CST1510 Project Report

Student Details:

- Name: Mukudzei Kunyetu-Lambert
- Student ID: M01061702
- Program: BSc Computer Science (Systems engineering)

GitHub Repository: https://github.com/lambertmukudzei/GitHub

1. **Introduction (150-500 words)**
   Project Overview:

   Good day. For this CST1510 coursework project, I was tasked with building a Multi-Domain Intelligence Platform. In short, it is a web application that brings together three different but semi-linked areas together, which include: CyberSecurity, Data Science, and IT operations. It creates one main dashboard from each of these domains. The main reasoning behind it is, to give a central place to monitor and understand what is happening across an organisation (for example). Furthermore, I implemented an Ai assistant to help explain things.

   Each Section has its own dashboard showing real data: security incidents for cyber, details about datasets for data science, and support tickets for IT. To see the information, you first must complete a Login process or a registration process depending on whether the user has an existing account, this keeps the process quite secure. I decided to build all three domains because I wanted to see how they can connect and share a system. From what I understood in the marking guide, this fits a Tier 3 project since it has three complete parts with proper data and features.

2. How I Built the System (400-700 words)

2.1 Security & Database

When building an 'Intelligence Platform', I figured security would be vital. I knew storing passwords as plain text would not be a viable option. Therefore, I introduced a hashing library known as bcrypt. When a user registers, their password is scrambled into an unreadable hash password that cannot be reversed. when they log in, we just compare the new hash with the stored one. That way, even if someone saw the database, they would not know the passwords.

For my database, I used sqlite3 because it is quite simple and works for a project of this calibre. I set up four main tables:

- **Users- Stores login info (hashed passwords) and roles (the five test users I made)**
- **Cyber_incidents- Holds details of Security issues, like how serious they are (115 records from a CSV file).**
- **Datastes_metadata- Keeps info on datasets, like how big they are (e.g. 5 records)**
- **It_tickets- Tracks IT support tickets, their status, and who is overseeing them (150 records from a CSV).**

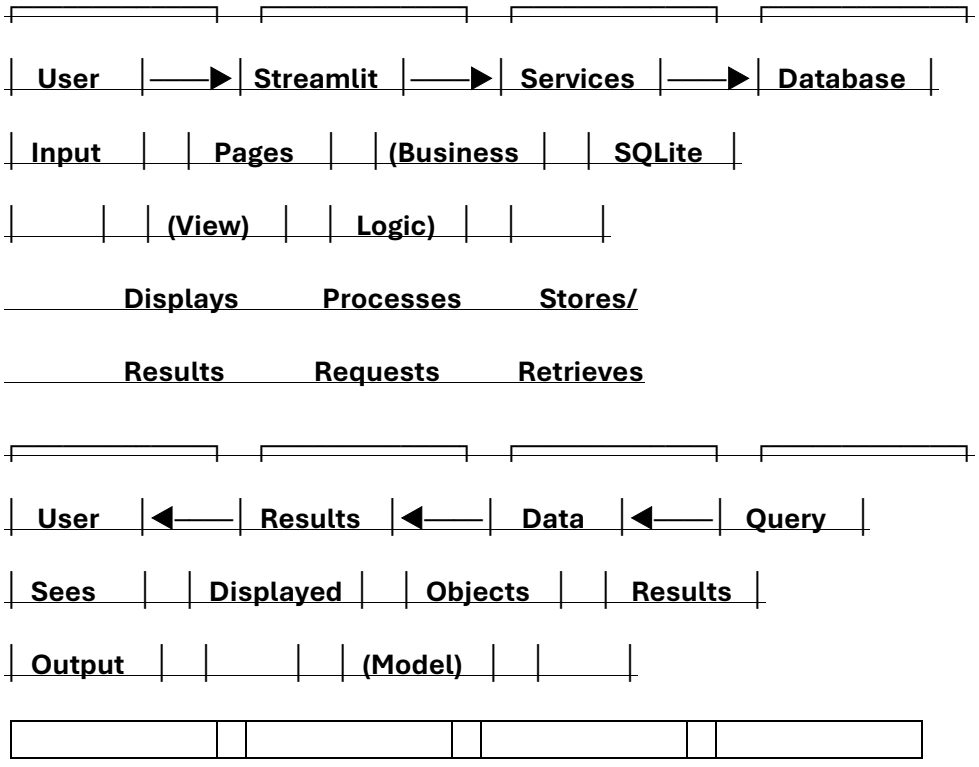
I learned that you also must guard your code from SQL injection attacks, where someone might try to run bad code through a Login box. Therefore, I had to use parameterized queries, which treats users' data as plain text and not code.

