API Key". The dropdown menu is highlighted with a red rectangle. Below the dropdown, the chat window has a message input area with a placeholder "Ask anything..." and a "Send" button. Above the input area, there is a speech bubble icon and the text "What can I help with?".

- Persistent chat history across sessions.



The screenshot shows the same web interface as the previous one, but the dropdown menu is closed. Instead, the chat window on the right now displays a list of "Recent Chats". The first chat entry is titled "An attacker could ex..." and has a subtitle "Last updated: 15/05/2025". There is a trash icon next to this entry. Above the chat list, there is a header "Edit with an AI Helper" and a "Clear" button. The chat window also has a message input area with a placeholder "Ask anything..." and a "Send" button. Above the input area, there is a speech bubble icon and the text "What can I help with?". The chat history section is highlighted with a red rectangle.

- File Upload and Voice Input Buttons



3.1.2. Technical Architecture

The backend API routes were built using Next.js App Router with TypeScript, providing a modular and type-safe environment. Communication between the frontend and backend leverages secure HTTP APIs, with requests routed through Azure-hosted OpenAI endpoints.

To ensure persistent storage of chat history across sessions, a database layer was introduced using Prisma ORM with a PostgreSQL backend. This allowed for efficient CRUD operations on chat threads and messages, associated with unique user identifiers.

Key technical choices included:

- React hooks for managing local UI state, such as input fields, loading indicators, and message rendering
- Server-side session management to securely track active chats and user authentication status without exposing sensitive tokens or cookies
- Environment variables and secret management to securely store and access API keys and configuration values without hardcoding them into the source files

Additionally, the system was designed with scalability and security in mind:

- All sensitive operations were performed server-side to prevent exposure of internal logic or credentials
- Rate limiting and request validation were enforced at the API level to prevent abuse and ensure system stability
- Session-based cleanup mechanisms ensured that temporary user data (e.g., uploaded documents for context) were automatically discarded after the browser session ended, preventing unnecessary database growth and maintaining system performance

This architecture laid the foundation for a robust, maintainable, and secure implementation of the Sidebar Chat Helper, enabling seamless integration with other platform features while adhering to company standards for data privacy and system efficiency.

3.1.3. Challenges and Solutions

One of the early challenges encountered during the development was ensuring correct formatting and validation of API requests sent to the Azure-hosted OpenAI service. Due to variations in model input requirements and strict schema validations on the server side, improperly structured prompts could lead to failed or inconsistent responses.

This issue was addressed by implementing a multi-layered validation system within the backend API. Before any request was forwarded to the AI service, it underwent content sanitization and format verification. Additionally, comprehensive error-handling logic was introduced to capture and return meaningful error messages to the frontend, allowing for graceful user feedback and preventing application crashes. This became especially important when debugging issues related to incorrect parameters being passed to the Azure-hosted GPT models (as noted during Weeks 4 and 5 of the weekly internship report).

A key shift in the technical implementation occurred when moving from a direct prompt-based model to a LangChain agent-based architecture. Initially, prompts were manually constructed and sent directly to the language model, requiring extensive logic to handle different types of queries. However, as the complexity of interactions increased — including dynamic tool usage and contextual awareness — managing these flows became increasingly difficult.

To address this, I transitioned to using LangChain agents, which allowed the AI to dynamically decide which tools to use based on the user's query. This provided greater flexibility and reduced the need for hard-coded branching logic. While this approach simplified the codebase and improved adaptability, it also introduced some unpredictability in output structure, which required additional post-processing and validation layers to ensure consistency in the final response.

Despite this trade-off, the agent-based model proved valuable for scenarios requiring multi-step reasoning or integration with external tools like vector search or database queries.

These solutions contributed significantly to the stability and reliability, enhancing the overall user experience and ensuring seamless interaction with backend services.

3.2. Observations Improvement Feature

The Observations Improvement Feature assists users in refining and enhancing their written observations — such as vulnerability descriptions and remediation steps — by offering AI-generated improvements. This functionality includes actions like:

- Summarizing lengthy texts
- Rewriting for clarity or conciseness
- Generating example content

Users can accept or reject suggested changes, enabling fine-grained control over the final output. This feature is particularly useful in improving the quality of penetration test reports and ensuring consistency across templates.

3.2.1. Design and User Experience

The feature includes a dynamically appearing AI button that shows up at the end of editable text fields when hovered over. Clicking the button opens a dialog where users can select an action and view the AI-generated result. This contextual integration ensures that users can access AI assistance without leaving their current workflow.

Styling was aligned with the company's shared theme.ts file, ensuring coherence with the rest of the application's UI components.

