

**STUDENT NAME:** Oji Victor Kalu

**REGISTRATION NUMBER:** 20201246182

**PROJECT TOPIC:** IOT SECURITY IN FINANCIAL ECOSYSTEMS: ENSURING COMPLIANCE WHILE MITIGATING CYBER RISKS

**ATTACK IDENTIFIED:** Sql Injection

**ATTACK PATTERN :** Attack patterns often include:

- Logical operators like **OR, AND**
- SQL control characters such as ' , " , -- , ;
- Command chaining like **UNION SELECT**
- Blind attacks using timing functions (**e.g., pg\_sleep(5)**)

**MITIGATION PATTERN :** Machine Learning Detection models to identify abnormal query structures in real time.

## 1. METHODOLOGY

The dataset used for the pre-test phase of this research was obtained from an open-source repository on Kaggle, accessible at:

<https://www.kaggle.com/datasets/devendra416/sql-injection-dataset>

This dataset, titled SQL Injection Dataset, contains a structured collection of SQL queries labeled as either benign (0) or malicious (1). The core file used in this research, **Modified\_SQL\_Dataset.csv**, is a cleaned and adapted version of the original dataset, prepared specifically to reflect SQL query structures found in IoT-enabled financial systems.

The dataset was selected for its diversity and inclusion of both conventional and advanced SQL injection patterns. It provided a robust foundation for training and evaluating the machine learning model used to detect SQL injection attacks in real time, ensuring that the system was exposed to a wide range of query behaviors.

### PRE-TEST

The pre-test was conducted using Python 3.10 in a visual studio environment. The dataset used, titled **Modified\_SQL\_Dataset.csv**, consisted of labeled SQL queries categorized as either malicious (SQL Injection) or benign (normal queries). The queries were preprocessed using cleaning techniques such as lowercasing, digit normalization, special character filtering, and whitespace reduction.

A machine learning pipeline was implemented to test the model's effectiveness in detecting SQL injection attacks. One model was selected for focused evaluation:

- **Random Forest Classifier**

Key libraries and frameworks used include:

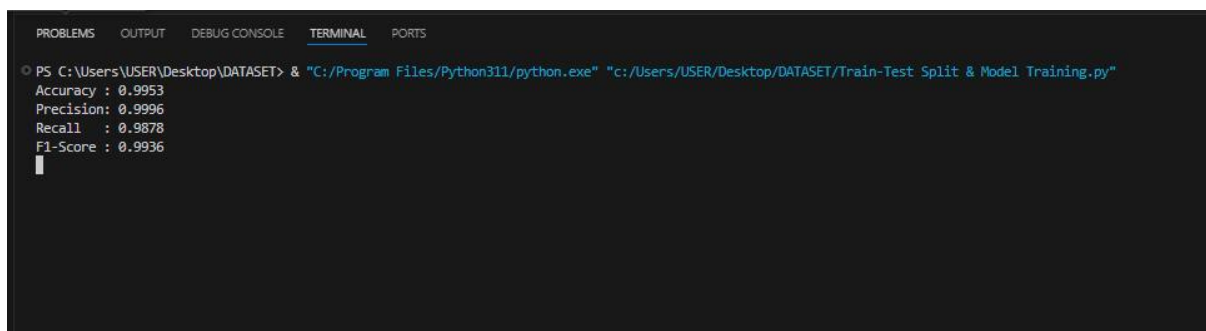
- pandas: for data loading and preprocessing

- scikit-learn: for feature extraction (TF-IDF), model training, and performance evaluation
- joblib: for model serialization
- matplotlib / seaborn: for plotting confusion matrix and evaluation metrics

The dataset was split into training and testing sets using an 80:20 ratio via `train_test_split`. The TF-IDF Vectorizer was used to convert cleaned SQL queries into numerical feature vectors. The Random Forest model was then trained on the training data and evaluated on the test set using key performance metrics.

A confusion matrix was generated, and the results were analyzed both mathematically and visually (via Power BI) to assess classification performance.