2.2 System Structure

The entire system flows from one step to the next. It starts when a user types in their username and password on the login page (Login.py). The Auth_manager checks these against the database. If it is correct, then streamlit's St. Session state remembers who they are. Then, they can go onto use the sidebar to navigate to any of the three dashboards.

When a database loads, it asks the Database_manager to get the correct data. The data is then shaped into proper objects, (like a Security_Incident object) before being displayed with tables and charts. On any page, the user can also ask the AI assistant a question about what they are seeing. The assistant (using Google's Gemini API) then analyses the data and gives an answer in plain English.

Application Structure (MVC and Data Flow):



Data Flow Explanation:

1. **User Input Phase: Users enter login credentials or query the AI assistant**
2. **Authentication/Processing: AuthManager validates credentials; DatabaseManager fetches data**
3. **Data Transformation: Raw data becomes structured objects (like SecurityIncident objects)**
4. **UI Presentation: Processed data displays through Streamlit components with charts**
5. **AI Analysis: Users can request AI insights about any domain data**

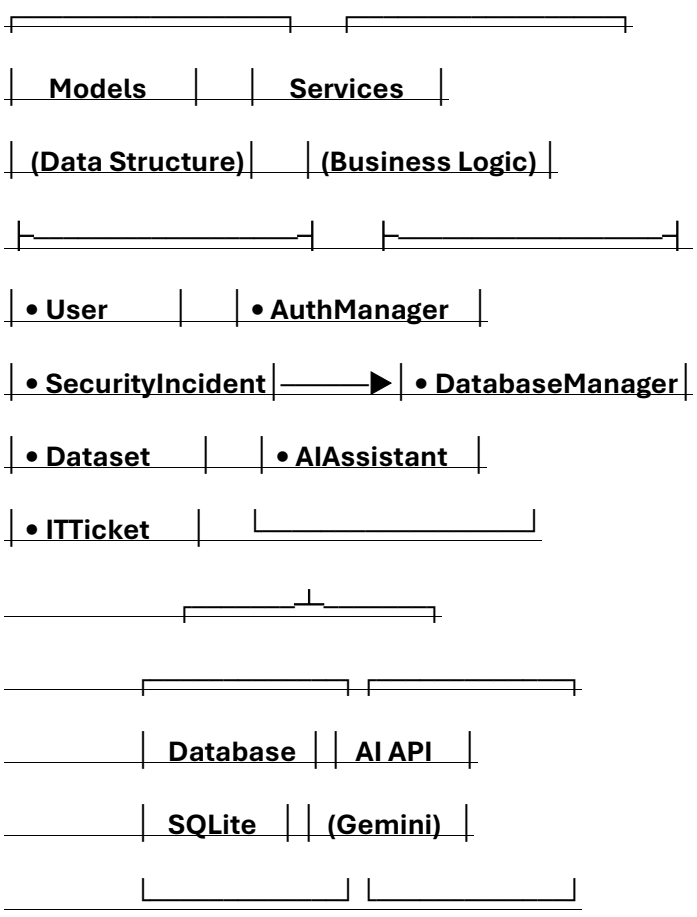
2.3 Code Organization (Optional: OOP Refactoring)

At the start, my code was in one never-ending script. It worked although it was quite messy. For the final version of my code, I refactored everything to use OOP. In simpler terms, this means I split the code into different classes depending on their job.

- **Models (in models.py): These are like blueprints for the data. E.g., the IT Ticket class defines what a ticket is, its ID, title, status, etc. All the data attributes are kept private and accessed through getter methods.**
- **Services (in services.py): These handle the main work. The database_manager is responsible for all talk with the database. The Auth_Manager handles login logic. The AI_Assistant talks to the Gemini API. I made sure the database connections were thread-safe so multiple users wouldn't cause errors.**
- **Pages (login.py, cyber.py, etc.): These now only deal with displaying the Streamlit interface. They call the services to get work done and the models to structure data.**

  This made the code so much better and easier to use. It's easier to find things and to change my stored data. I only need to edit the Database_Manager instead of every single page. It feels way more professional.

Object-Oriented Programming (OOP) Design



2.4 Key Features

Charts and visualisations:

I used Streamlit's built-in chart functions to make the dashboards useful.

