

**BLOCKCHAIN-BASED INTRUSION DETECTION SYSTEM WITH  
ARTIFICIAL INTELLIGENCE**

BY  
SOH WEN KAI

A REPORT  
SUBMITTED TO  
Universiti Tunku Abdul Rahman  
in partial fulfillment of the requirements  
for the degree of  
BACHELOR OF COMPUTER SCIENCE (HONOURS)  
Faculty of Information and Communication Technology  
(Kampar Campus)

JANUARY 2025

## **COPYRIGHT STATEMENT**

© 2025 Soh Wen Kai. All rights reserved.

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

## **ACKNOWLEDGEMENT**

I would like to express my heartfelt gratitude to my supervisor, Ts Dr Gan Ming Lee, for offering me the incredible opportunity to develop my first cybersecurity and blockchain project. His continuous motivation, valuable thoughts, and constructive advice have been invaluable throughout this journey.

## **ABSTRACT**

This project presents the development of a Blockchain-Based Intrusion Detection System with Artificial Intelligence, designed to address the limitations of traditional intrusion detection frameworks that lack contextual awareness, secure alert storage, and automated response. The system integrates a rule-based signature engine with a large language model to detect both known and previously unseen network threats through real-time traffic analysis. Signature-based detection matches flows against predefined patterns, while the LLM performs context-aware reasoning to identify complex or ambiguous behaviours, producing alerts with human-readable explanations and severity levels. To ensure alert integrity and traceability, all detection events are logged to a private Ethereum blockchain using smart contracts, providing a decentralised and tamper-resistant audit trail. Simultaneously, off-chain logging is enabled to ensure efficient notification of intrusion events. A web-based dashboard offers live monitoring of packet capture, active flows, alert statistics, and blockchain synchronisation. The system was designed for modularity, with configurable components for flow processing, AI analysis, and on-chain logging. It achieved a detection accuracy of 93.95% and a false positive rate of 5.00%, confirming the effectiveness of its hybrid detection approach. This prototype demonstrates the feasibility of combining AI and blockchain technologies to build an IDS that is not only accurate but also transparent, explainable, and resistant to tampering, thus making it a promising foundation for modern, resilient cybersecurity systems.

### **Area of Study:**

Cybersecurity, Blockchain

### **Keywords:**

Intrusion Detection System (IDS), Blockchain, Large Language Model (LLM), Tamper-Proof Logging, AI Explainability

## TABLE OF CONTENTS

<b>TITLE</b>	<b>I</b>
<b>COPYRIGHT STATEMENT</b>	<b>II</b>
<b>ACKNOWLEDGEMENT</b>	<b>III</b>
<b>ABSTRACT</b>	<b>IV</b>
<b>TABLE OF CONTENTS</b>	<b>V</b>
<b>LIST OF FIGURES</b>	<b>XI</b>
<b>LIST OF TABLES</b>	<b>XIII</b>
<b>LIST OF SYMBOLS</b>	<b>XIV</b>
<b>LIST OF ABBREVIATIONS</b>	<b>XV</b>
<b>CHAPTER 1 INTRODUCTION</b>	<b>1</b>
1.1 Problem Statement	1
1.1.1 Absence of an Integrated IDS for Detection and Contextual Analysis	1
1.1.2 Limited Automation and Decentralisation in Threat Response	2
1.1.3 Vulnerability of Alerts to Tampering and Unauthorised Access	3
1.2 Motivations	4
1.2.1 Addressing the Growing Complexity of Cyber Threats	4
1.2.2 Ensuring Integrity and Trust in Security Alerts	4
1.2.3 Enhancing Explainability in AI-Based Intrusion Detection Systems	5
1.3 Project Objective	6
1.3.1 To Develop a Unified IDS for Detection and Contextual Analysis	6
1.3.2 To Automate and Decentralise Threat Response via Smart Contracts	6

1.3.3	To Enable Secure and Verifiable Access to Alert Records	7
1.4	Project Scope	8
1.5	Impact, Significance, and Contribution	10
1.6	Background Information	11
1.7	Report Organisation	13
<b>CHAPTER 2 LITERATURE REVIEW</b>		<b>14</b>
2.1	Review of Relevant Technologies	14
2.1.1	Blockchain	14
2.1.2	Large Language Model	16
2.1.3	Intrusion Detection System	17
2.2	Review of Current Integrations	20
2.2.1	Integration of AI in Intrusion Detection Systems	20
2.2.2	Integration of Blockchain in Cybersecurity	21
2.2.3	Integration of AI and Blockchain in Intrusion Detection Systems	23
2.3	Review of Current Applications	25
2.3.1	Snort	25
2.3.2	Suricata	26
2.4	Summary and Research Gaps	28
<b>CHAPTER 3 SYSTEM METHODOLOGY</b>		<b>30</b>
3.1	Development Methodology	30
3.2	System Architecture Overview	33
3.2.1	System Architecture Diagram	33
3.2.2	Use Case Diagram and Description	35
3.2.3	Activity Diagram	44
3.3	Design Specifications	51