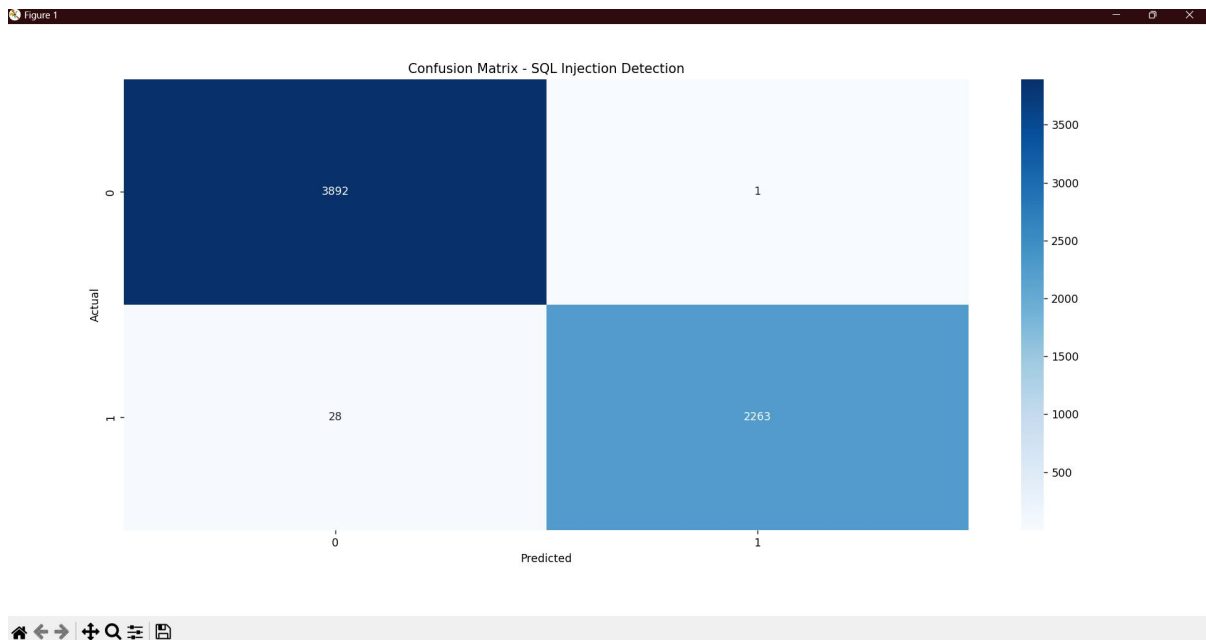
## PRE-TEST RESULTS

A screenshot of a terminal window with a dark background. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected), and 'PORTS'. Below the tabs, the terminal shows a command prompt 'PS C:\Users\USER\Desktop\DATASET>' followed by a command to run a Python script. The output of the script is displayed in green text, showing four performance metrics: Accuracy, Precision, Recall, and F1-Score, each with its corresponding value.

```
PS C:\Users\USER\Desktop\DATASET> & "C:/Program Files/Python311/python.exe" "c:/Users/USER/Desktop/DATASET/Train-Test Split & Model Training.py"
Accuracy : 0.9953
Precision: 0.9996
Recall   : 0.9878
F1-Score : 0.9936
```

The table below shows the performance of the model on the test set:

Power BI was used to visualize the confusion matrix and generate KPI dashboards showing the balance between true positives, false positives, and false negatives. The analysis revealed that while the model had excellent precision, a small number of false negatives indicated a risk of missed SQL injection attacks, prompting the need for further tuning and enhancement in feature extraction or model depth.