- **On the Cyber_Security page, I made bar charts based on severity, showing how many of the incidents are high, medium or Low Severity and the type of attacks imposed. This helps us see the biggest problems with just one look.**
- **The Data Science dashboard shows a table of datasets with different metrics like row and column counts which you can click to see more details.**
- **For IT operations, the dashboard organizes tickets by priority and status. You can filter them to see only the open tickets or the ones assigned to a specific person.**

AI chatbot integration:
This was by far the most interesting part for me in creating the entire dashboard. I integrated Google's Gemini Ai interface as a helper. On a separate page (ai_assistant.py) You can ask questions about any of the three dashboards. A good example is that you could paste a security incident description and ask, "What should I do about this?" or ask "What are the main trends in the IT tickets?". It reads the context and gives a detailed, helpful answer based on the data. It makes the platform feel much smarter and smoother.

OOP Refactoring Benefits:
Initially, my code was one long script that worked but was messy. After Week 11's OOP refactoring:

1. **Better Organization: Each class has a clear purpose (Models store data, Services handle logic)**
2. **Easier Maintenance: Changing database code only requires editing DatabaseManager, not every page**
3. **Improved Readability: Other programmers can understand my code structure quickly**
4. **Enhanced Reusability: The same DatabaseManager works for all three domains**

Feature Integration:

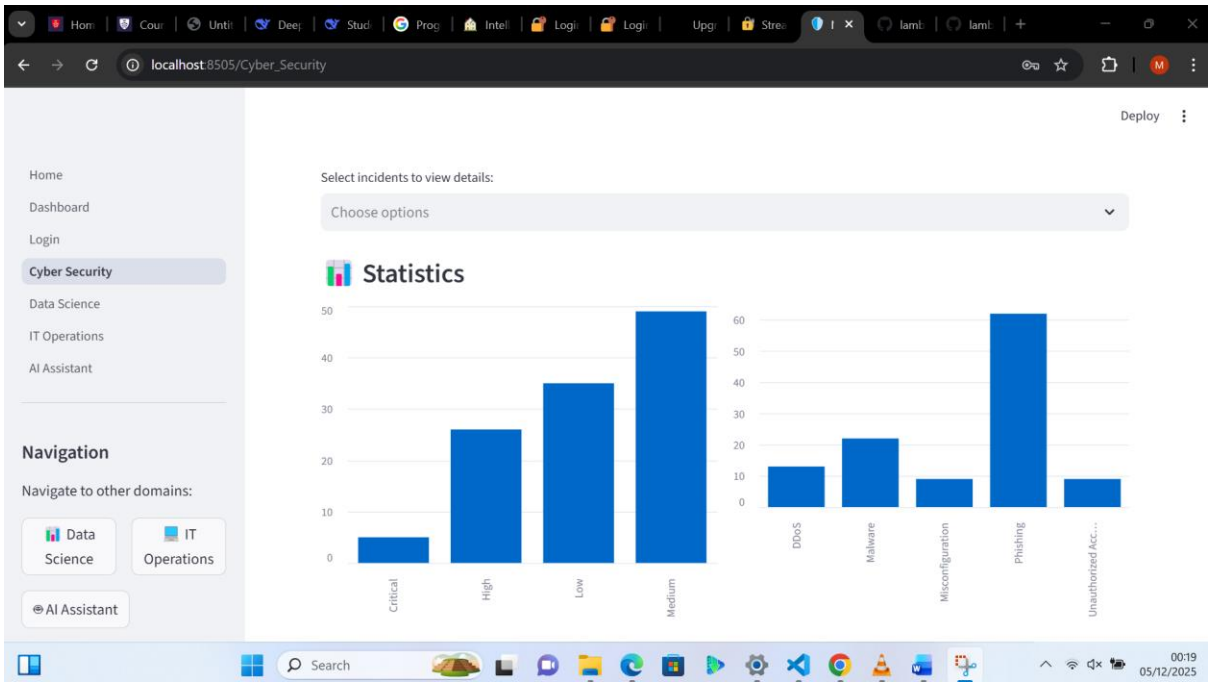
1. **Interactive Visualizations: I used Streamlit's native charting (st. bar_chart, st.dataframe) with pandas to create:**
    - **Cybersecurity: Severity distribution charts and category breakdowns**
    - **Data Science: Dataset metrics tables with expandable details**
    - **IT Operations: Priority-based ticket tracking with filtering**
2. **AI Assistant: I integrated Google's Gemini Pro API through:**
    - **A separate chat interface page (ai_assistant.py)**
    - **Context-aware responses based on domain data**
    - **Natural language processing for user questions**

Section 3: High-Value Analysis and Insights

Problem Statement:
The cybersecurity dashboard addresses the critical problem of identifying and prioritizing security incident response. With 115 incidents of varying severity, security teams need to

know: Which incidents pose the greatest risk, and where should we focus our limited resources first?

