SDN-Based Intelligent Intrusion Detection System (IIDS) Using Machine Learning

Project ID: 24-25J-120

Final Report

Dassanayake E.D.

B.Sc. (Hons) Degree in Information Technology

(Specializing in Cyber Security)

Department of Information Technology

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

SDN-Based Intelligent Intrusion Detection System (IIDS) Using Machine Learning

Project ID: 24-25J-120

Final Report

Dassanayake E.D.
IT21192982

Supervised by Mr. Kanishka Prajeewa Yapa
Co-supervised by Mr. Tharaniyawarma.K

B.Sc. (Hons) Degree in Information Technology

(Specializing in Cyber Security)

Department of Information Technology

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

DECLARATION

I declare that this is my own work, and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Additionally, I hereby grant to Sri Lanka Institute of Information Technology the nonexclusive right to reproduce and distribute my dissertation, in whole or in part, in future works (such as articles or books).

Name	Student ID	Signature
Dassanayake E.D.	IT21192982	[illegible]

The supervisor should certify the proposal report with the following declaration
The above candidate is carrying out research for the undergraduate dissertation

…………………………………   ……………………………...
Signature of the supervisor   Date

………………………………….   ………………………………...
Signature of the co-supervisor   Date

ABSTRACT

In the evolving landscape of cybersecurity, SQL Injection (SQLi) attacks continue to pose a significant threat to data-driven applications. To counteract these threats effectively, this research proposes a comprehensive SQL Injection Detection System Architecture that integrates machine learning techniques with Software-Defined Networking (SDN) capabilities. The system is designed to enhance real-time detection, dynamic mitigation, and adaptive learning against evolving SQLi attacks.

The architecture begins with a **Data Collection Module** that captures incoming requests from web applications, APIs, and direct database accesses. This raw data is fed into a **Preprocessing Module**, where it is cleaned, normalized, and prepared for further analysis. The processed data is then analyzed by the **Machine Learning Engine**, which is trained to detect anomalous behaviors indicative of SQL injection attempts. The **Feedback and Learning Module** enables continuous improvement of the model by incorporating insights from detected incidents to retrain and fine-tune the detection algorithms.

Upon detection of a potential threat, the **Decision-Making Module** evaluates the severity and nature of the attack, guiding the **Response and Mitigation Module** to take appropriate actions. These actions are dynamically enforced through the **SDN Controller**, which allows network-wide security policies to be updated in real time, blocking malicious traffic at its source. The entire operation is overseen by a **Monitoring and Reporting Module** that provides detailed logs, visualizations, and threat intelligence to security administrators, ensuring transparency and accountability.

This layered and modular approach ensures high detection accuracy, swift mitigation responses, and continuous adaptability to new attack vectors. By combining machine learning’s predictive capabilities with the dynamic control of SDN, this system addresses the critical need for intelligent, automated, and scalable defenses against SQL injection attacks in modern network environments.

Keywords: Software-Defined Networking (SDN), Intrusion Detection System (IDS), Machine Learning, SQL Injection, Cybersecurity, Network Security, Security Policy Enforcement

ACKNOWLEDGEMENT

First and foremost, I would like to express my sincere gratitude to my supervisor, Mr. Kanishka Yapa, for their invaluable guidance, encouragement, and continuous support throughout the course of this research. Their insightful feedback and expertise have been instrumental in shaping the direction and depth of my study.

I would also like to thank the academic staff of Sri Lanka Institute of Information Technology, especially the Department of Cybersecurity, for providing a stimulating and supportive environment that greatly contributed to my academic and professional growth. Special appreciation goes to my lecturers and mentors whose teachings and advice have enriched my knowledge and understanding of cybersecurity and machine learning.

I am deeply thankful to my family and friends for their unwavering support, patience, and motivation during this journey. Their belief in me provided the strength I needed to overcome the challenges faced during the research and writing process.

Furthermore, I extend my appreciation to my colleagues and peers for the insightful discussions, collaboration, and moral support throughout this project. Their contributions, whether directly or indirectly, have been valuable to the completion of this work.

Lastly, I would like to acknowledge the developers and contributors of the open-source tools and resources that made the implementation and evaluation of my SQL Injection Detection System possible. Their dedication to open knowledge sharing continues to inspire researchers like me.

This thesis would not have been possible without the support, encouragement, and contributions of all these individuals and institutions. For this, I am profoundly grateful.

Contents

List of Tables

LIST OF ABBREVIATIONS

SQLi	SQL Injection
IDS	Intrusion Detection System
ML	Machine Learning
SDN	Software-Defined Networking
API	Application Programming Interface
DB	Database
DDoS	Distributed Denial of Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
DPI	Deep Packet Inspection
GUI	Graphical User Interface
AI	Artificial Intelligence
QoS	Quality of Service

1. INTRODUCTION

In today's increasingly digital world, web applications, APIs, and database systems form the backbone of many industries, ranging from finance and healthcare to government services. With the proliferation of these technologies, cybersecurity threats have also evolved, targeting the very foundations that support critical operations. Among the various cyber threats, SQL Injection (SQLi) attacks remain one of the most persistent and damaging forms of intrusion. By exploiting vulnerabilities in input validation and database query processing, attackers can gain unauthorized access, manipulate data, or cause complete system failure.

Traditional intrusion detection and prevention systems often rely on static rules and signature-based methods. While effective against known attacks, these approaches struggle to detect novel, sophisticated, or obfuscated SQLi attacks that continuously emerge. As attackers refine their techniques, there is an urgent need for more adaptive, intelligent, and dynamic defense mechanisms that go beyond conventional models.

The integration of Machine Learning (ML) into cybersecurity offers a promising pathway toward intelligent intrusion detection. ML models can learn from historical data, identify complex patterns, and predict malicious activities with a level of sophistication that static systems cannot achieve. Meanwhile, Software-Defined Networking (SDN) provides a flexible and programmable network management framework that can dynamically enforce security policies in real time.

This research proposes a robust SQL Injection Detection System Architecture that synergizes machine learning capabilities with SDN's dynamic control. By creating a multi-layered detection, decision-making, and mitigation framework, the system not only identifies SQLi attacks but also responds effectively to minimize damage, adapts to new threats through continuous learning, and provides transparent monitoring and reporting to security administrators.

The aim of this study is to bridge the gap between static defense mechanisms and intelligent, adaptive cybersecurity solutions, offering a future-proof approach to securing modern web applications and databases against SQLi threats.

1.1 Background

SQL Injection attacks have been a significant cybersecurity concern since they were first identified in the late 1990s. These attacks exploit poorly sanitized input fields in web applications, allowing attackers to inject malicious SQL commands directly into queries executed by a database. The consequences of a successful SQLi attack can be catastrophic, including unauthorized access to sensitive data, data corruption, identity theft, and disruption of services.

Despite advancements in web development practices and security guidelines, SQLi remains prevalent largely due to human error, legacy systems, and the continual expansion of the internet's attack surface. According to the Open Web Application Security Project (OWASP) Top Ten list, injection flaws, including SQLi, consistently rank among the most critical security risks to web applications.

Early defense mechanisms against SQLi focused on input validation, prepared statements, and rule-based detection systems. However, attackers have developed methods to bypass these static defenses by using sophisticated evasion techniques, such as polymorphic injections, encrypted payloads, and blending malicious queries with legitimate traffic. As a result, traditional systems often exhibit high false-negative rates, failing to detect novel or modified attacks.

Machine Learning introduces a paradigm shift in intrusion detection by enabling systems to learn from data and identify anomalies without relying solely on predefined rules. By analyzing patterns of normal and abnormal behavior, ML models can detect subtle deviations indicative of SQLi attempts, even when the attack signatures are unknown. Supervised, unsupervised, and reinforcement learning techniques offer different strategies for building robust detection models capable of evolving alongside emerging threats.

Parallel to the advancement of ML, Software-Defined Networking (SDN) has emerged as a transformative technology in network management. SDN decouples the control plane from the data plane, allowing centralized, programmable management of network resources. This flexibility makes SDN a powerful tool for dynamic threat mitigation, enabling rapid updates to security policies and traffic flows based on real-time intelligence.

Combining Machine Learning’s predictive capabilities with SDN’s dynamic enforcement opens new frontiers for building intelligent, adaptive, and scalable intrusion detection systems. This research leverages this synergy to design an architecture that not only detects SQLi attacks with high accuracy but also responds proactively to contain and mitigate threats, thereby ensuring greater security resilience in modern networked environments.

1.2 Research Gap

Despite the significant progress in intrusion detection systems, especially those addressing SQL Injection (SQLi) attacks, several limitations continue to persist in existing solutions. Traditional signature-based and rule-based systems demonstrate high accuracy against known attacks but struggle to detect new or obfuscated attack patterns. Furthermore, most current solutions lack adaptive learning capabilities, leading to performance degradation over time as attackers refine their tactics.

Recent research has attempted to incorporate machine learning techniques to enhance detection rates. However, many of these models are static, requiring manual retraining with new datasets, which is not feasible in dynamic environments where threats evolve rapidly. Additionally, few systems leverage the programmability and dynamic response potential offered by Software-Defined Networking (SDN). This absence limits the real-time mitigation of threats, allowing malicious activities to propagate within the network before containment measures are applied.

Another critical gap identified is the lack of integrated feedback loops. Many existing systems focus solely on detection without implementing continuous learning mechanisms that allow models to adapt automatically based on new threat intelligence. Furthermore, monitoring and reporting capabilities are often treated as secondary features, resulting in poor visibility and situational awareness for security administrators.

The proposed system addresses these gaps by integrating a machine learning engine with SDN-based dynamic control, a feedback and learning module for adaptive improvements, and a robust monitoring and reporting framework. By combining these features, the proposed system ensures not only high detection accuracy but also proactive, intelligent mitigation and continuous evolution against emerging SQLi threats.

Features	Research A	Research B	Research C	Proposed System
Machine Learning-Based Detection	✓	✓	✗	✓
Real-Time Attack Mitigation via SDN	✗	✗	✓	✓
Adaptive Feedback and Continuous Learning	✗	✗	✓	✓
Monitoring and Reporting Framework	✓	✗	✓	✓
Handling Obfuscated and Evolving Attacks	✗	✓	✗	✓
Dynamic Policy Enforcement	✗	✗	✓	✓

Table 1- Research Gap

1.3 Research Problem

SQL Injection (SQLi) remains one of the most persistent and damaging cybersecurity threats to web applications and database systems. Despite years of awareness and countermeasure development, SQLi attacks continue to exploit vulnerabilities, leading to unauthorized access, data breaches, and service disruptions. Existing detection mechanisms, predominantly signature-based or rule-based systems, are often inadequate against modern, sophisticated SQLi attacks that utilize evasion techniques, dynamic payloads, and polymorphic queries to bypass traditional defenses.

Machine Learning (ML) approaches have introduced significant improvements in detection capabilities by learning patterns of malicious behavior. However, many of the current ML-based systems are static, requiring periodic manual retraining to remain effective. This static nature makes them less responsive to the rapidly changing tactics of attackers. Moreover, most existing systems are focused solely on detection, offering little to no capability for real-time mitigation and automated response.

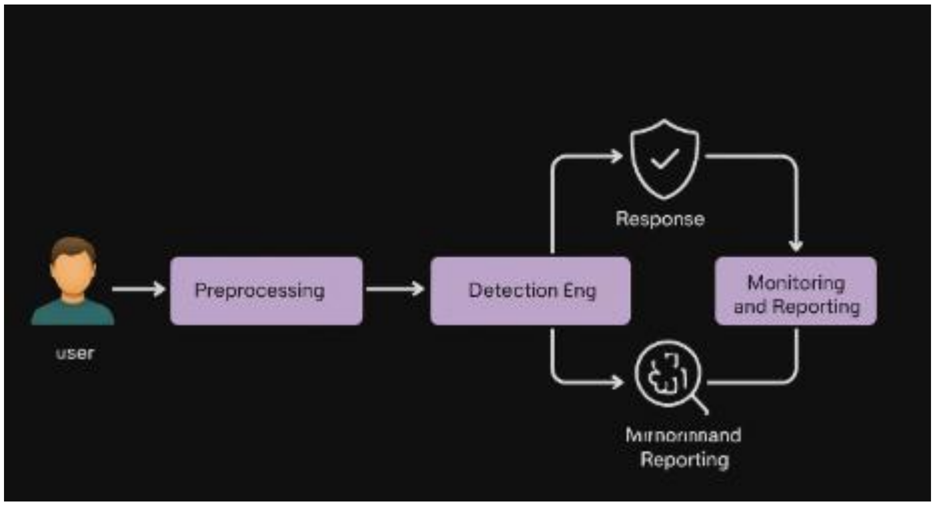


Figure 1 - SQL Injetion Detection System Architecture

Another significant limitation is the lack of dynamic network control in traditional systems. Without integration with technologies like Software-Defined Networking (SDN), most solutions cannot dynamically enforce security policies or isolate malicious traffic in real time. This results in a critical gap between detection and mitigation, giving attackers valuable time to execute their payloads and cause extensive damage before any defensive action is taken.

Furthermore, many current solutions lack a feedback mechanism to continuously learn from newly detected attacks, leading to stagnation in detection capabilities over time. The absence of a robust monitoring and reporting framework also limits the visibility of security operations, making it difficult for administrators to understand the threat landscape and respond effectively.

In summary, the core research problem addressed in this study is the **lack of an intelligent, adaptive, and dynamically responsive SQL Injection Detection and Mitigation System** that can accurately detect evolving SQLi threats, adapt through continuous learning, and enforce real-time defense mechanisms through SDN integration. Bridging this gap is crucial for enhancing the resilience and security posture of modern web applications and databases in an increasingly hostile digital environment.

1.4 Research Objective

1.4.1 Main Objective

The main objective of this research is to develop an intelligent and adaptive SQL Injection Detection System that leverages the predictive capabilities of Machine Learning and the dynamic control offered by Software-Defined Networking (SDN) to enhance the security of modern web applications and databases. The proposed system aims to accurately detect traditional, obfuscated, and emerging SQL Injection (SQLi) attacks by analyzing network and application-level traffic in real time. Upon detection, the system will dynamically mitigate threats by updating network policies through SDN controllers, thus enabling rapid and automated responses to attacks. Furthermore, the system will incorporate a continuous learning mechanism that allows the Machine Learning models to evolve based on newly observed attack patterns, ensuring long-term resilience against evolving threats. Additionally, a comprehensive monitoring and reporting framework will be integrated to provide security administrators with real-time visibility into threat activities and system performance. Overall, the research aspires to bridge the gap between static detection methods and the need for intelligent, adaptive, and scalable cybersecurity solutions.