The feature was tightly integrated into the report-writing interface without disrupting the original functionality. Input validation and sanitization were performed before sending queries to the AI model, preventing malformed or unsafe requests.

- Dialog-based UI triggered by an AI button dynamically placed at the end of editable fields.

The screenshot displays a web interface for a 'Test Observation Form'. At the top, there is a section for 'Upload Tenant specific files:' with an 'Upload File' button. Below this is the 'Test Observation Form' label. A large text input field with a placeholder 'Enter text' is shown. At the right end of this input field, a small, dark button labeled 'AI Assistant' is visible, highlighted by a red rectangular box. The interface also includes a small icon in the bottom left and a purple circular button in the bottom right.

- Dropdown menu with support of multiple actions: summarize, rewrite, generate new content

This screenshot shows the same 'Test Observation Form' interface. In addition to the text input field, there is a 'Type Prompt' input field above it. To the right of the 'Type Prompt' field, a dropdown menu is open, showing the option 'Write Content' selected, which is highlighted by a red rectangular box. The rest of the interface, including the 'Upload Tenant specific files:' section and the bottom navigation elements, remains the same.

- AI generated placeholder suggestion

Upload Tenant specific files: [Upload File](#)

Type Prompt:

An attacker could exploit the Backup and Restore Vulnerability by gaining unauthorized access to the system's backup files. Once they have access to the backup files, they could extract sensitive information such as login credentials, personal information, or financial data. The attacker could use this information for identity theft, financial fraud, or other malicious purposes. Additionally, the attacker could modify or delete the backup files, making it difficult or impossible for the system to recover from a disaster or data loss event. This could result in significant downtime for the system and potentially cause financial or reputational damage to the organization.

- Users have the ability to review, accept, or discard AI-generated suggestions before finalizing their reports.

Financial and Reputational Damage: Prolonged downtime and data loss could result in substantial financial losses and damage to the organization's reputation.

3. Recommendations:

- **Immediate Access Review:** Conduct a thorough review of access controls and permissions associated with backup files to ensure that only authorized personnel have access.
- **Implement Stronger Security Measures:** Enhance security protocols around backup files, including encryption and multi-factor authentication.
- **Regular Audits:** Establish a routine audit process to monitor access to sensitive files and detect any unauthorized attempts.
- **Incident Response Plan:** Develop and implement an incident response plan to address potential breaches and mitigate risks associated with unauthorized access.

Conclusion:

The unauthorized access to sensitive backup files poses a serious threat to the organization's data security and integrity. Immediate action is required to address the identified vulnerabilities and prevent potential exploitation by malicious actors. Regular monitoring and enhancement of security measures are essential to safeguard sensitive information and maintain the organization's operational resilience.

Follow-Up Actions:

- Schedule a meeting with the IT security team to discuss the findings and recommendations.
- Initiate a security assessment to identify and remediate any additional vulnerabilities within the system.

Accept Cancel

3.2.2. Implementation Details

This feature leverages the same AI backend as the Sidebar Chat Helper but uses distinct prompts tailored to each action. Default prompt templates were defined for each function (e.g., "summarize the following text"), but users also have the option to override these templates for more customized outputs.

For example:

- For Summarize, the prompt might be: "Provide a concise summary of the following vulnerability description."
- For Rewrite, the prompt may ask: "Rewrite the following paragraph to improve clarity."

The system combines both the selected action and the user's input to generate relevant responses. Flexibility was a key consideration — users can modify default prompts to suit specific use cases or domain-specific terminology, ensuring AI-generated suggestions remain contextually appropriate.

3.2.3. Challenges and Solutions

One of the early challenges was dynamically positioning the AI button inside various form fields where users could edit vulnerability observations. Since different input components had varying DOM structures and event behaviours, implementing a consistent hover-triggered UI across all elements required careful handling of focus and mouse events.

This was addressed by implementing conditional rendering logic based on DOM events such as `focus`, `hover`, and `click`. A custom React hook was developed to detect cursor position and render the AI button precisely at the end of the text field, ensuring a smooth and intuitive user experience regardless of the input type.

A significant improvement came in Week 9, when I worked on chaining multiple retrievers to enrich AI-generated suggestions with relevant cybersecurity knowledge from public datasets like CVEs, CAPECs, CWEs, and MITRE ATT&CK. However, this introduced new complexities in query formatting and embedding model inference. After troubleshooting these issues and adjusting similarity search parameters, the system began generating more accurate and contextually relevant responses.