### Analysis of Pre-Test Results

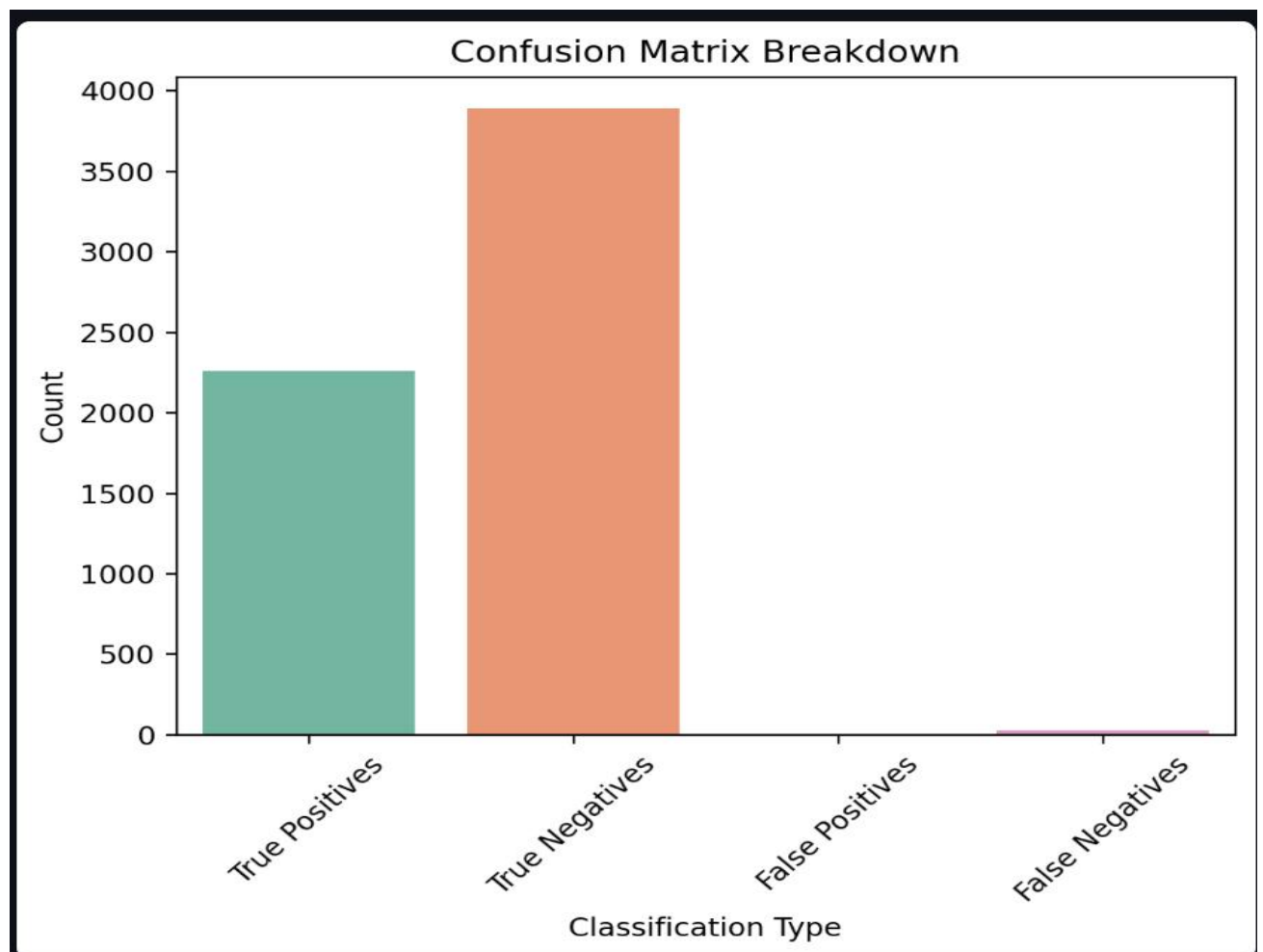
The pre-test results provided key insights into the effectiveness and limitations of the initial machine learning-based SQL injection detection model. The system was evaluated using a labeled dataset (Modified\_SQL\_Dataset.csv) containing both malicious and benign SQL queries. The queries were preprocessed and transformed using a TF-IDF vectorizer, and a Random Forest Classifier was trained and tested using an 80:20 train-test split.