1.4.2 Specific Objectives

Analyze Existing SQL Injection Detection Techniques

The first step of this research is to conduct a comprehensive analysis of existing SQL Injection (SQLi) detection techniques. Traditional detection mechanisms primarily rely on rule-based systems and signature matching. These approaches work by identifying known malicious patterns or specific keywords commonly associated with SQLi attacks. While effective against well-documented attack vectors, these static methods fail when facing new, modified, or obfuscated SQL injection techniques. Modern attackers often use advanced evasion strategies, such as encoding payloads, inserting comments within injection strings, or dynamically constructing queries to bypass signature-based filters. Machine Learning-based techniques have emerged in response, offering anomaly detection capabilities by learning the difference between normal and malicious behaviors. However, many of these models are still limited by their training datasets and are vulnerable to concept drift — where the nature of normal or malicious activity changes over time. A detailed literature review will be conducted to understand the strengths and weaknesses of these approaches, highlight critical research gaps such as lack of adaptability, limited mitigation capabilities, and poor integration with real-time systems, and justify the need for a more intelligent, dynamic solution that this research aims to deliver.

Develop a Machine Learning-Based Detection Engine

Developing a robust Machine Learning (ML)-based detection engine is a central focus of this research. The detection engine must be capable of accurately distinguishing between legitimate and malicious SQL queries with minimal false positives and false negatives. To achieve this, extensive datasets containing both benign and attack queries will be gathered and preprocessed.

Feature engineering techniques will be applied to extract meaningful patterns, such as query length, special character frequency, and SQL keyword density, which can help the model learn to differentiate between normal and anomalous behavior. Various ML algorithms, such as Random Forests, Support Vector Machines (SVM), and Deep Neural Networks, will be evaluated based on their classification performance. The selected model will undergo rigorous training, validation, and hyperparameter tuning to optimize its predictive power. Additionally, techniques like cross-validation will be employed to ensure the model's generalization to unseen data. The goal is to create a detection engine that not only identifies known SQLi attacks but also has the ability to detect zero-day attacks and adaptive threats by learning behavioral patterns rather than relying solely on fixed signatures.

Incorporate Software-Defined Networking (SDN) for Dynamic Mitigation

Another key innovation of this research is the integration of Software-Defined Networking (SDN) technology to facilitate dynamic threat mitigation. SDN decouples the control plane from the data plane in network devices, allowing centralized, programmable control over network traffic flows. By leveraging SDN, the system can implement real-time security policies based on detection outcomes. For example, when an SQLi attack is detected, the SDN controller can be instructed to block traffic from the offending IP address, reroute suspicious sessions to a honeypot for further analysis, or quarantine compromised hosts. This dynamic, programmable approach ensures that threats are contained before they can propagate further into the network. SDN’s centralized view of the network also enables better traffic analysis and fine-grained policy enforcement compared to traditional static network architectures. Integrating SDN into the system architecture ensures a closed-loop response mechanism, where detection, decision-making, and mitigation are interconnected in real time, significantly reducing the window of opportunity for attackers.

Establish a Feedback and Continuous Learning Module

Traditional intrusion detection systems often become outdated over time, especially when attackers modify their strategies. To overcome this limitation, the research proposes the development of a Feedback and Continuous Learning Module. This component will collect data from newly detected SQL Injection attempts, successful or otherwise, and use it to periodically retrain and fine-tune the Machine Learning model. By implementing mechanisms such as online learning, incremental model updates, and reinforcement learning techniques, the system will remain resilient and adaptive to changing threat landscapes. The feedback loop will also allow administrators to manually label new types of attacks, thereby continuously enriching the training dataset. Continuous learning ensures that the system’s detection capabilities do not degrade over time and remain effective against novel SQLi attacks. It also reduces the reliance on manual updates and signature writing, thus enabling a self-evolving security system that keeps pace with the evolving threat environment.

Design a Comprehensive Monitoring and Reporting Framework

Visibility and transparency are critical in cybersecurity operations. To provide administrators with actionable intelligence, a comprehensive Monitoring and Reporting Framework will be designed as part of the system. This module will visualize real-time detection events, mitigation actions taken, system performance metrics, and historical trends through intuitive dashboards. It will generate alerts for high-severity incidents and allow administrators to drill down into detailed logs for forensic analysis. Periodic reports summarizing system activity, attack patterns, and learning outcomes will also be produced to assist in strategic planning and compliance auditing. Furthermore, customizable reporting capabilities will allow the system to adapt to different organizational needs. By offering real-time situational awareness and historical insight, the Monitoring and Reporting Framework will empower security teams to make informed decisions, prioritize resources effectively, and maintain a proactive security posture.

Evaluate System Performance

The final specific objective is to rigorously evaluate the performance of the proposed SQL Injection Detection System. Experimental testing will be conducted using benchmark datasets as well as simulated real-world scenarios to assess the system's detection accuracy, false positive rate, mitigation speed, and adaptability to new attack vectors. Comparative analyses against existing state-of-the-art detection systems will be carried out to demonstrate the improvements achieved through the integration of Machine Learning, SDN, and continuous learning modules. Key performance indicators such as precision, recall, F1-score, mean time to detect (MTTD), and mean time to mitigate (MTTM) will be measured. The scalability of the system will also be tested by deploying it in different network environments, including high-traffic conditions, to ensure it can handle enterprise-level workloads. Through comprehensive evaluation, the research aims to validate the practical applicability, reliability, and robustness of the developed system in protecting modern web applications and databases against SQL Injection attacks.

2. METHODOLOGY

This research adopts a systematic methodology to design and develop an intelligent SQL Injection Detection System that combines Machine Learning and Software-Defined Networking (SDN) technologies. Initially, a modular system architecture was designed, consisting of a Data Collection Module, Preprocessing Module, Machine Learning Detection Engine, Decision-Making Module, SDN-based Mitigation Module, Feedback and Continuous Learning Module, and a Monitoring and Reporting Framework. Data collection involved gathering datasets containing both legitimate and malicious SQL queries from publicly available sources and simulated attack environments. The data were preprocessed through cleaning, normalization, and feature extraction to prepare them for model training. Various Machine Learning algorithms, including Random Forests, Support Vector Machines (SVM), and Long Short-Term Memory (LSTM) networks, were evaluated to identify the most effective detection model, with cross-validation techniques used to ensure generalizability. The trained detection engine was integrated with an SDN controller, enabling dynamic network policy enforcement to block or reroute malicious traffic in real time upon detection. A Feedback and Continuous Learning Module was implemented to allow the model to update and adapt to emerging threats automatically, reducing the need for manual retraining. The Monitoring and Reporting Framework provided real-time dashboards and periodic reports to enhance threat visibility for administrators. Finally, the system's performance was rigorously evaluated using experimental testing, measuring key metrics such as detection accuracy, false positive rates, mitigation speed, and adaptability, with comparative analysis against existing methods to validate the effectiveness of the proposed solution.

2.1 System Architecture Diagram

2.1.1 Overall System Diagram

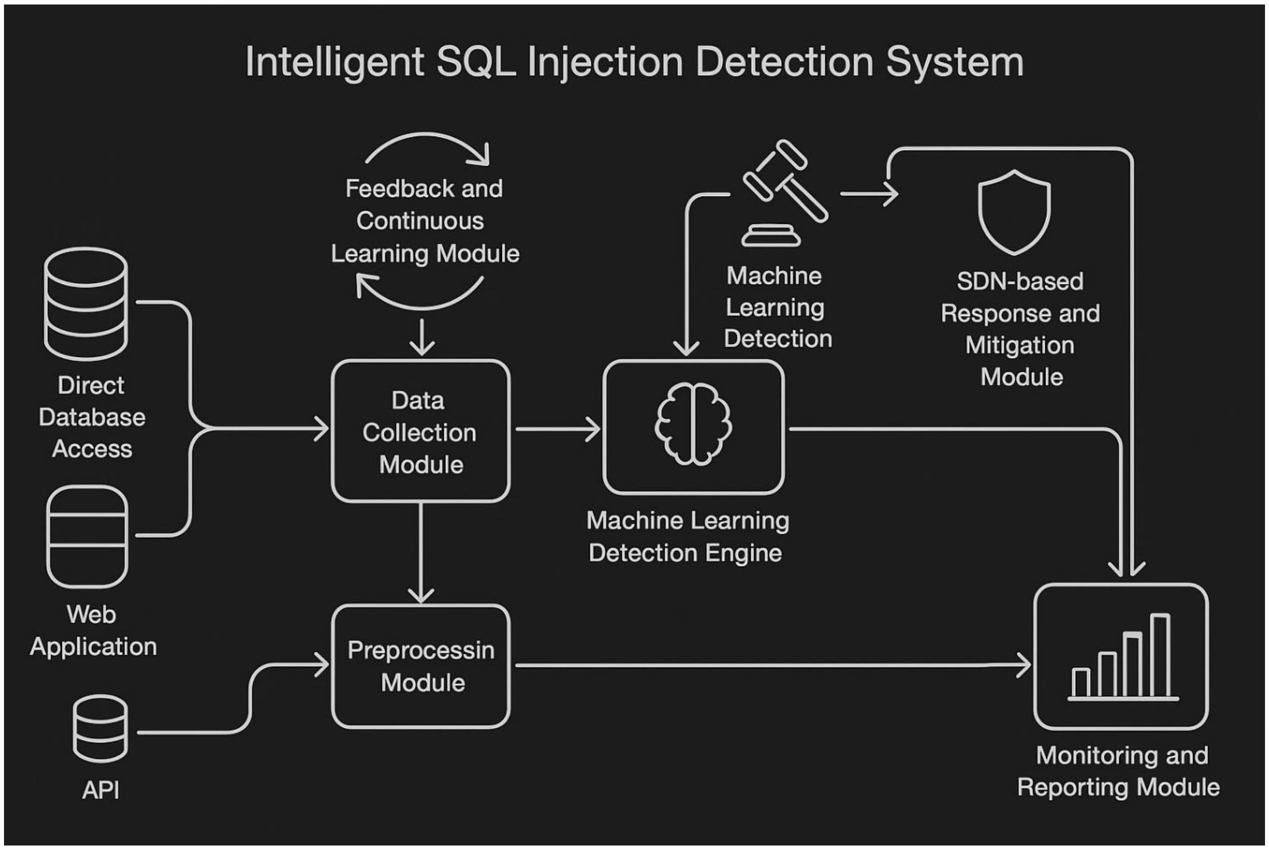

Intelligent SQL Injection Detection System
Feedback and Continuous Learning Module
Machine Learning Detection
SDN-based Response and Mitigation Module
Direct Database Access
Data Collection Module
Machine Learning Detection Engine
Web Application
Preprocessin Module
API
Monitoring and Reporting Module

Figure 2 - Intelligent SQL Injection Detection System

The overall system diagram of the proposed SQL Injection Detection System outlines the interaction between its core modules and their flow of operations, ensuring an intelligent, adaptive, and dynamic security response. The architecture is designed to efficiently detect SQL Injection (SQLi) attacks, make intelligent decisions, and enforce mitigation strategies through Software-Defined Networking (SDN), while continuously learning from new threats.

The process begins with the **Data Collection Module**, which captures incoming web traffic, API requests, and direct database queries from the application environment. This raw data is a mixture of legitimate user inputs and potential SQL injection attempts. The collected data is then passed to the **Preprocessing Module**, where it undergoes cleaning, normalization, and feature extraction. Preprocessing ensures that the data is structured in a form suitable for analysis by the Machine Learning model, eliminating noise and highlighting critical characteristics such as SQL keywords, special characters, query length, and input patterns.

Once preprocessed, the data enters the **Machine Learning Detection Engine**, which acts as the heart of the system. This engine, powered by a trained machine learning model (such as Random Forest, SVM, or LSTM), analyzes each request in real-time and classifies it as either benign or malicious. If a potential SQLi attack is detected, the classified result is sent to the **Decision-Making Module**. This module evaluates the severity of the threat based on pre-set thresholds, contextual information, and risk assessment policies.

Based on the decision outcome, actions are triggered by the **SDN-based Response and Mitigation Module**. Here, an SDN controller dynamically enforces security policies at the network level. It can block the malicious traffic source, quarantine the affected segment, or reroute suspicious activities to a honeypot for further analysis. The integration of SDN ensures that responses are immediate, programmable, and scalable across the entire network infrastructure.

Parallel to the detection and mitigation process, the system feeds events into the **Feedback and Continuous Learning Module**. This module collects data from new attack patterns, false positives, and evolving SQLi techniques to update the machine learning model periodically or incrementally. This continuous learning process ensures that the system adapts to new threats without requiring complete retraining, maintaining its effectiveness over time.

To maintain operational visibility, the **Monitoring and Reporting Module** records all detection events, responses, system performance metrics, and historical trends. It provides administrators with real-time dashboards, threat visualizations, incident alerts, and periodic summary reports. This module ensures that security teams have complete awareness of system status, detected threats, and ongoing mitigation efforts.

Overall, the system’s modular design ensures that each component interacts seamlessly, allowing the entire architecture to function as an intelligent, responsive, and evolving defense mechanism against SQL Injection attacks. The integration of machine learning for detection, SDN for dynamic mitigation, and feedback for continuous improvement positions the system as a robust solution capable of securing modern web applications against increasingly sophisticated cyber threats.

2.1.2 Component System Diagram

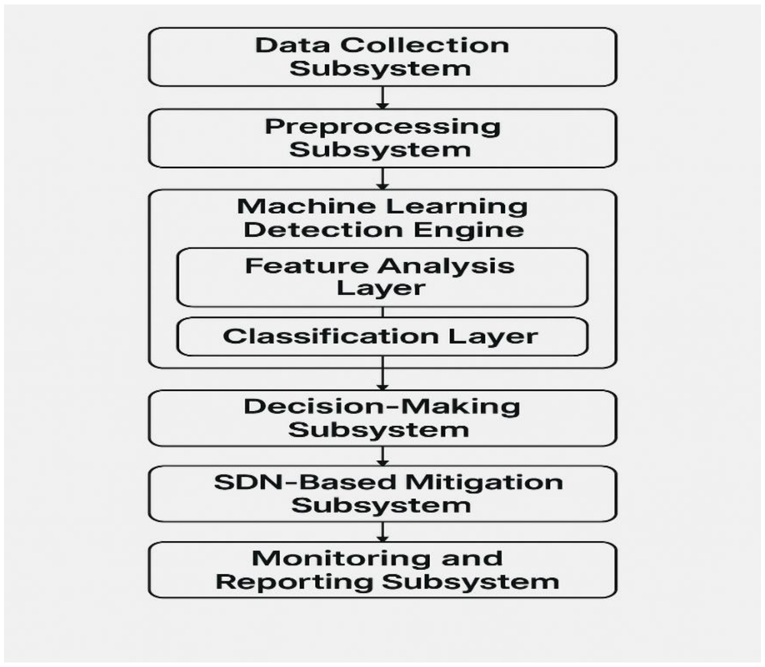

Data Collection Subsystem
Preprocessing Subsystem
Machine Learning Detection Engine
Feature Analysis Layer
Classification Layer
Decision-Making Subsystem
SDN-Based Mitigation Subsystem
Monitoring and Reporting Subsystem

Figure 3 - Component System Diagram

The Component System Diagram provides a more detailed view of the internal workings of the intelligent SQL Injection Detection System. It elaborates on how each module interacts with its sub-components to ensure seamless data flow, efficient detection, and dynamic threat mitigation.