In Week 10, I explored strategies for integrating structured numerical data (like EPSS scores) into the RAG pipeline. Since unstructured text was stored in the vector database and structured data needed a separate storage format (e.g., PostgreSQL), I implemented a hybrid search mechanism. This allowed the system to combine semantic similarity results with numerical relevance, significantly improving answer quality and enabling more nuanced filtering.

By Week 11, the hybrid search system was fully integrated and tested end-to-end. The Observations Improvement Feature now returned enriched answers that combined both semantic search results and structured numerical data, offering users more comprehensive and actionable suggestions.

These iterative improvements and problem-solving efforts ensured to deliver high-quality, secure, and scalable enhancements to the platform's reporting capabilities.

3.3. Automated Public Data Integration

To enhance the accuracy and relevance of AI-generated responses, an automated data integration pipeline was developed to ingest and process publicly available cybersecurity datasets. These include:

- CVEs (Common Vulnerabilities and Exposures)
- EPSS Scores (Exploit Prediction Scoring System)
- CAPEC & CWE (Common Attack Pattern Enumeration and Classification / Common Weakness Enumeration)
- MITRE ATT&CK

These datasets serve as a foundational knowledge base for Retrieval-Augmented Generation (RAG), enabling the AI to reference up-to-date security information when generating answers or improving vulnerability reports.

This pipeline ensures that the system remains informed about emerging vulnerabilities and attack techniques, providing real-time contextual support for both the Sidebar Chat Helper and the Observations Improvement Feature.

3.3.1. Data Sources and Processing Strategy

Each dataset was selected based on its relevance to penetration testing and vulnerability management workflows. The processing strategy involved retrieving, cleaning, and transforming each source into a format suitable for semantic search and retrieval.

The details of data sources are described in *Appendix C*.

All unstructured textual content was normalized, cleaned, and converted into vector embeddings using language models like all-mpnet-base-v2. These vectors were then stored in Qdrant, a vector database offering high query speed and scalability.

Structured numerical data (e.g., EPSS scores) was stored separately in PostgreSQL, forming the basis for hybrid search functionality, which combines semantic similarity with numerical filtering.

3.3.2. Architecture and Workflow

The integration workflow follows a modular, scalable structure consisting of the following stages:

1. Fetching:

Each dataset has a dedicated function to retrieve the latest version from public APIs or files. For example:

- CVEs were fetched from Microsoft's registry and third-party APIs.
- MITRE ATT&CK used REST endpoints for raw text extraction.

2. Preprocessing:

Text normalization (lowercasing, punctuation removal), metadata extraction, and filtering irrelevant or outdated entries ensured consistency and quality before further processing.

3. Embedding:

A local embedding model (all-mpnet-base-v2) was used to convert processed text into vector representations. This allowed for semantic search capabilities without relying on external APIs.

4. Storage:

Vectors and associated metadata were stored in Qdrant collections for unstructured text. Structured fields (e.g., EPSS scores) were stored in PostgreSQL for hybrid querying.

5. Querying:

Custom retrievers were implemented to fetch relevant documents during inference. These retrievers supported multi-level access control and hybrid search logic, combining semantic results with structured filters.

To handle large-scale ingestion efficiently, batch processing was introduced. Additionally, health checks were added to validate document counts, detect missing metadata, and ensure consistency across collections.

Multi-Tenancy and Data Isolation

The system supports three levels of data access to ensure proper isolation and prevent leakage:

- Instance Level: Public data shared across all tenants (e.g., CVEs, MITRE framework)
- Tenant Level: Client-specific uploaded files and configurations
- User Level: Temporary files uploaded during a session — automatically cleared at the end of the browser session

This hierarchical approach ensured compliance with data privacy policies and allowed for secure, scalable deployment across multiple clients.

3.3.3. Challenges and Solutions

One of the first major issues was related to handling large volumes of data efficiently, particularly with datasets like CVEs. Due to the sheer volume of CVE entries — often numbering in the tens of thousands — embedding all of them into the vector database became computationally expensive and time-consuming. This posed a challenge for real-time query performance and system responsiveness. To mitigate this, a decision was made to limit the ingestion to only the most recent entries, specifically those from the past seven days. This ensured that the knowledge base remained up to date while also maintaining reasonable system performance. Additionally, batch insertion methods were optimized to improve ingestion efficiency and reduce overhead.

Another significant challenge involved working with non-standard or incompatible data formats, particularly MITRE DEFEND, which was stored in an OWL (Web Ontology Language) format. OWL files required a completely different processing pipeline. Initial attempts to work with Langchain's `GraphCypherQChain` and Neo4j were explored to parse and query the data, but due to complexity and time constraints, full integration was deferred. Instead, I focused was shifted to other datasets which had usable APIs or downloadable JSON/XML formats.