3.3.1	Software Requirements	51
3.3.2	Hardware Requirements	53
3.3.3	Functional Requirements	55
3.3.4	Non-Functional Requirements	56
3.3.5	Design Constraints	57
3.4	Development Timeline	59
<b>CHAPTER 4 SYSTEM DESIGN</b>		<b>61</b>
4.1	System Architecture	61
4.1.1	High-Level System Flow	61
4.1.2	System Block Diagram and Data Flow	63
4.2	Component-Level Design	67
4.2.1	Packet Capture Module	67
4.2.2	LLM-Based Detection Module	69
4.2.3	Signature-Based Detection Module	72
4.2.4	Alert Logger Module	74
4.2.5	Blockchain Logger Module	76
4.2.6	Frontend GUI Module	79
4.3	Database and Storage Design	81
4.3.1	Network Flow Data Handling	81
4.3.2	Alert Database and File-Based Log	81
4.3.3	Blockchain-Based Immutable Storage	83
4.3.4	Temporary Queues and Runtime Storage	83
4.4	Smart Contract Design	84
4.5	Communication Interface Design	86
4.6	Compilation and Setup Design	89
<b>CHAPTER 5 SYSTEM IMPLEMENTATION</b>		<b>91</b>
5.1	Environment and Tools Setup	91

5.1.1	Hardware Setup	91
5.1.2	Software Setup	93
5.2	Blockchain Implementation	96
5.2.1	Installing and Running Ganache	96
5.2.2	Setting Up the Python Environment	99
5.2.3	Installing and Configuring the Solidity Compiler	99
5.2.4	Compiling and Deploying the Smart Contract	100
5.2.5	Saving Contract Metadata	101
5.2.6	Initialising the System with Blockchain Support	101
5.3	LLM Implementation	102
5.3.1	Installing Ollama	102
5.3.2	Pulling the Required Model	103
5.3.3	Configuring the Environment for LLM	103
5.3.4	Initialising the LLM Detection Engine	104
5.3.5	Implementing the RAG Mechanism	104
5.3.6	Formatting and Sending Network Flow Data	105
5.3.7	Handling and Logging AI-Generated Alerts	105
5.4	Backend Services Implementation	107
5.4.1	Setting Up the Flask Web Framework	107
5.4.2	Setting Up the Flask Web Framework	107
5.4.3	Initialising Packet Capture with Scapy	107
5.4.4	Configuring the Packet Processing Queue	108
5.4.5	Implementing Flow-Based Traffic Analysis	108
5.4.6	Integrating the Signature-Based Detection Engine	109
5.4.7	Managing Alerts and Local Storage	109
5.4.8	Configuring the Environment for Backend Services	110
5.4.9	Enabling Cross-Origin Requests and Frontend Communication	110
5.5	Frontend and UI Implementation	111
5.5.1	Setting Up the Project Structure	111
5.5.2	Designing the User Interface with HTML	111
5.5.3	Styling the Interface with CSS	112



5.5.4	Enabling Interactivity with JavaScript	112
5.5.5	Creating Real-Time Data Visualisation and Feedback	113
5.5.6	Managing Responsive Design and Browser Compatibility	113
5.5.7	Linking Frontend to Backend Services	113
5.6	System Operation	115
5.6.1	System Launch and Dashboard Overview	115
5.6.2	Network Interface Selection and Packet Capture Start	117
5.6.3	Active Network Flows Monitoring	120
5.6.4	Intrusion Alerts List	122
5.6.5	Signatures List	123
5.6.6	Blockchain Status and Synced Alerts	125
5.6.7	System Status	128
5.6.8	System Settings	129
5.6.9	Shutdown Status	131
5.7	Implementation Challenges and Solutions	133
5.7.1	Managing Real-Time Network Traffic	133
5.7.2	Flow Assembly and Analysis	133
5.7.3	Blockchain Integration	134
5.7.4	LLM Integration	134
5.7.5	Real-Time Frontend Components	134
<b>CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION</b>		<b>135</b>
6.1	Testing Methodology	135
6.1.1	Objectives of Testing	135
6.1.2	Test Setup	137
6.1.3	Test Procedures	139
6.2	System Testing	142
6.2.1	Functional Testing	142
6.2.2	Non-Functional Testing	143
6.3	Discussion	146
6.3.1	Interpretation of Results	146