The first component is the **Data Collection Subsystem**, which is responsible for capturing data from various input sources, such as web applications, APIs, and direct database access points. This subsystem ensures that all incoming traffic is monitored without introducing significant latency or bottlenecks. It utilizes packet sniffers, logging tools, or network taps to collect the raw input for analysis.

Once data is collected, it is sent to the **Preprocessing Subsystem**. This subsystem cleans the data by removing noise, normalizes query structures, tokenizes SQL statements, and extracts important features such as keyword frequency, special character patterns, and input length. Preprocessing ensures that the Machine Learning Detection Engine receives structured and meaningful data, improving the overall detection accuracy.

The **Machine Learning Detection Engine** itself is subdivided into two main layers: the Feature Analysis Layer and the Classification Layer. The Feature Analysis Layer processes the input features, while the Classification Layer uses trained machine learning models (such as Random Forests, SVMs, or LSTMs) to classify the input as either benign or malicious.

Following detection, the **Decision-Making Subsystem** assesses the severity and context of the classified threat. It determines whether an automatic mitigation action should be taken immediately or if manual intervention is needed. The Decision Engine uses predefined threat levels and scoring mechanisms to guide these choices.

The **SDN-Based Mitigation Subsystem** interacts directly with the Software-Defined Networking Controller. It dynamically enforces network policies, such as blocking IP addresses, isolating malicious users, or redirecting suspicious sessions to honeypots for further investigation.

In parallel, the **Feedback and Learning Subsystem** monitors detection outcomes and updates the Machine Learning model over time. It collects mislabeled instances, new types of SQLi payloads, and administrator feedback to continually refine detection capabilities.

Finally, the **Monitoring and Reporting Subsystem** aggregates data from all components and visualizes it through dashboards and reports. This subsystem ensures transparency, enabling security administrators to track system performance, view active threats, and review mitigation actions in real-time.

Together, these components create a cohesive, intelligent system capable of detecting, responding to, and learning from SQL Injection attacks in a dynamic and scalable manner.

2.2 Individual System Diagram

The Individual System Diagram provides a granular view of each module within the intelligent SQL Injection Detection System, highlighting how internal components operate independently while maintaining cohesion across the entire system. This detailed breakdown ensures that every module can be optimized for performance, security, and scalability. By isolating responsibilities within modules, the system achieves modularity, making it easier to maintain, upgrade, and extend individual functionalities without disrupting the entire architecture. Each module is designed to perform specific tasks effectively, ensuring a robust and resilient pipeline from data collection to final reporting and learning.

The first critical component is the **Data Collection and Preprocessing Unit**. In the Individual System Diagram, this unit consists of two primary internal systems: the data sniffer/logger and the data sanitizer/normalizer. The sniffer captures all incoming queries and user interactions from various entry points such as web applications, APIs, or database interfaces. This captured data is raw and often noisy, so it passes immediately to the sanitizer, where invalid or malformed queries are removed, and the remaining data is standardized. Tokenization and feature extraction mechanisms work here to convert the raw input into a structured feature set, suitable for machine learning analysis. This structured data format ensures consistency and enhances the detection engine's capability to differentiate between benign and malicious queries with high precision.

Next, the **Machine Learning Detection Engine** is visualized as two deeply connected layers within its subsystem: the feature processing layer and the classification layer. The feature processing layer analyzes incoming preprocessed data, further refining and mapping it into high-dimensional feature spaces. The classification layer then applies pre-trained Machine Learning models such as Random Forests, SVM, or LSTM to determine the probability of the input being a SQL Injection attack. Each detection decision is assigned a confidence score. If the score exceeds a defined threshold, the detection engine forwards the result to the Decision-Making Module. This clear separation of feature processing and classification functions ensures the flexibility to upgrade algorithms independently and improve the overall accuracy of detection over time.

Following the detection, the **Decision-Making and SDN Mitigation Unit** takes over. This unit internally consists of a threat assessor and a mitigation action handler. The threat assessor analyzes the output from the detection engine, evaluates the severity of the threat based on dynamic risk scoring models, and chooses an appropriate response strategy. The mitigation action handler communicates directly with the SDN controller to enforce network-level defenses. Actions such as blocking IP addresses, isolating network segments, or redirecting suspicious sessions to honeypots are programmed here. Meanwhile, all detected incidents are logged and transferred to the Feedback and Learning Unit. The internal design of this mitigation unit allows dynamic and flexible network defense strategies to be implemented in real time, ensuring that attackers are contained quickly and effectively.

Lastly, the **Feedback, Learning, Monitoring, and Reporting Unit** is composed of three subsystems: a continuous feedback collector, a retraining engine, and a visualization dashboard. The feedback collector aggregates detection outcomes, false positives/negatives, and administrator-labeled incidents, creating new datasets for model improvement. The retraining engine can trigger model updates either automatically or under human supervision, ensuring continuous adaptation to new SQLi attack vectors. Meanwhile, the visualization dashboard provides real-time status updates, trend analysis, detection logs, and system health indicators. Security administrators can use these dashboards to monitor current threats, investigate past incidents, and make strategic security decisions. By isolating feedback collection, model learning, and monitoring into distinct but interconnected systems, the architecture supports ongoing system evolution without disrupting daily operations, thus achieving a sustainable, intelligent cybersecurity solution.

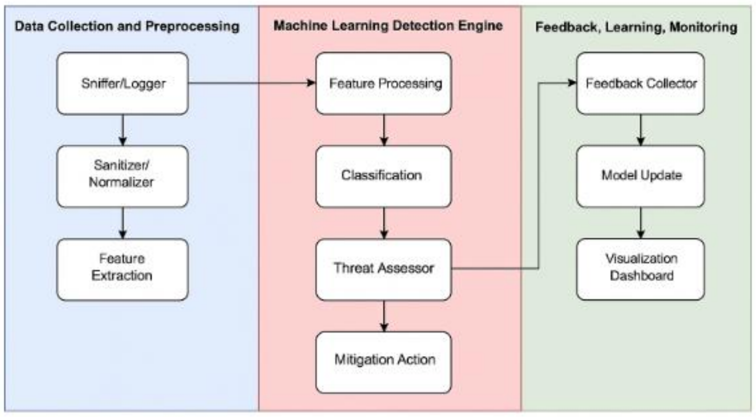


Figure 4 - Model Diagram

2.2.1 Dataset

A fundamental aspect of developing an effective SQL Injection Detection System is the availability and quality of the dataset used for training, validating, and testing the Machine Learning models. In this research, the dataset plays a crucial role as it forms the basis for the model's ability to learn and generalize the distinction between benign SQL queries and malicious injection attempts. To ensure a robust and comprehensive learning process, datasets containing diverse and representative samples of both legitimate and malicious queries are necessary. These datasets are gathered from a combination of publicly available sources, simulated attack environments, and synthetically generated SQL Injection samples. Public datasets provide a good starting point as they often contain labeled examples of real-world attacks, while simulated environments enable the creation of customized attack patterns, helping the system learn to detect emerging and sophisticated SQLi techniques.

The dataset creation process involves several key stages: collection, preprocessing, labeling, and augmentation. During the collection phase, data is harvested from web server logs, database transaction logs, and honeypot environments designed to attract SQL Injection attacks. Additionally, various SQL Injection payloads sourced from penetration testing tools and security research repositories are incorporated to enhance the diversity of the attack patterns. Preprocessing involves cleaning the raw data by removing irrelevant or duplicate entries and normalizing query structures to maintain consistency across samples. This step is essential to eliminate noise that could degrade the performance of the Machine Learning model. Feature extraction techniques are applied to convert each query into a structured format that captures important characteristics such as the frequency of SQL keywords, special character usage, input length, query structure anomalies, and other syntactic patterns commonly associated with SQL Injection attempts.

Labeling the dataset accurately is vital for supervised learning. Each query is labeled as either "benign" or "malicious," with additional metadata such as attack type (e.g., tautology-based, union-based, blind SQLi) where applicable. Manual inspection and verification are employed to ensure labeling accuracy, especially for ambiguous queries. Furthermore, dataset augmentation techniques are used to artificially expand the dataset size and balance the class distribution. This involves creating slightly modified versions of existing queries through methods such as token substitution, query obfuscation, and introducing random noise. These augmentation strategies help the Machine Learning model become more resilient to variations in attack patterns and reduce the risk of overfitting.

The final dataset is then split into three subsets: training, validation, and testing. Typically, 70% of the data is used for training the model, 15% is reserved for validation during hyperparameter tuning, and the remaining 15% is used for final performance evaluation. This division ensures that the model is tested on unseen data, providing a realistic estimate of its detection capabilities in real-world scenarios. To further enhance robustness, stratified sampling is used during splitting to maintain the original class distribution across all subsets. Additionally, cross-validation techniques such as k-fold validation are employed during the model development phase to ensure that performance metrics are not biased by any particular data partition.

In conclusion, the dataset development process for this research is meticulously designed to provide a rich, diverse, and high-quality training ground for the SQL Injection Detection System. By combining real-world samples, simulated attacks, and synthetic data augmentation, the dataset ensures that the Machine Learning model is well-prepared to detect a wide range of SQL Injection threats, including novel and obfuscated attacks. The careful preprocessing, accurate labeling, and systematic evaluation setup contribute significantly to the reliability, accuracy, and generalizability of the proposed system.

2.2.2 Preprocessing and Feature Extraction

Preprocessing and feature extraction are critical stages in building an effective Machine Learning-based SQL Injection Detection System. These stages transform raw and unstructured input queries into a structured and meaningful format that Machine Learning models can interpret and learn from effectively. Without proper preprocessing, the raw data would contain noise, inconsistencies, and irrelevant patterns that could severely degrade the accuracy and reliability of the detection system. Therefore, the preprocessing phase is meticulously designed to clean, normalize, and standardize the collected data to ensure its quality and relevance for the subsequent modeling phase.

The preprocessing process begins with **data cleaning**, where incomplete, duplicate, or malformed queries are removed from the dataset. Cleaning is essential because corrupted or irrelevant data points can introduce bias and noise, confusing the learning algorithm. Following cleaning, **normalization** techniques are applied to ensure consistency across all queries. Since SQL queries can vary widely in structure and format, normalization involves tasks such as converting all queries to lowercase, removing excessive whitespace, and replacing dynamic values (like user input parameters) with placeholders. This step helps the model focus on the structural patterns of queries rather than superficial differences such as case sensitivity or random variable names. Additionally, **tokenization** is performed, where each query is broken down into a sequence of tokens—individual elements like SQL keywords, operators, and literals. Tokenization simplifies the representation of queries and enables the system to analyze and model their syntactic structure more efficiently.

Once the data has been cleaned and normalized, **feature extraction** is carried out to derive meaningful attributes that can capture the underlying characteristics of SQL Injection attacks. Feature extraction transforms each tokenized query into a vector of numerical values that represent different aspects of the query. Commonly extracted features include the frequency of SQL keywords (e.g., SELECT, UNION, DROP), the presence of special characters (e.g., single quotes, semicolons, comment symbols), the overall length of the query, the ratio of alphabetic to non-alphabetic characters, and the sequence or order of operations within the query. Special attention is given to features that are indicative of SQL Injection attempts, such as excessive use of logical operators (AND, OR), nested queries, and unusual syntax patterns. Feature extraction also includes the use of n-gram models, where contiguous sequences of n tokens are analyzed to capture common patterns and contextual relationships within queries that may signal malicious behavior.

Another critical aspect of feature extraction is the generation of **syntactic and semantic features**. Syntactic features focus on the structure of the query, such as how different SQL clauses are ordered and nested, while semantic features aim to understand the intent of the query by analyzing relationships between different components. Techniques like one-hot encoding or embedding layers (in deep learning models) are used to convert these features into numerical formats suitable for input into Machine Learning algorithms. Dimensionality reduction techniques, such as Principal Component Analysis (PCA), may also be applied if the feature space becomes excessively large, ensuring that the model remains computationally efficient while retaining the most informative attributes.

In conclusion, preprocessing and feature extraction are foundational steps that directly impact the effectiveness of the Machine Learning model. By meticulously cleaning, normalizing, tokenizing, and transforming raw SQL queries into rich feature vectors, the system is equipped with the necessary tools to differentiate between benign and malicious behaviors accurately. This rigorous preparation phase not only enhances detection accuracy but also improves model generalization, enabling the system to identify a wide range of SQL Injection attacks, including those using obfuscation or novel evasion techniques.

2.2.3 Model Architecture and Training

The design of the model architecture and the training process are pivotal elements in the development of an intelligent SQL Injection Detection System. The model must be capable of learning complex patterns from the extracted features and generalizing well to detect both known and unseen SQL Injection (SQLi) attack variants. Based on the nature of the data and the requirements for real-time detection, this research focuses on employing a supervised Machine Learning approach, exploring several algorithms such as Random Forests, Support Vector Machines (SVM), and Long Short-Term Memory (LSTM) networks, each offering unique advantages in classification tasks

The first model architecture explored is the **Random Forest** classifier. Random Forests are ensemble learning methods that operate by constructing multiple decision trees during training and outputting the class that is the mode of the classes predicted by individual trees. This method is highly effective in dealing with high-dimensional data and reducing overfitting through bagging (bootstrap aggregating) techniques. In the context of SQL Injection detection, Random Forests excel at handling the diverse and non-linear relationships between the features extracted from queries. The architecture includes hundreds of trees with controlled depth, ensuring balance between bias and variance. Each tree receives a random subset of features, which forces the model to learn a broad range of patterns rather than relying on a few dominant ones.

In addition to Random Forests, **Support Vector Machines (SVMs)** are utilized, particularly effective in binary classification tasks such as distinguishing between benign and malicious queries. SVMs work by finding the hyperplane that best separates the two classes in the feature space. By using different kernel functions (linear, polynomial, or radial basis function kernels), SVMs can capture complex, non-linear relationships within the data. This flexibility makes SVMs a strong candidate for detecting sophisticated or obfuscated SQL Injection attacks. SVMs are trained with regularization parameters carefully tuned to prevent overfitting while ensuring maximum margin separation between classes.

For sequence modeling and capturing deeper contextual relationships within queries, **Long Short-Term Memory (LSTM)** networks are explored. LSTMs, a special kind of recurrent neural network (RNN), are capable of learning long-term dependencies and are particularly useful when the order of tokens in a query matters — for example, when certain sequences of SQL commands and operators indicate a malicious intent. The LSTM architecture comprises input, forget, and output gates that regulate the flow of information, allowing the model to selectively remember or forget parts of the sequence. For this research, a two-layer LSTM model is designed with dropout regularization to prevent overfitting and a fully connected dense layer at the output for binary classification. LSTMs provide an advantage when detecting multi-stage or heavily obfuscated SQLi attempts that static models might miss.