A third challenge was integration of structured numerical data into the RAG system. While unstructured textual content could be effectively embedded and searched using Qdrant, structured fields like EPSS required a different storage and querying approach. This led to the development of a hybrid search mechanism, where queries would first attempt to match structured filters (e.g., specific CVE identifiers or score ranges) via PostgreSQL before falling back to semantic search in Qdrant if no direct match was found. This hybrid model allowed for both semantic relevance and numerical filtering, significantly improving the quality and accuracy of AI-generated responses.

Additionally, early in the development phase, there were issues related to using libraries like onnxruntime-node. These were exacerbated by hardware limitations on certain platforms (e.g., Mac ARM64), leading to slow or failed inference runs. I resolved it by switching to local embedding models via Ollama which offered better compatibility and performance across different environments.

Lastly, during Week 12, a problem was identified with Qdrant's inability to efficiently delete or update individual points, which was necessary for maintaining clean and accurate collections over time. As a result, I integrated alternative vector database – Weaveate.

In summary, while the integration of public cybersecurity data introduced multiple complex challenges, each was met with a tailored solution — from limiting data scope and implementing hybrid search to exploring alternative technologies.

4. Conclusion

This internship at Refracted Security provided an opportunity to integrate AI-powered enhancements into a professional cybersecurity vulnerability management platform. The primary objective was to develop intelligent tools that support penetration testers in generating, refining, and contextualizing security findings — ultimately improving both the quality and efficiency of their reports.

During the internship, I successfully implemented three core components:

1. **Sidebar Chat Helper:** A lightweight, always-accessible AI assistant embedded within the application interface, enabling real-time assistance without requiring users to switch context.
2. **Observations Improvement Feature:** A tool allowing users to enhance written observations with AI-generated suggestions such as summarization, rewriting, translation, and example generation, while maintaining editorial control.
3. **Automated Public Data Integration Pipeline:** A system that fetches and processes publicly available cybersecurity datasets (e.g., CVEs, CWEs, CAPECs, MITRE ATT&CK) to enrich the AI's knowledge base using Retrieval-Augmented Generation.

These features were developed using modern web technologies including Next.js, React, TypeScript, and Material UI, and integrated securely with backend services via Azure-hosted OpenAI models and Qdrant as a vector database. Throughout the development process, architectural decisions prioritized modularity, scalability, and data privacy, ensuring alignment with company standards and infrastructure.

Key Achievements were:

- Implementation of a robust RAG pipeline that dynamically incorporates up-to-date threat intelligence into AI responses.
- Design and integration of a hybrid search mechanism combining semantic similarity with structured filtering significantly enhancing answer accuracy.
- Development of a modular codebase that allows future migration between vector databases or embedding models with minimal refactoring effort.
- Ensuring multi-level data isolation to maintain data integrity and prevent leakage across environments.

The internship also presented several technical and conceptual challenges, including:

- Efficient handling of large-scale public cybersecurity datasets.
- Compatibility issues with formats like OWL used by MITRE DEFEND.
- Ensuring consistent chat state across multiple browser tabs.
- Balancing performance and accuracy when implementing local embedding models.

In addition to technical achievements, the internship offered a rich learning experience. I gained hands-on exposure to full-stack development, AI integration, and secure data handling practices within a professional software development team.

Despite not completing every planned feature — such as the Template Quality Controller and full integration of MITRE DEFEND — the implemented components formed a solid foundation upon which future enhancements can be built. The modular architecture and clean code organization also allow for easy maintenance and potential migration to other vector databases, such as Weaviate, should that become necessary in the future.

While the current implementation provides a strong foundation, several areas offer opportunities for further improvement:

- Self-hosted AI Models: Exploring fully self-hosted LLMs could increase data sovereignty and reduce dependency on external services.
- Enhanced Tagging System: Integrating the output of the RAG system into the company's planned tagging feature would allow for automatic suggestion of relevant attack patterns (e.g., CWEs, CAPECs).
- Migration to Alternative Vector Databases: As discovered during the final week, Weaviate may provide better long-term compatibility for update/delete operations compared to Qdrant.
- Improved Handling of Structured Data: Expanding the hybrid search functionality to include more advanced filtering and visualization options could further improve usability for end-users.

In conclusion, this internship not only allowed me to contribute meaningful AI-driven improvements to a real-world cybersecurity product but also served as a valuable learning experience in practical software engineering, problem-solving, and professional collaboration.