## Table of Contents

6.3.2	Limitations Observed	147
6.3.3	Validity of the Evaluation	148
<b>CHAPTER 7 CONCLUSION AND RECOMMENDATION</b>		<b>150</b>
7.1	Conclusion	150
7.2	Recommendation	152
<b>REFERENCES</b>		<b>153</b>
<b>APPENDICES</b>		<b>A-1</b>
<b>POSTER</b>		<b>A-2</b>

## LIST OF FIGURES

<b>Figure No.</b>	<b>Title</b>	<b>Page</b>
Figure 3.1.1	System Prototyping Methodology	30
Figure 3.2.1	System Architecture Diagram	33
Figure 3.2.2	Use Case Diagram	35
Figure 3.2.3	Activity Diagram: Start Packet Capture	44
Figure 3.2.4	Activity Diagram: Stop Packet Capture	45
Figure 3.2.5	Activity Diagram: View Real-Time Network Flows	46
Figure 3.2.6	Activity Diagram: View Intrusion Alerts	47
Figure 3.2.7	Activity Diagram: View Intrusion Signatures	48
Figure 3.2.8	Activity Diagram: View System Status	49
Figure 3.2.9	Activity Diagram: View System Settings	50
Figure 3.4.1	Project Development Timeline	59
Figure 4.1.1	High Level System Flow	62
Figure 4.1.2	System Block Diagram	64
Figure 4.6.1	Folder Structure	90
Figure 5.1.1	Laptop Specifications	92
Figure 5.2.1	Ganache Workspace Configuration	97
Figure 5.2.2	Ganache Server Configuration	98
Figure 5.2.3	Ganache Environment	98
Figure 5.2.4	Smart Contract Deployment	100
Figure 5.2.5	Contract Creation on Blockchain	101
Figure 5.3.1	Ollama Local Server	102
Figure 5.3.2	Running LLM Hosted via Ollama	103
Figure 5.3.3	Interaction with LLM via HTTP	103
Figure 5.6.1	System Logs Showing Successful System Launch	116
Figure 5.6.2	Idle System Dashboard	117
Figure 5.6.3	Network Interface Selection	118
Figure 5.6.4	Configured Packet Capture Settings	119
Figure 5.6.5	Dashboard View During Active Capture Session	120
Figure 5.6.6	Active Network Flow Interface	122
Figure 5.6.7	Intrusion Detection Alerts Table	123

## List of Figures

Figure 5.6.8 Attack Signatures Table	125
Figure 5.6.9 Ganache Blockchain Dashboard	126
Figure 5.6.10 Syncing of Historical Blockchain Alert Logs	127
Figure 5.6.11 Final Blockchain Status	127
Figure 5.6.12 System Status Dashboard	128
Figure 5.6.13 System Settings Interface	131
Figure 5.6.14 Console Output During System Shutdown	132
Figure 6.2.1 Confusion Matrix	145

## LIST OF TABLES

<b>Table No.</b>	<b>Title</b>	<b>Page</b>
Table 3.2.1 UC001:	Start Packet Capture	37
Table 3.2.2 UC002:	Stop Packet Capture	38
Table 3.2.3 UC003:	View Real-Time Network Flows	39
Table 3.2.4 UC004:	View Intrusion Alerts	40
Table 3.2.5 UC005:	View Intrusion Signatures	41
Table 3.2.6 UC006:	View System Status	42
Table 3.2.7 UC007:	Manage System Settings	43
Table 3.3.1	Software Requirements	52
Table 3.3.2	Hardware Requirements	54
Table 3.3.3	Functional Requirements	56
Table 3.3.4	Non-Functional Requirements	57
Table 3.4.1	Phased Breakdown of the Development Plan	60
Table 4.3.1	Database Schema for Alert Storage	82
Table 5.1.1	Programming Language Requirements	93
Table 5.1.2	Library and Framework Requirements	94
Table 5.1.3	Tool and Platform Requirements	95
Table 5.1.4	Operating System Requirements	95
Table 6.1.1	Component Specification for Test Setup	137
Table 6.1.2	System Performance Metrics	139
Table 6.2.1	Functional Testing Result	143
Table 6.2.2	Non-Functional Test Result	144