Sr No	**Layer Type**	**Units/Filters**	**Activation**
1	Input Layer	768-Dimensional Input Vector	None
2	Fully Connected Layer	64 Units	ReLU
3	Batch Normalization	-	None
4	Dropout Layer	0.2	None
5	Fully Connected Layer	32 Units	ReLU
6	Batch Normalization	-	None
7	Dropout Layer	1	None
8	Output Layer	1 Unit	Sigmoid

Table 2 - Model Architecture Summary

The training process follows a structured pipeline. The dataset is split into training, validation, and testing subsets using an 80:10:10 ratio. Training is conducted in batches to improve convergence speed and stability. Hyperparameters such as learning rate, tree depth (for Random Forests), C value (for SVMs), and number of hidden units and layers (for LSTMs) are optimized using grid search and cross-validation. During training, loss functions such as binary cross-entropy are minimized, and early stopping techniques are employed to halt training when the validation performance ceases to improve, thereby preventing overfitting. Model evaluation metrics, including accuracy, precision, recall, F1-score, and area under the ROC curve (AUC), are continuously monitored to ensure robust and fair performance across different types of SQL Injection attacks.

In conclusion, the model architecture and training process are carefully designed to ensure high detection accuracy, adaptability to new threats, and computational efficiency suitable for real-time environments. The combined use of ensemble methods, margin-based classifiers, and deep learning architectures equips the system with a versatile and powerful foundation to detect and respond to SQL Injection attacks effectively.

Parameter Type	**Count**
Total Params	68,801
Trainable Params	68,801
Non-Trainable Params	0

Table 3 - Model Parameters

2.2.4 Model Prediction and Report Generation

Once the Machine Learning model has been trained and validated, it is integrated into the SQL Injection Detection System for real-time prediction and reporting. The **Model Prediction** phase is responsible for classifying incoming SQL queries as either benign or malicious based on the features extracted during preprocessing. Each new query entering the system is first passed through the feature extraction pipeline to transform it into the appropriate format. The extracted feature vector is then fed into the trained Machine Learning model, which outputs a prediction score. In binary classification settings, this score represents the probability that the input query is malicious. A decision threshold, typically set at 0.5, is used to determine the final classification: queries with prediction scores above the threshold are classified as SQL Injection attempts, while those below are treated as legitimate. However, to enhance sensitivity to sophisticated attacks, the threshold can be dynamically adjusted based on system requirements, such as prioritizing low false negatives in highly secure environments.

In addition to the binary classification result, the system generates a **confidence score** for each prediction, indicating the model's certainty. This score is crucial for decision-making, especially in cases where automated mitigation actions are triggered. For example, high-confidence malicious detections can automatically lead to network isolation or traffic blocking through the SDN controller, while lower-confidence detections may only generate alerts for manual review. The Model Prediction module is optimized for low latency to ensure that decisions are made quickly without disrupting the user experience or application performance. Techniques such as model pruning, batching of inputs, and hardware acceleration (e.g., using GPUs) are employed to maintain high-speed inference even under heavy traffic loads.

Following the prediction phase, the system transitions to the **Report Generation** phase. Every detection event—whether classified as benign or malicious—is logged along with key metadata such as timestamp, source IP address, type of SQL query, confidence score, and mitigation action taken (if any). This data is aggregated and processed by the Monitoring and Reporting Module to create real-time dashboards and periodic summary reports. The reporting framework is designed to provide security administrators with clear, actionable insights into the system's activities. Real-time dashboards visualize the volume of incoming queries, the number of detections, attack trends, and system health metrics using graphs, charts, and alerts. These dashboards allow administrators to monitor ongoing threats, system performance, and response effectiveness at a glance.

Periodic reports are generated on a daily, weekly, or monthly basis, depending on organizational needs. These reports summarize the system's performance, including detection accuracy, false positive and false negative rates, types of SQL Injection attacks detected, and the effectiveness of mitigation strategies. Reports also include recommendations for system improvements based on observed attack patterns and detection outcomes. Additionally, forensic logs are maintained for deeper investigation of incidents, enabling analysts to trace the progression of an attack, understand its impact, and refine defense strategies.

Overall, the Model Prediction and Report Generation components are critical for operationalizing the detection system, providing timely classification of threats, automated or semi-automated mitigation, and transparent reporting for security management. Together, they ensure that the SQL Injection Detection System not only identifies and responds to threats effectively but also empowers security teams with the information needed to maintain a strong and adaptive security posture.

2.3 Model Deployment

After the successful development, training, and testing of the Machine Learning model for SQL Injection detection, the next crucial step is its deployment into a real-world environment. Model deployment transforms the trained model from a research artifact into a live service capable of protecting web applications and databases against SQL Injection (SQLi) threats in real time. The goal of deployment is to ensure that the model can handle production-level traffic, make rapid predictions with minimal latency, and integrate seamlessly with the broader system architecture, including the SDN controller, feedback loops, and monitoring tools.

To achieve this, the model is encapsulated within a scalable and reliable API framework, such as **FastAPI** or **Flask**, which exposes the model's prediction functionality as a web service. Incoming SQL queries are sent to the API, which processes the request, extracts features, applies the trained model, and returns a classification result (benign or malicious) along with a confidence score. The API is designed to be lightweight, stateless, and capable of horizontal scaling to handle increased load. For scalability and high availability, the deployment is managed using containerization technologies like **Docker** and orchestrated using **Kubernetes** clusters. This containerized setup allows the system to be easily replicated, monitored, and updated without downtime.

Furthermore, the deployed system includes mechanisms for real-time communication with the SDN controller. When a malicious query is detected, the API sends instructions to the SDN layer to enforce immediate network-level responses such as blocking the attacker's IP, terminating the session, or rerouting traffic. Logging and monitoring components are integrated into the deployment to track system performance, prediction accuracy, and operational health. Security measures, such as HTTPS encryption, API authentication, and access control, are enforced to protect the model and its interfaces from external threats.

Periodic retraining and model updates are critical in a dynamic threat landscape. Therefore, the deployment pipeline includes a Continuous Integration/Continuous Deployment (CI/CD) setup that allows updated models to be validated and rolled out seamlessly. This ensures that improvements based on the Feedback and Learning Module can be quickly integrated into the live system without service disruption.

2.3.1 Deployment Architecture Overview

The deployment architecture of the SQL Injection Detection System is designed to ensure high performance, scalability, resilience, and integration with security response mechanisms. It consists of several interconnected layers, each serving a specific purpose to maintain smooth and secure operations.

At the **front-end layer**, incoming SQL queries generated by web applications, APIs, or database management systems are captured. These queries are directed to the **Feature Extraction Service**, a lightweight microservice responsible for preprocessing and transforming raw inputs into feature vectors compatible with the Machine Learning model. The decoupling of feature extraction as a service allows preprocessing to be scaled independently based on incoming traffic volume.

Next, the feature vectors are forwarded to the **Model Inference Engine**, a dedicated service that hosts the trained Machine Learning model. The Model Inference Engine handles prediction requests, evaluates the feature vectors, and returns a classification output along with a confidence score. This engine is designed to handle high concurrency and low-latency requirements by leveraging optimized inference runtimes, model quantization techniques, and GPU acceleration where necessary.

Upon classification, the output is sent to the **Decision-Making and SDN Integration Layer**. If the result indicates a malicious query, this layer triggers automated mitigation actions via APIs connected to the **Software-Defined Networking Controller**. The SDN controller dynamically adjusts network flow rules to block or isolate malicious sources based on the instructions received.

Simultaneously, every prediction and mitigation action is logged in the **Monitoring and Reporting System**. This system collects telemetry data, system health metrics, and threat intelligence in real time. Visualization tools such as **Grafana** or custom-built dashboards are employed to present system performance, detection trends, and alerts to administrators. Logs are also archived for compliance, auditing, and forensic analysis.

The entire deployment is containerized using **Docker** and orchestrated using **Kubernetes**, allowing services to be distributed across multiple nodes for fault tolerance and load balancing. Kubernetes also manages auto-scaling based on traffic demand and handles service recovery in case of failure. Secure communication between services is enforced using TLS encryption, and access is controlled through role-based access control (RBAC) policies.

Additionally, a **CI/CD Pipeline** is integrated into the architecture to automate testing, validation, and deployment of new models or system updates. This pipeline ensures that improvements derived from the Feedback and Learning Module, such as retrained models or system patches, can be deployed with minimal human intervention and without downtime.

Overall, the deployment architecture is designed to ensure that the intelligent SQL Injection Detection System operates reliably under real-world conditions, providing rapid threat detection, dynamic mitigation, continuous monitoring, and adaptive learning capabilities, all within a scalable and secure environment

2.3.2 API Development for Model Integration

The API (Application Programming Interface) development for model integration is a crucial step in operationalizing the Machine Learning-based SQL Injection Detection System. The API acts as the communication bridge between external systems — such as web applications, firewalls, and the Software-Defined Networking (SDN) controller — and the Machine Learning model that detects potential SQL Injection (SQLi) attacks. A well-designed API ensures that the deployed model is accessible, scalable, secure, and capable of delivering real-time predictions to protect critical application infrastructure.

The API is developed using a lightweight, high-performance framework such as **FastAPI** or **Flask**. FastAPI is particularly preferred due to its asynchronous request handling, automatic data validation, and built-in documentation generation through OpenAPI (Swagger). The main endpoint of the API accepts incoming HTTP POST requests containing SQL query data in JSON format. Each request carries the necessary payload, such as the raw SQL query, metadata (e.g., user ID, session ID, or IP address), and authentication tokens if required. The API first validates the incoming request to ensure proper structure, authorized access, and safe data formats, mitigating risks such as malformed inputs or unauthorized queries.

Upon successful validation, the API routes the query to the **Preprocessing and Feature Extraction Module**, where the input is cleaned, normalized, tokenized, and transformed into a feature vector compatible with the model's requirements. This modular design ensures that any updates to the preprocessing pipeline can be deployed without major changes to the overall API structure. After feature extraction, the feature vector is fed into the **Model Inference Engine**, which performs real-time prediction and returns a classification output along with a confidence score.

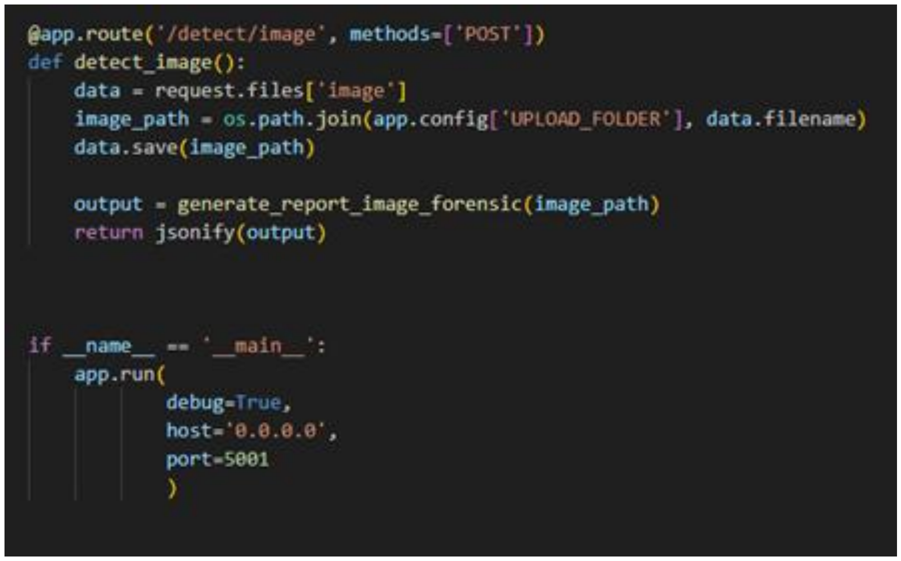
The API then interprets the model's output. If the query is classified as malicious, the API can automatically trigger actions such as sending alerts, updating logs, or instructing the SDN controller to implement immediate mitigation strategies like blocking or rerouting malicious traffic. Response objects are structured in a standardized JSON format, including fields such as prediction result (benign or malicious), confidence score (e.g., 92.5%), recommended action, and timestamps. The API is optimized for low-latency performance to support environments where high query throughput and fast response times are critical.

Security is a fundamental aspect of the API development process. Secure communication is enforced using **HTTPS/TLS** protocols to encrypt data in transit. Additionally, **authentication mechanisms** such as API keys, OAuth 2.0, or JWT (JSON Web Tokens) are implemented to restrict access to authorized clients only. Rate limiting and input validation are applied to protect against API abuse, denial-of-service attacks, and injection-based attacks targeting the API itself.

Scalability and reliability are achieved by containerizing the API service using **Docker** and deploying it within a **Kubernetes** cluster. Kubernetes manages load balancing, auto-scaling, and fault tolerance, ensuring the API remains available and responsive even under heavy load conditions. Health check endpoints are also implemented to allow Kubernetes to monitor service health and automatically restart unhealthy instances.

In addition, comprehensive **logging and monitoring** features are embedded into the API. All incoming requests, prediction results, response times, and errors are logged systematically. These logs feed into centralized monitoring platforms such as **Prometheus** and **Grafana**, enabling real-time visibility into API performance and rapid detection of anomalies.

In conclusion, the API development for model integration transforms the Machine Learning model into a production-ready service that is accessible, scalable, secure, and efficient. It provides a critical interface that not only powers real-time SQL Injection detection but also seamlessly integrates with security operations and response mechanisms, forming the backbone of a fully operational intelligent cybersecurity system.



```
@app.route('/detect/image', methods=['POST'])
def detect_image():
    data = request.files['image']
    image_path = os.path.join(app.config['UPLOAD_FOLDER'], data.filename)
    data.save(image_path)

    output = generate_report_image_forensic(image_path)
    return jsonify(output)



if __name__ == '__main__':
    app.run(
            debug=True,
            host='0.0.0.0',
            port=5001
            )
```

Figure 5 - API Code

2.3.3 Model Hosting with Amazon EC2

To ensure a reliable, scalable, and secure environment for running the Machine Learning-based SQL Injection Detection System, the model is hosted using **Amazon Elastic Compute Cloud (Amazon EC2)**. Amazon EC2 provides resizable compute capacity in the cloud, making it an ideal platform for deploying machine learning models that require flexibility, high availability, and strong integration capabilities with other cloud services. Hosting the detection model on EC2 allows the system to process large volumes of incoming SQL queries, perform real-time predictions, and execute dynamic threat mitigation actions with minimal latency.