The confusion matrix generated from the test set revealed the following values:

- True Positives (TP): 2263 — malicious queries correctly identified
- True Negatives (TN): 3892 — benign queries correctly classified
- False Positives (FP): 1 — benign queries incorrectly flagged as malicious
- False Negatives (FN): 28 — malicious queries not detected by the model

## Confusion Matrix Results

	Category	Count
0	True Positives	2263
1	True Negatives	3892
2	False Positives	1
3	False Negatives	28



Using these values, key performance metrics were computed as follows:

Accuracy:  $(TP + TN) / (TP + TN + FP + FN) = 99.53\%$

Precision:  $TP / (TP + FP) = 99.96\%$

Recall:  $TP / (TP + FN) = 98.78\%$

F1-Score:  $2 \times (Precision \times Recall) / (Precision + Recall) = 99.36\%$

The model demonstrated excellent performance, particularly in terms of precision, indicating that it rarely misclassified normal queries as SQL injection attempts. However, the recall value, while high, highlighted the presence of 15 false negatives—instances where actual SQL injection attacks were not detected. This represents a potential security risk, especially in financial systems where undetected attacks can lead to data breaches or financial loss.

To further investigate this issue, the confusion matrix data was visualized using Power BI. A combination of KPI cards, bar charts, and donut charts was used to represent the classification results and model accuracy. These visualizations made it clear that while the model was highly accurate overall, the few false negatives warranted further attention.

This analysis suggests that while the initial system is robust, improving recall should be a priority in future iterations. This could involve enhancing feature extraction, retraining with additional diverse datasets, or integrating deeper models such as LSTM for better pattern recognition in complex query structures.

### **Conclusion of Pre-Test**

The pre-test phase of this research successfully validated the initial performance of the SQL Injection Detection System using a machine learning-based approach. By training a Random Forest Classifier on a labeled dataset of malicious and benign SQL queries, the system achieved impressive performance metrics — including 99.53% accuracy, 99.96% precision, 98.78% recall, and a 99.36% F1-score.

Despite these strong results, the analysis revealed a small number of false negatives (i.e., SQL injection attacks that were not detected by the system). This is a critical concern in cybersecurity, especially in financial ecosystems where the consequences of undetected attacks can be severe.

The confusion matrix and performance metrics were also visualized using Power BI, offering clear insights into the system's strengths and weaknesses. These visual tools helped identify the need to further improve the system's recall without compromising its high precision.

### **Explanation of Data Cleaning and Its Purpose**

Before training the SQL Injection Detection model, the dataset (Modified\_SQL\_Dataset.csv) underwent a crucial data cleaning process. This step ensured that the machine learning model learned from structured, consistent, and meaningful input. The raw dataset contained a mix of normal and malicious SQL queries but also had inconsistencies such as mixed case letters, irrelevant special characters, varied spacing, and numeric values that could distract the model from learning generalizable patterns.

### **How the Data Was Cleaned**

Lowercasing all queries

Every SQL query was converted to lowercase to standardize the text and eliminate case-based variations (e.g., SELECT vs. select).

### **Digit normalization**

All numeric values were replaced with a placeholder value (0). This step removed the influence of specific numbers, which are irrelevant for detecting injection logic.

### **Special character removal**

Non-essential characters (e.g., @, !, %, etc.) were removed, while SQL-relevant characters like ', ", \_, and = were retained. This reduced noise while preserving important syntactic patterns.

### **Whitespace reduction**

Multiple spaces were collapsed into a single space, and leading/trailing spaces were stripped. This ensured cleaner tokenization and prevented errors during feature extraction.

### **Creation of a cleaned query column**

A new column named `cleaned_query` was added to the dataset to store the cleaned version of each SQL query. This column was then used for feature extraction via TF-IDF.