## **LIST OF SYMBOLS**

$\approx$  Approximately equal

## LIST OF ABBREVIATIONS

ABI	Application Binary Interface
AI	Artificial Intelligence
API	Application Programming Interface
Bi-GRU	Bidirectional Gated Recurrent Unit
BPF	Berkeley Packet Filter
CLI	Command-Line Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial of Service
DoS	Denial of Service
ETH	Ether (Ethereum's native cryptocurrency)
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HIDS	Host-Based Intrusion Detection System
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
JS	JavaScript
JSON	JavaScript Object Notation
LAN	Local Area Network
LLM	Large Language Model
LSTM	Long Short-Term Memory
NIC	Network Interface Card
NIDS	Network-Based Intrusion Detection System
PBFT	Practical Byzantine Fault Tolerance
PoA	Proof of Authority
PoW	Proof of Work

## List of Abbreviations

RAG	Retrieval-Augmented Generation
RAM	Random Access Memory
REST	Representational State Transfer
RNN	Recurrent Neural Network
SDK	Software Development Kit
SIEM	Security Information and Event Management
SQL	Structured Query Language
SSD	Solid-State Drive
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
VPN	Virtual Private Network
WAN	Wide Area Network
XSS	Cross-Site Scripting



## **CHAPTER 1 INTRODUCTION**

This chapter provides an overview of the project titled Blockchain-Based Intrusion Detection System with Artificial Intelligence (hereinafter referred to as “the project”), laying the foundation for the work undertaken. It begins by presenting the background and context of the project, highlighting the current challenges in intrusion detection and the motivation behind the proposed solution. The chapter then outlines the specific objectives that guide the project’s development and defines the scope to clarify the boundaries and focus areas. Furthermore, it summarises the key contributions made through this project and explains how they address existing limitations in the field. Finally, the chapter concludes with an outline of the report structure to help in understanding the flow and organisation of the subsequent chapters.

### **1.1 Problem Statement**

This section outlines the issues to address, including current challenges that need remediation and areas for potential optimisation and improvement.

#### **1.1.1 Absence of an Integrated IDS for Detection and Contextual Analysis**

Intrusion Detection Systems (IDSs) are essential components in modern network security. Yet, in the current cybersecurity landscape, IDSs often operate in isolated environments with limited ability to contextualise threats or correlate them across multiple sources in real time. Traditional IDSs typically rely on either signature-based or anomaly-based mechanisms, but they rarely integrate advanced AI analysis and immutable logging to provide comprehensive insight. This siloed architecture restricts the system’s ability to adapt and understand evolving attack patterns, especially in dynamic networks. As highlighted by Alshamrani et al., existing IDS frameworks frequently fail to provide adequate context for detected threats, resulting in high false positive rates and inadequate incident response capabilities [1]. Furthermore, many IDS frameworks are constrained by their inability to analyse high-dimensional network data in real time and adapt to sophisticated attacks through contextual correlation. Similarly, Khenwar and Nawal highlight that traditional IDS architectures do not incorporate

advanced analytical capabilities needed for meaningful threat classification, especially in heterogeneous network environments [2]. This demonstrates a critical gap: the absence of a unified, intelligent, and context-aware intrusion detection mechanism capable of providing comprehensive and accurate threat insight in real time. A potential solution involves an IDS framework enhanced with Artificial Intelligence (AI) like Large Language Models (LLMs) with contextual analysis capabilities that integrates with existing networks.

### **1.1.2 Limited Automation and Decentralisation in Threat Response**

Despite advances in intrusion detection technologies, most systems still depend heavily on manual processes for threat response, which can delay mitigation and increase the risk of damage during fast-moving attacks. The lack of automation in threat handling limits the scalability and effectiveness of security operations, especially in environments where threats evolve rapidly. Furthermore, centralised control mechanisms present a single point of failure, making the response process vulnerable to disruption or manipulation. As stated by Zuech et al., conventional IDS frameworks lack real-time automated capabilities that can adapt dynamically to detected threats without human intervention [3]. In addition, the absence of decentralised decision-making or action-sharing among distributed nodes hampers resilience and responsiveness in large or segmented networks. Research by Dorri et al. has highlighted the potential of decentralised architectures in improving the robustness and fault-tolerance of cybersecurity systems, yet such models remain underutilised in current IDS implementations [4]. This underlines the need for intrusion detection systems that incorporate both automated response mechanisms and decentralised control to enable faster, more reliable threat mitigation. The problem can be addressed by implementing smart contract-driven alerts, which automate the detection and response workflows based on predefined conditions, reducing reliance on human intervention.