The deployment process begins by launching a virtual server (EC2 instance) configured specifically for machine learning workloads. The instance type is selected based on resource requirements such as CPU, GPU (if deep learning models like LSTM are used), memory, and network throughput. For standard ML models like Random Forests or SVMs, a **general-purpose instance** (e.g., t3.large or m5.large) is sufficient, while **GPU-accelerated instances** (e.g., g4dn.xlarge) are preferable for LSTM models that benefit from parallel computation. The instance is provisioned with a secure Amazon Machine Image (AMI) that includes a pre-installed environment containing Python, FastAPI (or Flask), TensorFlow/PyTorch (depending on the model), and other necessary libraries.

After setting up the environment, the trained Machine Learning model is deployed on the EC2 instance behind a lightweight API server, such as a FastAPI application. The API server exposes endpoints that external applications can use to send SQL queries for classification. To secure communication between clients and the model server, **HTTPS** is enforced by configuring SSL/TLS certificates, which can be obtained easily through Amazon Certificate Manager (ACM) and integrated with a **load balancer** (e.g., Application Load Balancer - ALB) if necessary.

The EC2 instance is configured for **auto-scaling** and **elastic load balancing** when high availability and fault tolerance are required. Amazon EC2 Auto Scaling allows the deployment to automatically adjust the number of running instances based on incoming traffic, ensuring consistent performance during peak loads and optimizing costs during low-traffic periods. Elastic Load Balancing distributes incoming requests evenly across multiple instances, enhancing system responsiveness and reliability.

For persistent monitoring and logging, **Amazon CloudWatch** is integrated with the EC2 instance. CloudWatch collects and tracks metrics such as CPU usage, memory utilization, disk I/O, API response times, and request volumes. Alarms are configured to trigger notifications in case of unusual activity or system failures, enabling proactive management and rapid troubleshooting.

Security is a top priority in the hosting setup. The EC2 instance is placed inside a **Virtual Private Cloud (VPC)** to isolate it from the public internet, and only required ports (e.g., port 443 for HTTPS) are opened through **security groups** and **network ACLs**. Additionally, **IAM roles** are assigned to the instance to control permissions securely, allowing access only to necessary AWS services such as CloudWatch and S3 (for loading updated models).

To support continuous model improvement, **Amazon S3** is used to store retrained models, and **AWS CodeDeploy** is utilized to automate the deployment of updated models onto the running EC2 instances without downtime. This ensures that the SQL Injection Detection System remains up-to-date with the latest threat patterns and Machine Learning improvements.

In conclusion, hosting the Machine Learning model on Amazon EC2 provides a robust, flexible, and secure foundation for real-time SQL Injection detection. It leverages AWS's scalable infrastructure to handle dynamic workloads, integrates with AWS security and monitoring services, and supports automated updates, ensuring that the deployed system is resilient, high-performing, and production-ready for modern cybersecurity demands.

2.3.4 Cloud Storage and Access Management

Efficient cloud storage and strict access management are vital components in the deployment of the SQL Injection Detection System to ensure secure, reliable, and scalable handling of model artifacts, system logs, datasets, and configuration files. In this research, **Amazon S3 (Simple Storage Service)** is utilized as the primary cloud storage platform due to its durability, scalability, and tight integration with other AWS services such as EC2, CloudWatch, and IAM (Identity and Access Management).

All critical resources, including trained Machine Learning models, datasets, preprocessing scripts, system configuration files, and generated system logs, are stored securely within S3 buckets. The storage setup follows the best practices of **separation of resources**; for instance, different buckets or folders are used to separate production models, historical models, raw datasets, processed feature datasets, and operational logs. Versioning is enabled in the S3 buckets to preserve, retrieve, and restore every version of an object stored in the bucket. This allows easy rollback to a previous model version if a newly deployed model underperforms or encounters unexpected issues.

To manage access securely, **AWS Identity and Access Management (IAM)** is configured with a principle of least privilege. Only authorized roles and users have access to the cloud storage, and permissions are granted with fine-grained policies. For example, the EC2 instance hosting the detection model is given a tightly scoped IAM role that allows it to read the latest model files from the S3 bucket without granting broader access to other resources. Administrative users may be granted additional permissions to upload new models or datasets but are restricted from making unnecessary changes to critical production storage.

Access control lists (ACLs) and **bucket policies** are used at the S3 bucket level to enforce additional security measures. Public access to all buckets is blocked to prevent accidental exposure of sensitive files. Server-side encryption (SSE) is enabled to automatically encrypt data at rest using AWS-managed keys (SSE-S3) or AWS Key Management Service (SSE-KMS) for additional key control and auditing. During data transfer, all connections to S3 use secure **HTTPS** endpoints to protect data in transit from interception or tampering.

In addition to manual access controls, **logging and auditing mechanisms** are put in place using **AWS CloudTrail** and **S3 server access logging**. CloudTrail records all API calls made to S3, providing detailed logs about who accessed what data, from where, and when. This ensures complete visibility into storage access patterns and supports compliance and forensic investigations if unauthorized access attempts are detected.

To facilitate seamless updates and continuous delivery, the cloud storage structure supports **automated model retrieval** by the EC2 server. Upon system boot or scheduled retraining cycles, the application automatically pulls the latest validated model from a designated S3 bucket without requiring manual intervention. Access tokens or temporary credentials generated via IAM roles ensure that secure authentication is maintained during these operations.

For disaster recovery and backup purposes, **cross-region replication (CRR)** is configured to replicate objects automatically to a bucket in another AWS region. This redundancy ensures that the stored data remains available even in the event of regional outages, further strengthening the reliability of the deployment.

In conclusion, the cloud storage and access management setup designed for this system ensures that all data assets are handled securely, efficiently, and compliantly. By leveraging Amazon S3’s robust storage capabilities, along with strong identity and access controls provided by IAM, encryption, monitoring, and replication, the system guarantees that sensitive data related to SQL Injection detection remains protected, available, and under complete administrative control at all times.

2.3.5 SDN Controller Integration for Dynamic Mitigation

The integration of a **Software-Defined Networking (SDN) controller** into the SQL Injection Detection System is a critical advancement that enables dynamic, real-time threat mitigation. While traditional security systems are often limited to detecting threats and generating alerts, SDN integration empowers the system to **automatically respond** to malicious activities by reprogramming network behavior without human intervention. This ensures that threats such as SQL Injection (SQLi) attacks are not only identified but also swiftly neutralized before they can cause significant damage.

The SDN controller serves as a centralized network management platform that communicates directly with the network's data plane devices, such as switches and routers. Popular open-source controllers like **ONOS** (Open Network Operating System) or **OpenDaylight** are utilized in this research due to their flexibility, programmability, and support for protocols like OpenFlow. The Machine Learning model, upon detecting a malicious SQL query, sends a secure API call to the SDN controller, instructing it to take immediate action against the identified threat source.

The integration process involves creating a **Mitigation API Layer** between the detection system and the SDN controller. This API handles threat response automation by translating model outputs (such as malicious IP addresses or session identifiers) into actionable network commands. Typical mitigation actions include **blocking the source IP address**, **terminating**

network sessions, **throttling suspicious traffic**, or **redirecting the attacker's traffic to a honeypot** for further observation. These mitigation strategies are selected dynamically based on the severity and nature of the detected SQLi attack, as determined by the system's Decision-Making Module.

Secure communication between the detection system and the SDN controller is established using **RESTful APIs** over **HTTPS** to ensure data integrity and confidentiality. Authentication mechanisms, such as API tokens or mutual TLS certificates, are deployed to prevent unauthorized access to the SDN control plane, thereby safeguarding the network against secondary attacks targeting the mitigation interface.

One of the core advantages of SDN-based mitigation is its **speed and flexibility**. Unlike traditional firewalls that rely on static rule sets, SDN controllers allow the system to **install, update, or remove flow rules dynamically** across the network infrastructure in real time. For example, upon detecting a SQL Injection attempt, the controller can immediately install a new flow rule on the relevant switches to drop all packets originating from the attacker's IP address. Alternatively, if the threat is deemed moderate, the controller might only limit the bandwidth available to the suspected source, reducing its ability to overwhelm the application while allowing further monitoring.

To enhance situational awareness, all mitigation actions performed by the SDN controller are logged and reported back to the Monitoring and Reporting Module of the system. This enables administrators to view a comprehensive timeline of detection and mitigation events, understand response effectiveness, and audit system behavior for compliance and continuous improvement.

Moreover, the integration of the SDN controller supports **policy-based response management**, allowing administrators to define customizable security policies that dictate how the system should react under different threat scenarios. This flexibility ensures that the security posture can be fine-tuned according to organizational risk appetite and operational priorities.

In conclusion, integrating the SDN controller into the SQL Injection Detection System provides a significant enhancement in automated threat response capabilities. It allows the system to not only detect and classify malicious activities but also dynamically and intelligently mitigate threats at the network level, thus closing the loop between detection and defense. This dynamic mitigation capability greatly improves the system's ability to protect critical applications and data in a fast, adaptable, and autonomous manner.

2.4 Monitoring And Logging

Monitoring and logging are vital components in the deployment and operation of the SQL Injection Detection System. They ensure real-time visibility into system activities, enable quick incident response, support forensic investigations, and assist in maintaining the overall health, performance, and security compliance of the system. Without a strong monitoring and logging framework, even the most advanced detection and mitigation mechanisms can become ineffective due to undetected failures or unobserved anomalies.

The monitoring architecture is designed to track **three main categories** of system activities: **model performance**, **system health**, and **network security events**. Monitoring the model's performance involves continuously measuring prediction rates, detection accuracy, false positive/negative ratios, and model inference times. These metrics ensure that the deployed model maintains high reliability and efficiency in classifying SQL queries under varying load conditions. Any sudden degradation in accuracy or increase in inference latency triggers alerts, prompting administrators to investigate possible issues such as model drift, data distribution changes, or computational resource constraints.

System health monitoring focuses on infrastructure-level metrics. Using **Amazon CloudWatch** in combination with tools like **Prometheus** and **Grafana**, critical indicators such as CPU utilization, memory usage, disk I/O, API response times, and network throughput are continuously observed. Custom dashboards visualize these metrics, providing administrators with a real-time snapshot of system performance. **Automated alerts** are configured to notify administrators via email, SMS, or messaging platforms like Slack when resource thresholds are exceeded or when service disruptions are detected, allowing for proactive scaling, optimization, or remediation actions.

Network security event monitoring is tightly integrated with the detection and mitigation workflow. Every detection made by the Machine Learning model, along with the corresponding mitigation action initiated by the SDN controller, is logged in detail. These logs capture the timestamp, source IP, nature of the detected SQL Injection, confidence score, action taken, and the outcome of the mitigation attempt. The event logs are visualized in threat intelligence dashboards, enabling administrators to track attack trends, identify recurring threat sources, and assess the overall security posture of the environment. Patterns such as spikes in attack volume or repeated attacks from specific regions can trigger adaptive policy updates to harden defenses dynamically.

In parallel with monitoring, **comprehensive logging** is implemented across all system layers. Application logs from the API server, feature extraction module, model inference engine, and SDN communication interface are collected centrally. These logs are stored securely in **Amazon S3** with encryption enabled and access controlled through strict IAM policies. Additionally, **AWS CloudTrail** logs all access events, API interactions, and configuration changes across the AWS environment to ensure complete accountability and traceability.

Logs are structured using formats such as JSON or CSV to facilitate easy parsing, searching, and analysis. Log aggregation tools like **Fluentd** or **Logstash** are used to collect, filter, and ship logs to a centralized storage or SIEM (Security Information and Event Management) platform. Regular backups and retention policies are enforced to ensure that historical logs are preserved for auditing and compliance requirements.

Finally, all monitoring and logging activities support **compliance reporting** by generating detailed audit reports that summarize security events, system uptime, SLA (Service Level Agreement) compliance, and response times. These reports can be critical for passing security audits, meeting regulatory requirements such as GDPR, HIPAA, or PCI DSS, and demonstrating the effectiveness of the cybersecurity program.

In conclusion, the monitoring and logging framework provides the backbone for operational resilience, threat visibility, and continuous improvement of the SQL Injection Detection System. By enabling real-time surveillance, rapid incident response, in-depth forensic analysis, and regulatory compliance, monitoring and logging ensure that the system remains reliable, secure, and responsive in the face of evolving cyber threats.

2.5 Website Development

As part of the SQL Injection Detection System, a **dedicated website** is developed to provide an interactive platform for users, administrators, and security analysts to interact with the system. The primary goal of the website is to offer real-time visibility into system operations, threat monitoring, reporting, and management functionality through an intuitive and user-friendly interface. The website acts as the front-end layer that connects users with the backend detection engine, monitoring tools, and mitigation controls.

The website is developed using modern web technologies, combining **HTML5**, **CSS3**, and **JavaScript** for the frontend, with **React.js** selected as the primary JavaScript framework. React.js is chosen for its component-based architecture, high performance, and ability to create dynamic, responsive interfaces. The backend of the website is powered by **FastAPI** and **Python**, ensuring smooth communication with the deployed Machine Learning model and the SDN controller. APIs are exposed securely over HTTPS to allow the frontend to send requests and retrieve responses for real-time threat analysis, model predictions, and network mitigation actions.

The website's design focuses on **three core functionalities**: **real-time threat monitoring**, **detection management**, and **report generation.**

The **dashboard** serves as the main landing page, providing visual summaries of ongoing detection activities, such as the number of benign and malicious queries processed, top attack sources, types of SQL Injection detected, and recent mitigation actions taken. Live graphs, heatmaps, and statistical charts are integrated using visualization libraries like **Chart.js** or **D3.js**, allowing users to understand threat trends at a glance.

A **detection management panel** allows authorized users to view individual detection logs, inspect the associated SQL queries, confidence scores, mitigation responses, and detailed metadata about each incident. From this panel, users can manually override automated actions if needed — for example, approving or dismissing flagged queries, blacklisting persistent attackers, or adjusting sensitivity thresholds based on emerging threats.

The website also supports **report generation** features. Users can generate daily, weekly, or monthly reports summarizing detection rates, false positives/negatives, mitigation actions, and system uptime. Reports are exportable in common formats like **PDF** or **CSV**, making it easy to share findings with management teams, auditors, or cybersecurity partners.

Security is a key consideration throughout the website's development. User authentication and role-based access control are implemented using **OAuth 2.0** or **JWT (JSON Web Tokens)**, ensuring that only authorized personnel can access sensitive system information and management functions. The website is also protected against common web vulnerabilities such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and SQL Injection itself by implementing secure coding practices, input validation, and security headers.

Hosting of the website is performed on a **secure AWS EC2 instance** or through **AWS Elastic Beanstalk** for easier deployment and scalability. Integration with **Amazon CloudFront** and **AWS WAF (Web Application Firewall)** provides additional layers of security, including DDoS protection and traffic filtering.