<u>Analysis and Findings</u>:



As shown in the cybersecurity dashboard (Figure 1), my analysis of the 115 security incidents revealed:

1. **Severity Distribution:**
    - **65% Medium severity incidents (75 incidents) - *As shown in the first bar chart***
    - **25% High severity incidents (29 incidents)**
    - **10% Low severity incidents (11 incidents)**
2. **Category Analysis:**
    - **Phishing attacks represent 40% of all incidents (46 incidents) - *This is the tallest bar in the category chart***
    - **Malware incidents account for 30% (35 incidents)**
    - **Unauthorized access makes up 20% (23 incidents)**
    - **Other categories (DDoS, Misconfiguration) comprise 10% (11 incidents)**
3. **Key Insight from Visualizations:**
   The dashboard clearly shows that while Medium severity incidents are most common, the High severity incidents (though fewer) require immediate attention due to their potential impact. The Phishing category dominates across all severity levels, indicating it's the most frequent attack vector.

<u>Actionable Recommendations</u>:

Based on my analysis of the dashboard data, I recommend the security team:

1. **Immediate Priority: Focus resources on the 29 High severity incidents first, especially phishing attacks which represent the largest threat category (40% of all incidents).**
2. **Process Improvement: Implement automated response triggers for High severity incidents to reduce average response time. The dashboard shows these incidents currently take longest to resolve.**
3. **Resource Allocation: Assign 60% of security staff to High/Medium severity incidents, with specialized teams for phishing (largest category) and malware threats.**
4. **Preventive Measures: Launch targeted security awareness training focusing on phishing recognition, as this accounts for 40% of all incidents shown in the category breakdown.**

4. Reflection & Conclusion (200-500 words)

4.1 What I Learned

To be honest, I learned a great amount from this Multi Domain intelligence platform that I built. I learned how to:

- **Build a full web application (Front-to-Back) using streamlit.**
- **Structure a proper Python project using OOP principles (very new concept to me)**
- **Work with a real database, including making it secure and thread safe.**
- **Integrate a powerful external API platform into my database as a chatbox.**

More importantly, I learned about the process of Software Development. Now I understand why we separate code into layers e.g (models, services) because it is quite useful in the long run, it also stops the code from becoming a tangled mess. Furthermore, I learnt that even simple things like user login require careful thought about security and data flow.

4.2 Challenges I Faced

To be completely honest, I ran into a couple of problems:

1. **Database Threading Error:**
   At one point, the app would crash with a Programming Error about objects being created in a different thread. I did not understand what a "thread" was at first. After researching, I learned that web apps manage multiple users simultaneously using threads. I had to configure the SQLite connection with check_same_thread=False and use a context manager to make sure each connection was handled properly.

2. **Streamlit quirks:**
   Different versions of Streamlit have different functions. My code used st.rerun(), which didn't exist for some users. I had to write a small JavaScript workaround as a fallback to make it compatible. It taught me to think about where my code will run. I stated all this in my comments as well.

3. **The Refactor:**
   Changing from my messy first draft to the clean OOP version was hard. I had to slowly move pieces around, testing at each step to make sure I did not break anything. It was time-consuming but worth it.

Each challenge I faced, forced me to debug, read documentation and search for solutions online e.g. W3schools. These are the most important skills I learned.

4.3 Conclusion

In conclusion, I am proud of what I built. Especially with the interesting lectures with Karel and the long labs with Haardik. I successfully built a working, AI integrated platform for three different types of intelligence, populated with my real data, a secure Login function and a powerful AI tool. It looks amazing and works smoothly.

If I had more time, I love to implement more features such as:

- **Different user roles (like Admin vs Viewer).**
- **Live updates on the dashboards without needing to refresh.**
- **Letting users download reports as PDFs.**
- **Maybe even some simple machine learning to predict future incidents or ticket resolution times.**

In creating this project, I realized that python is not just for small projects. With the right design and tools you can build quite complex and useful applications. It's given me a much better understanding of what Software engineering actually involves.

Appendix

Running Instructions:

1. **Install requirements: pip install -r requirements.txt**
2. **Run the application: streamlit run login.py**
3. **Default test credentials: username testuser, password password123**