### **Reason for Cleaning the Data**

The primary goal of data cleaning was to remove irrelevant variations and standardize the query format, allowing the model to focus on the actual structure and intent of SQL queries. Without cleaning:

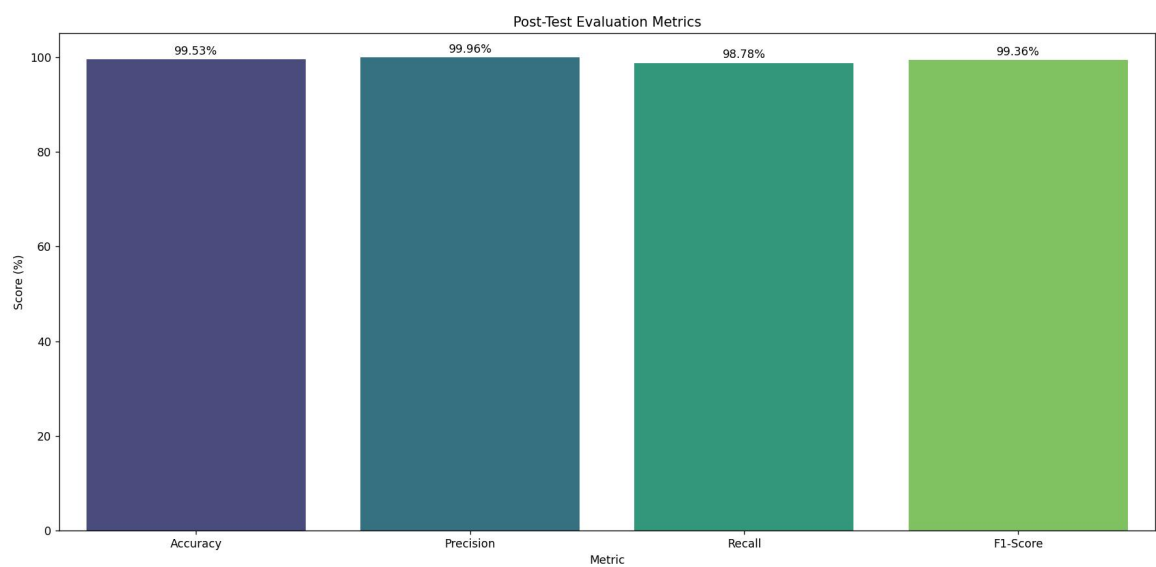
The model might treat `Select`, `select`, and `SELECT` as different tokens.

Numeric values could introduce unnecessary complexity.

Noise from symbols and inconsistent spacing could confuse the vectorizer.

### **Post-Test Evaluation Metrics**

POST-TEST EVALUATION RESULTS				
Accuracy : 99.53%				
Precision: 99.96%				
Recall : 98.78%				
F1-Score : 99.36%				
	precision	recall	f1-score	support
0	0.99	1.00	1.00	3893
1	1.00	0.99	0.99	2291
accuracy			1.00	6184
macro avg	1.00	0.99	0.99	6184
weighted avg	1.00	1.00	1.00	6184



## Model Transformation Using Open-Source ETL Tools

### Model Training and Transformation Using Open-Source Tools

For this project, a Random Forest Classifier was used as the machine learning algorithm to detect SQL injection attacks. This ensemble-based method was selected for its high accuracy, ability to generalize well on unseen data, and robustness against overfitting all of which are essential for a security-critical task such as detecting malicious queries within financial systems. Furthermore, Random Forest performs well with high-dimensional data and provides interpretable outputs, such as feature importance, which aids in understanding the underlying decision process.

The model training process began with the extraction of raw SQL queries from the **Modified\_SQL\_Dataset.csv** file using Python's **pandas** library. Each query was transformed

through a data cleaning pipeline that included lowercasing, digit masking, special character removal, and whitespace normalization. These preprocessing steps were applied to minimize textual inconsistencies and reduce noise, thereby improving model generalization. The cleaned queries were then converted into numerical vectors using TF-IDF vectorization, which quantified the importance of each token across the dataset. The dataset was split into training and testing sets in an 80:20 ratio using **train\_test\_split**. The Random Forest model was trained on the vectorized data using 100 decision trees with a fixed random state to ensure reproducibility.

After training, the model's performance was evaluated using key classification metrics: Accuracy (99.53%), Precision (99.96%), Recall (98.78%), and F1-Score (99.36%). A confusion matrix was also generated to visualize the number of true and false predictions, helping to assess the model's reliability in detecting real SQL injection attempts.