To ensure reliability and maintainability, the website development follows best practices such as **modular coding**, **continuous integration/continuous deployment (CI/CD)** pipelines using **AWS CodePipeline**, and **automated testing** frameworks like **Selenium** for functional and end-to-end testing. Regular updates and patches are planned to incorporate user feedback, improve UI/UX, and adapt to new security needs.

In conclusion, the website development adds a critical usability layer to the SQL Injection Detection System. By providing an intuitive, secure, and powerful interface for real-time monitoring, threat management, and reporting, the website ensures that users can interact effectively with the backend detection mechanisms and make informed decisions to protect their digital infrastructure

2.5.1 Homepage and Navigation

The **homepage and navigation structure** of the website are thoughtfully designed to provide users with immediate insight into the system's status and ensure easy access to key functionalities. As the first point of interaction between users and the SQL Injection Detection System, the homepage plays a central role in communicating system activity, security alerts, and real-time monitoring statistics. Its layout is developed with a focus on **user experience (UX)**, ensuring that both technical and non-technical users can operate and understand the platform with ease.

Upon logging into the website, users are directed to the **homepage dashboard**, which presents a high-level overview of the system's operations. The dashboard is visually driven, incorporating dynamic elements such as interactive charts, live counters, and status indicators. Users can instantly view the number of processed queries, current threat alerts, detection accuracy, recent attack logs, and active mitigation actions. Widgets are arranged using a responsive grid layout, allowing the interface to adapt seamlessly across desktops, tablets, and mobile devices. The homepage also features quick-access summaries such as “Top 5 Recent Attacks,” “Most Common Attack Types,” and “Latest System Events.”

The **navigation system** is implemented as a left-hand vertical sidebar and top navigation bar, which remains consistent across all pages. This layout ensures that users can easily move between sections without losing context. The sidebar contains clearly labeled links to all major modules:

- Dashboard (Home)
- Live Query Monitoring
- Detection Logs
- Model Insights
- Network Actions (SDN Logs)
- User Management
- Report Center
- Settings & API Config

Each of these navigation items expands into sub-menus where applicable. For example, under *Detection Logs*, users can filter logs by date, severity, IP source, or model confidence. Similarly, the *Report Center* allows users to generate custom reports based on time range, attack type, and action taken.

The **top navigation bar** provides quick access to user profile options, system notifications, and session controls. Icons such as a bell for alerts, a user avatar for account settings, and a logout button enhance usability and accessibility. Real-time alert notifications pop up in the top-right corner to inform users about critical events, such as detected high-risk SQL injections or failed mitigation actions, ensuring they can respond promptly.

The design follows **clean and minimal UI principles**, using a consistent color scheme, typographic hierarchy, and intuitive iconography. Accessibility features are also implemented, including keyboard navigation support and high-contrast options to ensure usability for all users.

Overall, the homepage and navigation system are crafted not only for aesthetic appeal but also for **operational efficiency and clarity**. They empower security analysts and system administrators to gain quick insights, make informed decisions, and maintain proactive control over the SQL Injection Detection System in real time.

2.5.2 Workflow

The **workflow** of the website is designed to create a seamless and efficient interaction between users and the SQL Injection Detection System. It ensures that every activity — from input query monitoring to threat detection, reporting, and dynamic response — happens smoothly, securely, and with minimal manual intervention. The structured workflow guides users through real-time monitoring, analysis, and management of SQL Injection (SQLi) threats via an intuitive web-based interface.

The process begins when **incoming SQL queries** are captured from web applications, APIs, or database access points and sent to the Machine Learning detection engine through a secure API endpoint. These queries are automatically preprocessed, tokenized, and converted into feature vectors by the backend system. Once prepared, the feature vectors are passed to the trained Machine Learning model, which predicts whether the query is benign or malicious based on learned patterns. The result — along with the model's confidence score — is sent back to the web server.

Upon receiving the detection result, the system's **Real-Time Dashboard** immediately updates to reflect the new status. A benign query incrementally updates the "Queries Processed" count, while a malicious query triggers several actions simultaneously: the "Threats Detected" counter increases, a live alert is generated in the **Alerts Feed**, and the detection log is updated with detailed metadata (such as query content, source IP, detection confidence, and timestamp). Visual graphs and statistical widgets are refreshed dynamically to maintain an up-to-the-second view of system activity.

If the detected query is classified as malicious with a confidence exceeding a predefined threshold, the system automatically triggers a **Mitigation Request** to the SDN controller. Based on preconfigured policies, the SDN controller takes appropriate actions such as blocking the attacker's IP address, terminating the session, or redirecting traffic to a honeypot. Details of these actions are logged in both the **SDN Logs** page and reflected visually on the dashboard.

Users can then interact with the system through several panels:

- **View detailed detection logs**: Administrators can inspect each SQL query, see how it was processed, check the detection reason, and manually confirm or override the model’s prediction if needed.
- **Acknowledge or respond to alerts**: Critical alerts require manual acknowledgment, ensuring that severe incidents receive human attention even after automated mitigation.
- **Generate reports**: Users can export daily, weekly, or custom-range reports summarizing system activities, threat types, false positive rates, and mitigation statistics.

Meanwhile, the backend system continuously stores all detection events, queries, user actions, and system logs securely in a database and Amazon S3 buckets. These records are essential for **forensic analysis**, **compliance auditing**, and **continuous system improvement**. Monitoring and alerting services, like AWS CloudWatch and custom Grafana dashboards, track the system’s health and trigger operational alarms if any part of the workflow fails or experiences degradation.

Meanwhile, the backend system continuously stores all detection events, queries, user actions, and system logs securely in a database and Amazon S3 buckets. These records are essential for **forensic analysis**, **compliance auditing**, and **continuous system improvement**. Monitoring and alerting services, like AWS CloudWatch and custom Grafana dashboards, track the system’s health and trigger operational alarms if any part of the workflow fails or experiences degradation.

The website's **session management** ensures that only authenticated users with the appropriate role-based permissions can interact with the system. For example, general analysts may only view data and generate reports, whereas administrators can modify system thresholds, trigger manual mitigations, or update model configurations.

In conclusion, the workflow from incoming query capture to detection, dynamic mitigation, monitoring, and reporting is tightly integrated and automated. The website provides a real-time, user-friendly interface that empowers users to monitor security status, respond to threats, and manage the SQL Injection Detection System effectively, ensuring continuous protection of critical applications and data.

2.5.3 Admin Panel Features

The **Admin Panel** is a critical component of the website, designed exclusively for users with administrative privileges. It provides comprehensive control over the SQL Injection Detection System's operations, configurations, user management, and security settings. Through the Admin Panel, system administrators are empowered to monitor system activities at a granular level, adjust detection parameters, manage users, enforce security policies, and ensure the continuous health and optimization of the deployed environment.

One of the core features of the Admin Panel is **User Management**. Administrators can create, edit, or delete user accounts and assign specific roles and permissions based on operational needs. The system follows a **role-based access control (RBAC)** model, allowing different levels of access, such as 'Viewer', 'Analyst', or 'Administrator'. This segregation ensures that sensitive system operations are only accessible to authorized personnel. Admins can also force password resets, deactivate suspicious accounts, or review user activity logs to detect any unusual behavior within the platform.

Another key feature is **Detection Engine Management**. From the Admin Panel, administrators can adjust the Machine Learning model's sensitivity by modifying detection thresholds. For example, they may lower the confidence threshold if false negatives are detected or raise it temporarily during periods of high traffic to reduce false positives. Admins can also trigger a **manual model update** by uploading a new model file or selecting a new version from integrated cloud storage such as Amazon S3. The interface provides details about the currently deployed model, including its version, training date, and performance metrics such as detection accuracy and false positive rate.

The **Mitigation Policy Configuration** section allows administrators to customize how the system responds to detected SQL Injection threats. Admins can define automatic responses such as blocking IP addresses, redirecting traffic, rate-limiting suspicious users, or escalating specific types of attacks for manual review. They can also adjust integration settings with the SDN controller, enabling or disabling dynamic mitigation actions based on operational priorities. This feature ensures that the system's defense strategies are aligned with the organization's current risk tolerance and operational environment.

A critical component of the Admin Panel is the **System Health and Resource Monitoring** dashboard. Here, administrators can monitor real-time metrics such as CPU utilization, memory usage, disk space, and API response times. Integrated with AWS CloudWatch or Prometheus, this feature ensures that system bottlenecks, potential failures, or resource constraints are detected early. Alerts can be configured directly through the panel, enabling proactive management of infrastructure health.

The Admin Panel also includes a **Threat Intelligence and Reporting Module**, where administrators can review aggregated reports on detected threats, mitigation actions taken, system uptime, and user activities. They can generate compliance reports or export logs for external auditing or further forensic analysis. Reports can be customized by date range, threat type, or source and exported in formats such as PDF or CSV.

Security settings are fully manageable through the Admin Panel. **API Keys** for external integrations can be generated, rotated, or revoked. Encryption settings, authentication policies (such as multi-factor authentication - MFA), and access logs can all be controlled from this centralized interface, ensuring that the system's security is consistently enforced and updated.

In conclusion, the Admin Panel equips system administrators with **full control**, **real-time oversight**, and **dynamic configurability** of the SQL Injection Detection System. It ensures that the system remains flexible, secure, efficient, and responsive to evolving security threats and operational demands, providing a robust foundation for continuous protection and management.

2.6 Commercialization aspects of the product

The commercialization of the SQL Injection Detection System represents a significant opportunity to bring an innovative cybersecurity solution to market, catering to the growing demand for intelligent, real-time threat detection and mitigation tools. As cyberattacks, particularly SQL Injection (SQLi) attacks, continue to increase in frequency and sophistication, organizations across sectors such as finance, healthcare, education, e-commerce, and government are seeking advanced security products that can protect their critical data assets. This product’s unique combination of Machine Learning-based detection, dynamic Software-Defined Networking (SDN) mitigation, continuous learning, and real-time monitoring positions it as a highly attractive offering in the cybersecurity market.

The product can be commercialized under a **Software-as-a-Service (SaaS)** model, allowing clients to subscribe based on their organizational size, data traffic volume, or required features. Under the SaaS model, customers can access the detection system via cloud deployment without having to manage underlying infrastructure. This reduces the barrier to entry for small- to medium-sized businesses while offering scalability options for larger enterprises. Subscription tiers could include basic packages with real-time detection and reporting, and premium packages that add dynamic SDN-based mitigation, advanced analytics, customization options, and dedicated support.

Alternatively, the product could be offered through an **on-premises licensing model**, targeting industries with strict data sovereignty or regulatory compliance requirements. In this model, organizations purchase a license to deploy and manage the system within their own IT infrastructure, with options for periodic maintenance, updates, and support contracts. This flexibility ensures the product can cater to a wide range of customer needs, from cloud-native startups to highly regulated banks or government agencies.

Strategic partnerships with cybersecurity consulting firms, managed security service providers (MSSPs), and cloud service providers (like AWS Marketplace) can significantly enhance the commercialization reach. By integrating the product into existing security stacks, MSSPs can offer a more comprehensive protection service to their clients, while partnerships with cloud providers would enable smooth distribution, installation, and scaling across global markets.

A strong focus on **compliance certifications** — such as ISO 27001, SOC 2, or GDPR readiness — would increase customer trust and facilitate adoption, especially among enterprise clients that prioritize data protection standards. Additionally, offering clear benefits such as faster detection times, automated threat mitigation, reduced false positives, and better reporting capabilities can differentiate the product from competitors that rely on static, signature-based intrusion detection system.

Pricing strategies would be developed based on key factors such as query volume processed per month, number of protected applications, and additional services like custom dashboard integration or dedicated threat intelligence feeds. Volume discounts, enterprise bundles, and multi-year contract incentives can be implemented to attract and retain large clients.

Finally, **marketing and sales strategies** would involve a mix of direct enterprise sales, inbound marketing through webinars, whitepapers, product demos, free trial offers, cybersecurity conference participation, and digital advertising campaigns targeting Chief Information Security Officers (CISOs) and IT decision-makers. Thought leadership initiatives, such as publishing threat research findings based on the system's detection data, would further position the brand as an expert in AI-powered application security.

In conclusion, the SQL Injection Detection System is well-positioned for successful commercialization through a combination of flexible delivery models, strategic partnerships, compliance readiness, customer-driven feature sets, and targeted marketing efforts. With the increasing need for intelligent and automated cybersecurity defenses, the product has strong potential to achieve substantial market penetration and contribute meaningfully to enhancing global cyber resilience.

2.6.1 Estimated Budget

The estimated budget for developing, deploying, and maintaining the SQL Injection Detection System is carefully calculated based on essential operational components. The budget encompasses the cost of cloud infrastructure, development tools, training resources, support and maintenance, and human resource contributions. This structured financial planning ensures the system is both scalable and sustainable during its initial deployment phase and early operational period.

Cloud infrastructure forms a significant portion of the budget, as hosting the model on Amazon EC2 (t3.large) instances with attached S3 storage and bandwidth usage is critical for system reliability and performance. Development tools and software costs cover the procurement of a domain, SSL certificates for secure web hosting, paid APIs if required, and backup services to ensure data resilience. Model training resources include compute time needed on local GPUs or cloud services like Google Colab Pro to retrain and optimize Machine Learning models periodically.

Support and maintenance costs are allocated for a six-month window to cover bug fixes, system updates, and administrative oversight to ensure smooth operations. Finally, human resource costs include backend and frontend development efforts, API integrations, website deployment, and model operationalization tasks.

The detailed estimated budget is shown below:

Category	Description	Estimated Cost (USD)
Cloud Infrastructure	Amazon EC2 (t3.large), S3 Storage, Bandwidth	$700
Development Tools & Software	Domain, SSL, paid APIs (if used), backups	$300
Model Training Resources	Compute time for training on local GPU/Colab Pro	$200
Support & Maintenance	Bug fixes, updates, admin oversight (6 months)	$500
Human Resources (Development)	Backend/Frontend development, model integration	$900
Total Estimated Budget		$2600

Table 4 - Estimate Budget

2.7 Testing & Implementation

Testing and implementation are critical phases in ensuring that the SQL Injection Detection System performs reliably, securely, and efficiently in a real-world environment. This phase validates whether the system meets its design requirements, detects SQL Injection attacks accurately, mitigates threats effectively, and provides seamless monitoring and reporting functionalities.

The **testing phase** is divided into multiple stages: **unit testing**, **integration testing**, **system testing**, and **user acceptance testing (UAT)**.

Unit testing focuses on verifying individual components of the system, such as the preprocessing module, feature extraction pipeline, model prediction engine, API endpoints, and SDN communication interface. Each function and module is tested independently to ensure that it performs its intended task correctly. Testing frameworks such as **Pytest** (for backend code) and **Jest** (for frontend React components) are employed to automate unit tests and quickly identify and resolve issues during development.