REFERENCE LIST

- Microsoft. (n.d.). Microsoft Security Response Center (MSRC) API Documentation. <https://msrc.microsoft.com>
- Hugging Face. (2024). Sentence Transformers: all-mpnet-base-v2. <https://huggingface.co/sentence-transformers/all-mpnet-base-v2>
- Qdrant. (2024). Qdrant Vector Database Documentation. <https://qdrant.tech>
- LangChain. (2024). LangChain Documentation. <https://docs.langchain.com>
- Pinecone. (2024). Pinecone Vector Database Documentation. <https://www.pinecone.io>
- Weaviate. (2024). Weaviate Vector Search Engine Documentation. <https://weaviate.io/developers/weaviate>
- OpenAI. (2024). OpenAI API Documentation – Embeddings. <https://platform.openai.com/docs/guides/embeddings>
- MITRE Corporation. (2024). MITRE ATT&CK Framework. <https://attack.mitre.org>
- MITRE Corporation. (2024). MITRE DEFEND Framework. <https://defend.mitre.org>
- CVE Details. (2024). CVE Vulnerability Database. <https://www.cve.org>
- FIRST.org. (2024). Common Vulnerability Scoring System (CVSS). <https://www.first.org/cvss>
- Forum of Incident Response and Security Teams (FIRST). (2024). Common Weakness Enumeration (CWE). <https://cwe.mitre.org>
- Common Attack Pattern Enumeration and Classification (CAPEC). (2024). <https://capec.mitre.org>
- National Institute of Standards and Technology (NIST). (2024). National Vulnerability Database (NVD). <https://nvd.nist.gov>
- Neo4j. (2024). Neo4j Graph Database Documentation. <https://neo4j.com/docs>
- Ollama. (2024). Ollama Local LLM Hosting Tool. <https://ollama.ai>
- GitHub. (2024). Transformers.js – JavaScript Library for ONNX Models. <https://github.com/xenova/transformers.js>
- GitHub. (2024). LangChainJS – JavaScript Implementation of LangChain. <https://github.com/langchain-ai/langchain>
- PostgreSQL. (2024). PostgreSQL: The World's Most Advanced Open-Source Relational Database. <https://www.postgresql.org>
- Prisma ORM. (2024). Prisma – Modern Node.js and TypeScript ORM. <https://www.prisma.io>
- Figma. (2024). Figma Design and Prototyping Tool. <https://www.figma.com>
- Visual Studio Code. (2024). VS Code Official Website. <https://code.visualstudio.com>
- Vercel. (2024). Next.js Documentation. <https://nextjs.org/docs>

React. (2024). React Official Documentation. <https://reactjs.org>

Material UI. (2024). MUI – Material Design Library for React. <https://mui.com>

Azure. (2024). Azure OpenAI Service Documentation. <https://learn.microsoft.com/en-us/azure/cognitive-services/openai>

SonarCloud. (2024). Code Quality and Security Analysis. <https://sonarcloud.io>

ATTACHEMENTS

Appendix A: Embedding Model Evaluation Table

Criteria	Weight	all-mpnet-base-v2	all-MiniLM-L6-v2	OpenAI text-embed-3-large	OpenAI text-embed-3-small
Accuracy	30%	4.8	4.2	4.9	4.5
Speed (ms/query)	20%	4.5 (45ms)	5.0 (28ms)	3.8 (62ms)	4.7 (35ms)
Cost per 1M tokens	20%	5.0 (\$0)	5.0 (\$0)	2.0 (\$0.80)	3.0 (\$0.02)
Model Size	15%	4.0 (420MB)	5.0 (80MB)	3.5 (API-based)	4.0 (API-based)
Domain Adaptation	15%	4.7	4.3	4.5	4.2
Total Score	100%	4.56	4.48	3.96	4.20

Appendix B: Vector Database Comparison Table

Criteria	Weight	Qdrant	Pinecone	Weaviate
Query Speed	30%	5	4	3
Scalability	25%	4	5	4
Security	20%	5	4	5
Deployment Ease	15%	4	5	2
Cost Efficiency	10%	5	3	4
Total Score	100%	4.6	4.2	3.6

Appendix C: Public Cybersecurity Datasets Overview

Data Source	Description	Processing Approach
CVEs	Vulnerability identifiers with descriptions and impact details	Retrieved via public APIs; filtered to only recent entries (last 7 days) due to computational constraints during embedding
EPSS scores	Numerical scores indicating likelihood of exploit	Separately stored in PostgreSQL due to structured nature; integrated using hybrid search
CAPEC & CWE	Taxonomies of attack patterns and software weaknesses	Stored with metadata such as severity, likelihood, and relationships between entries
MITRE ATT&CK	Frameworks describing adversarial tactics and defensive strategies	ATT&CK data retrieved from API;