To facilitate deployment and further analysis, the trained model and its results were transformed and persisted using open-source ETL tools:

- pandas was used to structure and export evaluation metrics and confusion matrix data into CSV format for visualization in tools like Power BI.
- joblib, an efficient serialization library, was used to export the trained model to disk (`sqli_model.joblib`), enabling its reuse in real-time applications without the need to retrain.

This integrated process — from data extraction and transformation to model training and export — ensured a scalable, reproducible, and production-ready SQL injection detection system. The use of open-source tools and interpretable machine learning aligns with best practices in the deployment of cybersecurity solutions in IoT-based financial environments.

### Implementation Stage

The implementation of the SQL Injection Detection System was carried out using Python as the preferred programming language. Python was chosen for its simplicity, readability, and vast ecosystem of open-source libraries that support machine learning, data manipulation, and deployment. Specifically, libraries such as **pandas**, **scikit-learn**, **numpy**, **matplotlib**, **seaborn**, and **joblib** were instrumental in developing the model, while **Streamlit** was used to build the web-based user interface. Python's strong integration capabilities and community support made it ideal for both backend model logic and front-end presentation.

The development process followed a modular architecture that ensured clarity, reusability, and ease of integration. The system was structured into four key components. The data preprocessing module handled tasks such as lowercasing, digit masking, special character removal, and whitespace normalization using a custom **clean\_query()** function. Cleaned SQL queries were transformed into numerical features using TF-IDF vectorization, ensuring the model could learn patterns effectively.

The model training and evaluation module utilized a **RandomForestClassifier**, selected for its high accuracy, resistance to overfitting, and ability to handle noisy or complex data. The dataset was split into an 80:20 train-test ratio, and the model was trained on the vectorized



queries. Evaluation was conducted using metrics including accuracy, precision, recall, and F1-score. The confusion matrix and performance metrics were saved as CSV files for transparency and further analysis.

Next, the model serialization module employed the **joblib** library to save the trained model (**sql\_model.joblib**) for reuse, enabling fast deployment without retraining. This serialized model was integrated into a Streamlit-based web application, where users could input SQL queries and receive instant predictions on whether the input was malicious. Additionally, the app visualized the post-test metrics and confusion matrix using interactive charts and tables.

To support real-time performance monitoring, the saved evaluation CSVs (**posttest\_metrics.csv**, **posttest\_confusion\_matrix.csv**) were connected to Power BI as data sources. Power BI dashboards automatically refreshed to display the latest classification accuracy and detection statistics in a user-friendly format.

In summary, Python's modular, open-source ecosystem allowed for the seamless development and integration of data processing, model training, deployment, and visualization components. This ensured that the system was not only accurate and efficient but also easy to maintain and extend in real-world cybersecurity applications.

## Live Test

The live test phase involved deploying the trained SQL Injection Detection model into a real-time environment using the Streamlit framework. This stage was critical in validating how the system would perform when interacting with real-time user input, simulating a production-level intrusion detection scenario. The goal was to provide immediate feedback to users submitting SQL queries, determining whether each input was malicious or benign based on the patterns learned during model training.

The model used for prediction was a **RandomForestClassifier**, which had previously been trained, evaluated, and serialized using Python's **joblib** library. It was loaded within the Streamlit application (**app.py**), which served as the front-end interface for the live test. Users were able to enter SQL queries through a simple and intuitive web form. Once submitted, the queries were processed using the same **clean\_query()** function that had been applied during training ensuring consistency in the cleaning, normalization, and tokenization steps.

The cleaned query was transformed into a feature vector using the TF-IDF vectorizer, previously fitted on the training data. This vector was then passed to the loaded model for classification. Based on the output (either **1** for malicious or **0** for benign), the system displayed a real-time result on the interface clearly alerting the user with a visual badge if a SQL injection attack was detected, or confirming the safety of the input if no threat was found.