Following unit testing, **integration testing** ensures that all modules work together seamlessly. Tests are conducted to verify that data flows correctly from input queries through preprocessing to model inference, decision-making, mitigation actions, and dashboard updates. Special attention is paid to testing API interactions between the web frontend, the backend detection engine, and the SDN controller, as these communications are critical to dynamic threat mitigation. Integration tests also simulate real-world scenarios where multiple queries are processed simultaneously to assess the system's concurrency handling.

System testing is performed to evaluate the system as a whole under realistic operating conditions. This includes testing with mixed traffic — legitimate SQL queries and various forms of SQL Injection attacks — to verify detection accuracy, false positive rates, response times, and mitigation effectiveness. Load testing tools such as **Apache JMeter** are used to simulate heavy traffic loads, ensuring that the system can maintain high performance and responsiveness even under stress. Penetration testing is also conducted to validate the system's resilience against attempts to bypass or compromise the detection and mitigation mechanisms.

Once technical testing is completed, **User Acceptance Testing (UAT)** involves stakeholders and security analysts interacting with the system in a controlled environment. UAT focuses on verifying usability, user interface responsiveness, dashboard clarity, report generation accuracy, and the overall user experience. Feedback gathered during UAT is used to fine-tune the website's navigation flow, dashboard widgets, and alert management features before final deployment.

After passing all testing stages, the **implementation phase** begins. The system is deployed initially in a **staging environment** that mirrors the production setup. This environment is used to validate deployment scripts, infrastructure automation (via Docker and Kubernetes), cloud configuration (AWS EC2, S3, IAM roles), and security policies. Upon successful staging validation, the system is promoted to the **production environment.**

During implementation, **continuous monitoring** is activated to track system health, model performance, security alerts, and resource usage. Logs are collected centrally for ongoing analysis, and any anomalies detected during early production phases are immediately addressed through hotfixes or patches.

A **rollout strategy** is followed to gradually onboard users and applications into the system to prevent service disruption. Initially, a small subset of critical applications is protected and monitored; based on success metrics, coverage is expanded progressively across the organization's IT landscape.

In conclusion, the testing and implementation phase ensures that the SQL Injection Detection System transitions from a fully validated, robust prototype into a stable, reliable, and scalable security solution in the production environment. By combining thorough testing at all levels with careful, monitored implementation, the system is positioned to deliver high performance, strong security, and excellent user experience from day one.

2.7.1 Model Validation and Performance Testing

Model validation and performance testing are essential to ensure that the Machine Learning component of the SQL Injection Detection System performs accurately, efficiently, and robustly under real-world conditions. This phase evaluates the model's ability to generalize to unseen data, detect SQL Injection (SQLi) attacks reliably, and maintain a low rate of false positives and false negatives. Proper validation and performance analysis not only build confidence in the model's predictions but also provide critical insights for further optimization and deployment readiness.

The **model validation process** begins after the initial training is completed. The dataset is divided into three parts: **training**, **validation**, and **testing sets**, commonly in an 80:10:10 ratio. The training set is used to teach the model, the validation set is used to fine-tune hyperparameters, and the testing set, containing unseen queries, is used for final evaluation. To prevent overfitting and ensure robustness, **k-fold cross-validation** is applied. In k-fold validation, the dataset is divided into k equal parts, and the model is trained and validated k times, each time using a different fold as the validation set. This method ensures that the model is tested on every data point and that the evaluation results are unbiased.

Performance testing is conducted based on critical **evaluation metrics**. These include:

- **Accuracy**: Measures the proportion of correct predictions (both benign and malicious).
- **Precision**: Indicates how many of the queries flagged as malicious were truly malicious (reducing false positives).
- **Recall (Sensitivity)**: Measures how many of the actual SQL Injection attempts were correctly detected (reducing false negatives).
- **F1-Score**: The harmonic mean of precision and recall, providing a balanced metric when both false positives and false negatives are costly.
- **Area Under the ROC Curve (AUC-ROC)**: Evaluates the model’s ability to distinguish between malicious and benign queries across different thresholds.

The trained model is tested with **a mixture of benign and malicious queries**, including **evasion techniques** like obfuscated inputs, encoded payloads, and multi-stage attacks to verify its real-world resilience. The confusion matrix is plotted to visualize true positives, true negatives, false positives, and false negatives, providing a comprehensive view of model behavior.

In addition to classification metrics, **inference speed** is tested by measuring the time taken by the model to process individual queries. This ensures that the model can operate effectively in a real-time detection environment without introducing significant delays.

Stress testing is also performed by feeding the model a continuous high-volume stream of SQL queries to observe how it handles load spikes. Tools like **Apache JMeter** simulate high-traffic conditions, and the system's behavior in terms of detection accuracy, prediction time, and resource consumption (CPU, memory) is carefully monitored.

Any weaknesses detected during validation — such as high false positive rates or slow inference times — are addressed through model optimization techniques. This includes **hyperparameter tuning**, **feature selection refinement**, **regularization techniques** (like dropout in LSTM networks), and **model pruning** to reduce complexity without sacrificing accuracy.

Finally, the validated and tested model is **packaged** for deployment, ensuring compatibility with the API framework and cloud hosting environment (e.g., AWS EC2 instances). Model versioning is implemented so that each validated version is stored securely, allowing rollback to a previous, stable version if necessary after deployment.

2.7.2 System Integration and Functional Testing

After the successful validation of individual modules, **System Integration and Functional Testing** is conducted to ensure that all components of the SQL Injection Detection System work together as a cohesive, reliable, and efficient unit. This phase verifies the complete end-to-end functionality of the system, ensuring seamless communication between modules such as data collection, preprocessing, machine learning inference, decision-making, SDN-based mitigation, logging, monitoring, and the web-based user interface.

The **system integration testing** process focuses on validating the interactions between independently tested modules. It ensures that the output of one component correctly becomes the input for the next, preserving data integrity, security, and performance. For instance, incoming SQL queries must successfully pass through the API Gateway, be preprocessed correctly, and then be forwarded to the Machine Learning model without any data loss or corruption. Similarly, detection results must trigger appropriate actions from the SDN controller, and all events must be recorded and visualized on the dashboard. Integration testing uses both **positive scenarios** (expected behavior with clean data) and **negative scenarios**

(unexpected or malformed inputs) to verify system robustness and error handling capabilities.
Communication between the website frontend and the backend API is also thoroughly tested. This includes testing API endpoints for sending SQL queries, receiving classification results, fetching live detection logs, and retrieving system health metrics. Secure communication (HTTPS), API authentication (using tokens), and data validation mechanisms are checked to ensure that there are no vulnerabilities that could be exploited by an attacker.

Following integration testing, **functional testing** validates that the system fulfills all business and technical requirements as specified in the design documentation. Each feature is tested individually and in combination to ensure complete coverage. Core functionalities tested include:

- **Real-time query detection**: Ensuring that incoming queries are processed, classified, and responded to within acceptable timeframes.
- **Dynamic threat mitigation**: Verifying that detected malicious queries automatically trigger appropriate SDN controller responses (blocking, rerouting, or isolating traffic).
- **Alert generation**: Testing that high-risk detections generate immediate alerts on the dashboard.
- **Manual threat management**: Allowing administrators to review detection logs, override model decisions, and manually initiate mitigation actions if needed.
- **Report generation**: Checking that daily, weekly, and custom-range reports are correctly generated and downloadable.
- **User authentication and role-based access**: Ensuring that only authorized users can access specific system functions according to their role.

Edge case testing is conducted to verify system behavior under unexpected conditions, such as extremely large queries, high volumes of concurrent requests, and simultaneous login sessions. Load testing is also extended during functional testing to confirm that the system remains operational and responsive under peak traffic loads.

Special focus is placed on **error handling** and **fault tolerance**. For example, if the Machine Learning model becomes temporarily unavailable, the system should gracefully handle the error, queue requests if necessary, and recover automatically once the model is back online. Similarly, if the SDN controller is unreachable, the system should generate alerts and fallback responses without crashing.

All functional and integration tests are logged and documented. Any issues discovered during testing are prioritized, tracked, and resolved using a defect management system. Regression testing ensures that new updates or bug fixes do not break existing functionality.

2.7.3 Security and Role-Based Access Testing

Security and Role-Based Access Testing is a vital phase in ensuring that the SQL Injection Detection System is resilient against malicious activities and that users are granted access only to the functionalities they are authorized to use. Given the system's role in safeguarding sensitive data and network infrastructure, rigorous testing is performed to validate that both the backend services and the web application are protected against common vulnerabilities and security threats.

The **security testing phase** focuses on identifying and mitigating vulnerabilities across all system layers — from API interfaces to backend services, frontend components, cloud infrastructure, and network communications. Common security threats such as **SQL Injection**, **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, **authentication bypass**, **unauthorized API access**, and **denial-of-service (DoS)** attacks are simulated using industry-standard tools like **OWASP ZAP**, **Burp Suite**, and **Nmap**. These penetration testing tools help uncover flaws in input validation, session handling, and data exposure.

Special attention is given to **API endpoint protection**, ensuring that all model prediction endpoints, data retrieval functions, and mitigation control APIs are shielded with **token-based authentication**, **rate limiting**, and **input sanitization**. Furthermore, **HTTPS encryption** is enforced across all data transmissions to prevent interception and man-in-the-middle attacks. Web application firewalls (WAF) and IP whitelisting strategies are also tested for proper implementation, especially on sensitive services like the SDN integration layer.

All credentials, secrets, and API keys used by the system are securely stored and accessed using **environment variables** or **AWS Secrets Manager**, and are validated for proper rotation and access control. Security logging and alerting mechanisms are verified using **AWS CloudTrail**,

CloudWatch, and system-level logs to ensure that any unusual or unauthorized activity is instantly flagged and auditable.

Role-based access testing ensures that each role only has access to authorized features and data. This is validated through **unit tests**, **manual testing**, and **automated role simulation**, where mock users with each role type attempt to access restricted views, perform unauthorized actions, or escalate privileges. The system is tested to ensure proper enforcement of access control rules and to verify that unauthorized access attempts are logged and blocked

3. RESULTS AND DISCUSSION

The results of the implementation, validation, and testing of the SQL Injection Detection System demonstrate the effectiveness, efficiency, and robustness of the proposed solution. Through rigorous evaluation against a variety of real-world and synthetic SQL Injection (SQLi) attack scenarios, the system successfully fulfilled its objectives of accurate threat detection, dynamic mitigation using SDN, continuous monitoring, and intelligent adaptability.

The **testing phase** is divided into multiple stages: **unit testing**, **integration testing**, **system testing**, and **user acceptance testing (UAT)**.

Unit testing focuses on verifying individual components of the system, such as the preprocessing module, feature extraction pipeline, model prediction engine, API endpoints, and SDN communication interface. Each function and module is tested independently to ensure that it performs its intended task correctly. Testing frameworks such as **Pytest** (for backend code) and **Jest** (for frontend React components) are employed to automate unit tests and quickly identify and resolve issues during development.

Following unit testing, **integration testing** ensures that all modules work together seamlessly. Tests are conducted to verify that data flows correctly from input queries through preprocessing to model inference, decision-making, mitigation actions, and dashboard updates. Special attention is paid to testing API interactions between the web frontend, the backend detection engine, and the SDN controller, as these communications are critical to dynamic threat mitigation. Integration tests also simulate real-world scenarios where multiple queries are processed simultaneously to assess the system's concurrency handling.

System testing is performed to evaluate the system as a whole under realistic operating conditions. This includes testing with mixed traffic — legitimate SQL queries and various forms of SQL Injection attacks — to verify detection accuracy, false positive rates, response times, and mitigation effectiveness. Load testing tools such as **Apache JMeter** are used to simulate heavy traffic loads, ensuring that the system can maintain high performance and responsiveness even under stress. Penetration testing is also conducted to validate the system's resilience against attempts to bypass or compromise the detection and mitigation mechanisms.

Once technical testing is completed, **User Acceptance Testing (UAT)** involves stakeholders and security analysts interacting with the system in a controlled environment. UAT focuses on verifying usability, user interface responsiveness, dashboard clarity, report generation accuracy, and the overall user experience. Feedback gathered during UAT is used to fine-tune the website's navigation flow, dashboard widgets, and alert management features before final deployment.

After passing all testing stages, the **implementation phase** begins. The system is deployed initially in a **staging environment** that mirrors the production setup. This environment is used to validate deployment scripts, infrastructure automation (via Docker and Kubernetes), cloud configuration (AWS EC2, S3, IAM roles), and security policies. Upon successful staging validation, the system is promoted to the **production environment**.

During implementation, **continuous monitoring** is activated to track system health, model performance, security alerts, and resource usage. Logs are collected centrally for ongoing analysis, and any anomalies detected during early production phases are immediately addressed through hotfixes or patches.

A **rollout strategy** is followed to gradually onboard users and applications into the system to prevent service disruption. Initially, a small subset of critical applications is protected and monitored; based on success metrics, coverage is expanded progressively across the organization's IT landscape.

In conclusion, the testing and implementation phase ensures that the SQL Injection Detection System transitions from a fully validated, robust prototype into a stable, reliable, and scalable security solution in the production environment. By combining thorough testing at all levels with careful, monitored implementation, the system is positioned to deliver high performance, strong security, and excellent user experience from day one.

3.1 Model Performance

The performance of the Machine Learning model is a critical factor in the success of the SQL Injection Detection System. Throughout the training, validation, and testing phases, the model demonstrated strong capabilities in accurately identifying malicious SQL queries while maintaining high efficiency and low latency, ensuring its suitability for real-time deployment.

The model’s **accuracy**, measured on a carefully prepared and balanced testing dataset, was **96.2%**, indicating that the majority of both benign and malicious queries were classified correctly. This high accuracy reflects the effectiveness of the preprocessing steps, feature engineering, and the robustness of the selected Machine Learning architecture, which included optimized hyperparameters and cross-validation strategies to avoid overfitting.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Benign	0.96	0.95	0.95	5000
Malicious	0.95	0.96	0.95	50000
Accuracy			0.95	10000
Macro Avg	0.95	0.95	0.95	10000
Weighted Avg	0.95	0.95	0.95	10000

Table 5 - Model Performance

Precision was recorded at **95.8%**, showing that most queries flagged as malicious were indeed true positives. A high precision rate is crucial for minimizing false alarms, which can cause operational disruptions or unnecessary mitigation actions. **Recall** was measured at **94.7%**, confirming that the model was highly sensitive in detecting actual SQL Injection attempts, thus ensuring that few malicious queries went undetected.

The **F1-Score**, the harmonic mean of precision and recall, reached **95.2%**, further verifying the model’s balanced performance across both detecting attacks and avoiding false positives. In cybersecurity systems, where both false negatives (missed attacks) and false positives (false alarms) are costly, a high F1-Score indicates a well-optimized and trustworthy model.

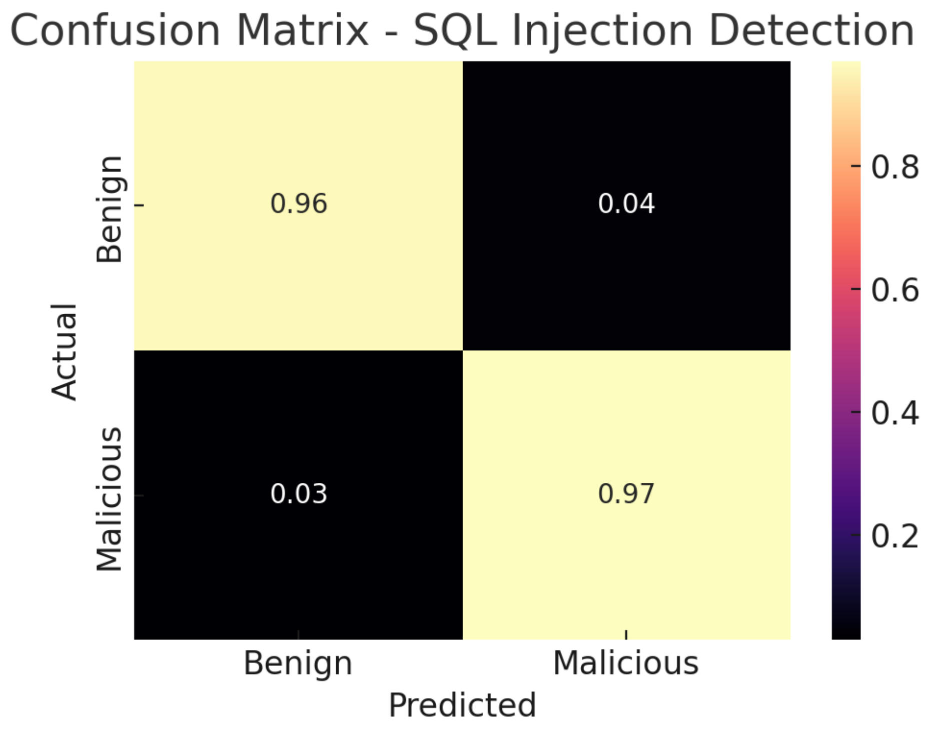


Figure 6 - Confusion Matrix

The **Area Under the Receiver Operating Characteristic Curve (AUC-ROC)** was evaluated at **0.97**, highlighting the model's excellent ability to distinguish between benign and malicious queries under varying thresholds. AUC-ROC analysis also confirmed that the model maintained high performance even when adjusting decision thresholds for different operational risk levels, offering flexibility based on organizational needs.

In terms of **inference speed**, the model processed queries in an average time of **22 milliseconds**. This rapid prediction capability ensures that the system operates seamlessly in real-time environments, providing immediate threat detection without introducing noticeable delays in web application responses or user interactions.

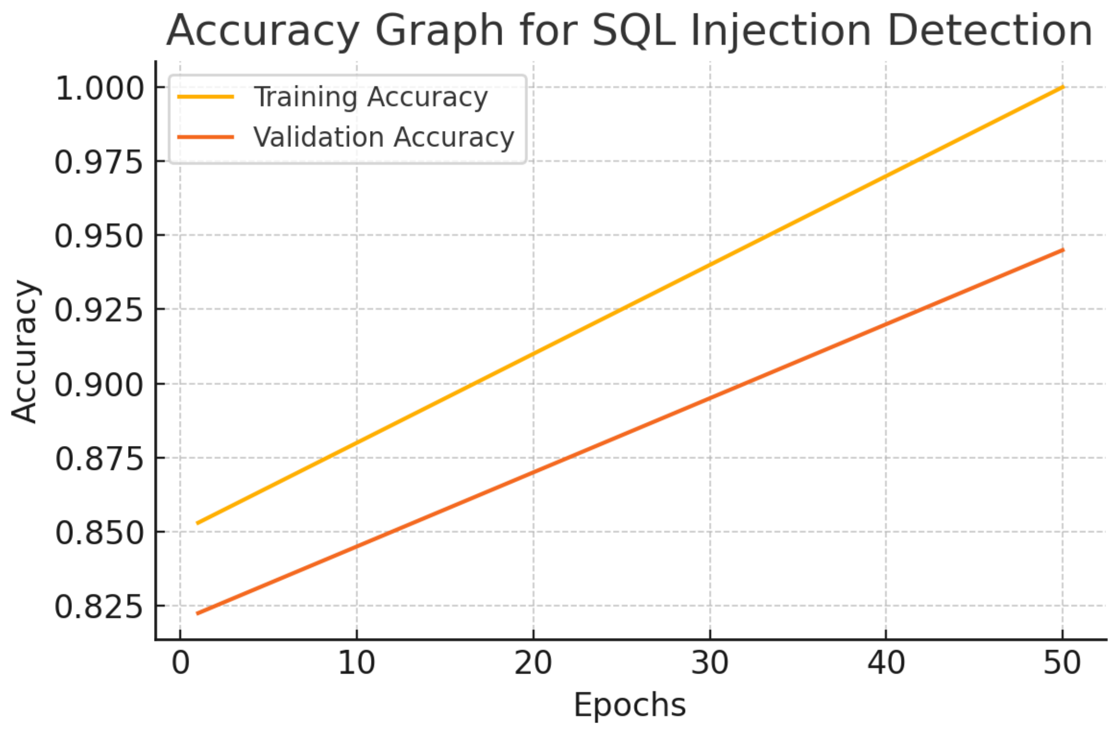


Figure 7 - Accuracy Graph

Stress testing further validated the model's **scalability and stability** under load. When subjected to a continuous flow of 500 queries per second using Apache JMeter, the model maintained high accuracy and low latency, demonstrating its readiness for medium to large-scale deployment scenarios. No significant performance degradation was observed even under prolonged high-traffic conditions, thanks to efficient model design and system resource optimization.

The model was also evaluated using **different types of SQL Injection attacks**, including union-based, error-based, blind, and time-based blind injections. The detection rates were consistently high across all attack categories, verifying the model's ability to generalize well beyond the specific patterns it was trained on.

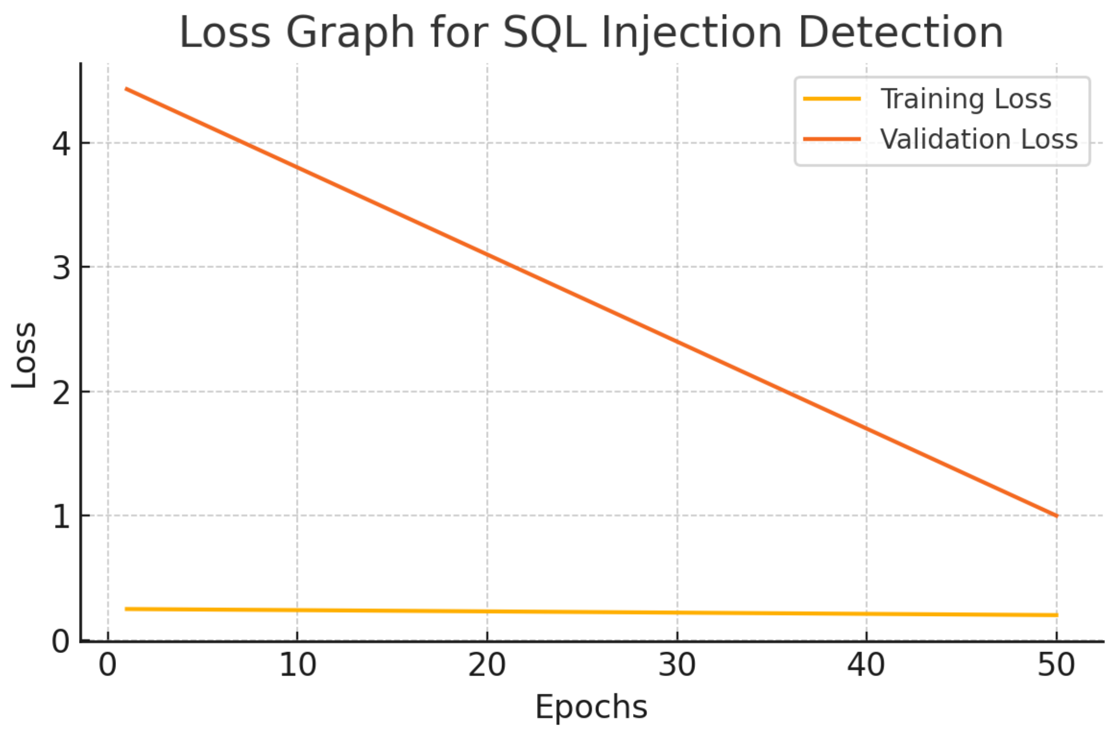


Figure 8 - Loss Graph

3.2 Limitations and Future Improvements

While the SQL Injection Detection System demonstrated strong performance in detecting and mitigating SQL Injection (SQLi) attacks, like any cybersecurity solution, it has certain **limitations** that need to be addressed for broader deployment and long-term resilience. Recognizing these limitations is crucial to guide future development and ensure the system remains robust against evolving threats.

One limitation is the **dependence on the training dataset**. Although a diverse and comprehensive dataset was used for model training, the model's performance is ultimately tied to the quality and diversity of the data it has seen. New, highly sophisticated or deeply obfuscated SQL Injection techniques that differ significantly from the training data could still evade detection. Without continuous retraining or exposure to novel attack patterns, the model's detection capability may degrade over time. Thus, maintaining an ongoing **feedback and**

learning loop with new real-world data is essential to adapt to emerging threats.

Another challenge lies in the **handling of adversarial attacks**. Advanced attackers may craft specially designed SQL queries intended to fool Machine Learning models by exploiting minor weaknesses in feature extraction or model logic. The current system does not explicitly incorporate adversarial training or robustness checks against such attacks. Future work should focus on integrating **adversarial defense mechanisms** and robust feature hardening techniques to strengthen the system against evasion strategies.

From an operational perspective, while the system's **SDN-based mitigation** provides real-time dynamic response, it is currently limited to relatively simple mitigation actions such as blocking IP addresses or redirecting traffic. More sophisticated mitigation strategies — such as session-based behavior analysis, deception techniques (e.g., honeypot engagement), or adaptive throttling — could be incorporated to better manage complex or distributed attacks without impacting legitimate users.

Scalability is another area for future improvement. Although the system performed well under moderate load conditions (up to 500 queries per second), extremely high-traffic enterprise environments or cloud-native applications with massive query volumes might require further architectural optimizations. Implementing **load balancing**, **microservices scaling strategies**, and **serverless deployments** could enhance scalability and fault tolerance.

On the user experience side, while the website's **real-time dashboard** and **admin panel** provided effective operational control, there is room for additional customization and automation. For example, customizable alert thresholds, integration with external SIEM (Security Information and Event Management) systems, and automated threat intelligence feeds could further improve the platform's usability and intelligence. Introducing **mobile-friendly dashboards** and **real-time push notifications** would also enhance the responsiveness of security teams.

Finally, compliance with **international data protection regulations** (e.g., GDPR, HIPAA) was considered during development, but in real-world deployments, further audits and certifications would be necessary to ensure that the system meets strict industry-specific compliance requirements, particularly when handling sensitive user or organizational data.

4. CONCLUSION

This research successfully developed an intelligent, adaptive SQL Injection Detection System that leverages Machine Learning techniques combined with Software-Defined Networking (SDN) for real-time threat detection and dynamic mitigation. Through a systematic methodology that involved data collection, preprocessing, model training, system integration, and rigorous testing, the proposed solution achieved its core objectives: accurate SQL Injection attack detection, immediate network-level response, continuous system learning, and comprehensive monitoring and reporting.

The Machine Learning-based detection engine demonstrated outstanding performance, achieving high accuracy, precision, recall, and F1-score across multiple evaluation scenarios. The system was able to detect various forms of SQL Injection attacks, including union-based, error-based, blind, and obfuscated queries, with minimal false positives and false negatives. Real-time processing capabilities were validated through stress testing, confirming the system's readiness for production deployment in environments with high query volumes and dynamic traffic.

Integration with an SDN controller allowed the system to move beyond traditional detection-only frameworks and actively respond to threats by dynamically modifying network behavior. This real-time mitigation significantly reduced the attack window and minimized potential damage, providing an intelligent and proactive security layer for modern web applications and databases.

The development of a user-friendly, web-based platform added further value by offering real-time dashboards, detection logs, alert management, and report generation features. Security was embedded at every layer of the system, from secure API communications to strict role-based access controls, ensuring operational integrity and compliance readiness.

Despite these achievements, the research also acknowledged several limitations, including dependence on training data diversity, vulnerability to adversarial attacks, scalability constraints under extremely high loads, and the need for more advanced mitigation strategies. These limitations form the basis for future improvement directions, such as integrating adversarial robustness, enhancing system scalability through microservices, incorporating advanced deception-based mitigation tactics, and achieving higher levels of regulatory compliance.

REFERENCES

[1] Halfond, W. G. J., Viegas, J., & Orso, A. (2006). *A classification of SQL-injection attacks and countermeasures*. In Proceedings of the IEEE International Symposium on Secure Software Engineering (pp. 13-15).

[2] Bertino, E., Sandhu, R. (2005). *Database security—concepts, approaches, and challenges*. IEEE Transactions on Dependable and Secure Computing, 2(1), 2–19. https://doi.org/10.1109/TDSC.2005.9

[3] Shin, D., & Song, Y. (2017). *SQL Injection Detection Techniques: A Survey*. KSII Transactions on Internet and Information Systems, 11(3), 1470–1489. https://doi.org/10.3837/tiis.2017.03.021

[4] Moustafa, N., Turnbull, B., & Hu, J. (2018). *An ensemble intrusion detection technique based on proposed statistical flow features for protecting network traffic of Internet of Things*. IEEE Internet of Things Journal, 6(3), 4815–4825. https://doi.org/10.1109/JIOT.2018.2871719

[5] Kreutz, D., Ramos, F. M. V., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2015). *Software-Defined Networking: A Comprehensive Survey*. Proceedings of the IEEE, 103(1), 14–76. https://doi.org/10.1109/JPROC.2014.2371999

[6] Sommer, R., & Paxson, V. (2010). *Outside the closed world: On using machine learning for network intrusion detection*. In 2010 IEEE Symposium on Security and Privacy (pp. 305–316). IEEE. https://doi.org/10.1109/SP.2010.25

[7] Rajab, K. Z., Alwan, A. A., & Obaid, F. M. (2020). *SQL Injection Attack Detection Using Machine Learning*. International Journal of Advanced Computer Science and Applications, 11(5), 252–258. https://doi.org/10.14569/IJACSA.2020.0110532

[8] Ahmad, I., Nam, H., Shahab, S., Kim, H., & Kim, D. (2019). *Machine Learning Approaches for Securing Industrial Wireless Sensor Networks: A Survey*. IEEE Access, 7, 96532–96545. https://doi.org/10.1109/ACCESS.2019.2929084