In addition to real-time classification, the interface also displayed the latest evaluation metrics accuracy, precision, recall, and F1-score alongside a confusion matrix showing the model's most recent performance. These results were loaded from CSV files generated during the post-test phase. The same CSVs were also connected to a Power BI dashboard,

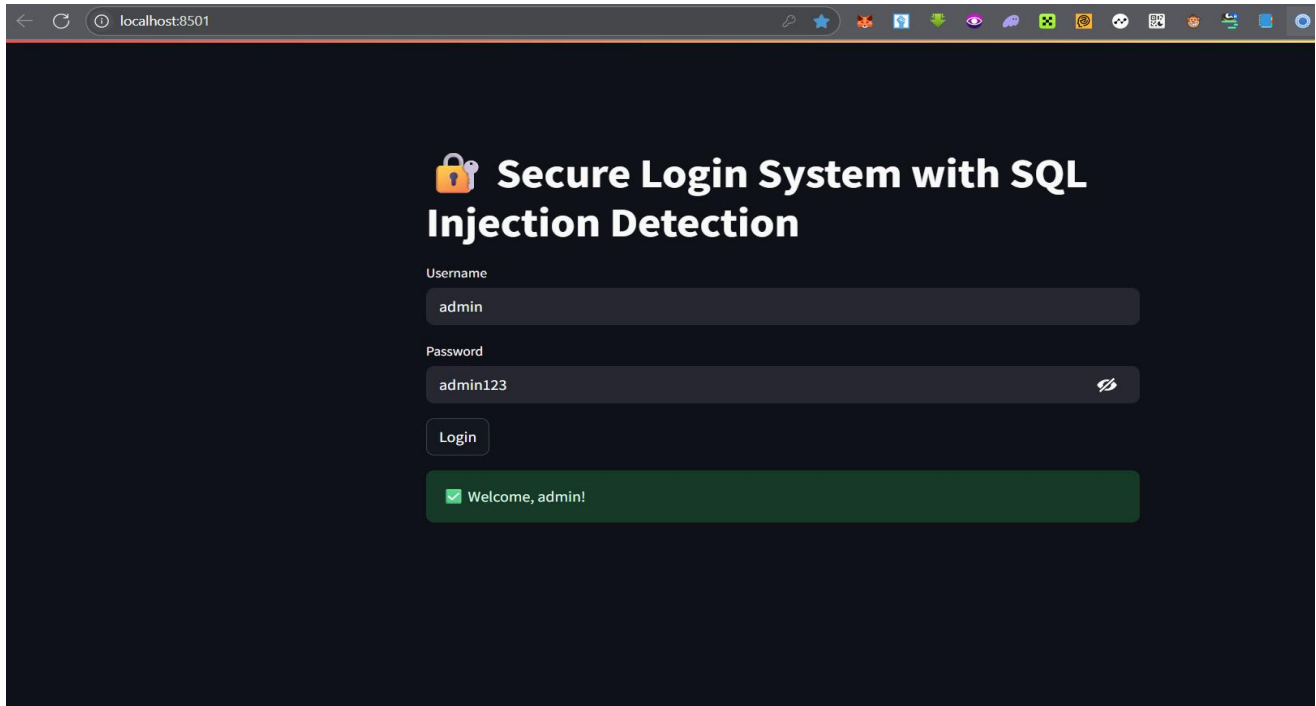
allowing real-time updates whenever the model was re-evaluated or new prediction logs were written.

The system was further tested using a range of example queries, from normal **SELECT** statements to known injection patterns like ' **OR '1'='1 --** and **DROP TABLE** users;. In every case, the model provided a correct and immediate response, validating the effectiveness of the deployment pipeline.

**First Live test using SQL Query (“ AND 1 = utl\_inaddr.get\_host\_address ( ( SELECT DISTINCT ( column\_name ) FROM ( SELECT DISTINCT ( column\_name ) , ROWNUM AS LIMIT FROM all\_tab\_columns ) WHERE LIMIT = 5 ) ) AND 'i' = 'i'”)**



**Second live test using rcorrect credentials “username=admin”,  
”password=admin123”**



Third live test using another SQL Query ( `select * from users where id = '1' union select ( \. ) ,@@VERSION -- 1' )` )