### **1.1.3 Vulnerability of Alerts to Tampering and Unauthorised Access**

In many existing intrusion detection systems, the alerts generated during detection processes are stored in centralised databases or log files that are susceptible to tampering, deletion, or unauthorised access. This poses a serious threat to the integrity and trustworthiness of forensic data, which is essential for post-incident analysis, compliance audits, and legal investigations. Without secure and verifiable alert storage, attackers who gain access to the system may alter or erase evidence of their intrusion to avoid detection and accountability. According to Diana et al., the lack of immutable logging mechanisms in conventional IDS architectures significantly weakens the overall security posture by enabling the manipulation of historical data [5]. Additionally, centralised systems often lack transparency and verifiability, which undermines confidence in the recorded security events and their traceability in distributed environments. These limitations highlight the pressing need for IDS frameworks that secure alert data against tampering through verifiable, decentralised storage mechanisms. Fortunately, this problem in IDS can be potentially mitigated using blockchain technology's decentralised and immutable ledger to securely store logs, alerts, network, and intrusion information.

## **1.2 Motivations**

This section provides a comprehensive overview of the underlying reasons and broader context that drive the need for the project. It explores the practical challenges, technological gaps, and industry trends that have inspired the project's conception. By examining these motivating factors, the section establishes the relevance and significance of the proposed work within its intended application domain.

### **1.2.1 Addressing the Growing Complexity of Cyber Threats**

In the modern digital ecosystem, cyber threats have become increasingly complex, frequent, and stealthy through advanced evasion techniques, automation, and even AI to bypass traditional defence mechanisms. This surge in sophistication places immense pressure on conventional IDSs, which often rely on static rules or signature-based models that are ill-equipped to handle novel or polymorphic attacks. Moreover, as critical infrastructures, financial systems, healthcare networks, and national services grow more interconnected, the consequences of a successful cyberattack have become far-reaching, affecting safety, economic stability, and public trust. The reactive nature and limited intelligence of most IDS result in delayed threat response and poor adaptability, especially in high-speed or large-scale network environments. This project is motivated by the urgent need to bridge this gap by enhancing IDS capabilities through automation, intelligence, and trust. By integrating artificial intelligence for dynamic threat detection and blockchain for tamper-proof alert verification, this project aims to deliver a forward-looking IDS architecture, one that is not only reactive but predictive, decentralised, and resilient in the face of today's and tomorrow's cyber threats.

### **1.2.2 Ensuring Integrity and Trust in Security Alerts**

In the context of cybersecurity, the integrity and trustworthiness of security alerts are essential, particularly for post-incident analysis, automated response, and compliance reporting. However, many existing IDS frameworks store alerts in centralised systems that are vulnerable to manipulation, unauthorised modification, or deletion, hence threatening the credibility of the entire detection process. In environments where decisions must be made quickly and evidence must be auditable, any compromise in

alert integrity weakens both operational confidence and forensic accuracy. This project is driven by the critical need to ensure that once an alert is generated, it remains immutable and verifiable. By incorporating blockchain technology as a decentralised and tamper-resistant ledger, the system can guarantee that all alerts are securely recorded, traceable, and immune to unauthorised alterations. This enhances not only technical reliability but also stakeholder trust, enabling organisations to rely on their IDS data with greater confidence for decision-making, auditing, and legal validation.

### **1.2.3 Enhancing Explainability in AI-Based Intrusion Detection Systems**

As artificial intelligence becomes more prevalent in IDSs, a key challenge that emerges is the lack of explainability in how AI models detect and classify threats. Many AI-based systems operate as black boxes, providing alerts without clear reasoning or context, which can reduce trust in their outputs and make it difficult for security analysts to validate or act upon them effectively. This issue is especially critical in environments where accountability, transparency, and quick decision-making are essential. Without understandable justifications, even accurate alerts may be disregarded or misinterpreted. This project is motivated by the need to improve the interpretability of AI-driven IDS by incorporating mechanisms that provide contextual explanations for each alert. By integrating natural language generation and structured metadata into the detection process, the system will offer more transparent and human-readable insights, empowering analysts to make faster, more confident decisions and contributing to greater trust in AI-assisted cybersecurity operations.

### **1.3 Project Objective**

This section defines the overall aim of the project by clearly stating what it seeks to achieve. It outlines the intended outcomes at a high level and serves as a guiding reference for the development process. The objectives presented here help establish the direction of the work and provide measurable goals against which the project's success can be evaluated.

#### **1.3.1 To Develop a Unified IDS for Detection and Contextual Analysis**

The primary objective of this project is to develop a unified IDS that combines real-time threat detection with contextual analysis to enhance the accuracy and relevance of alerts. Unlike traditional IDS that rely solely on signature matching or basic anomaly detection, this system will integrate multiple detection techniques, including rule-based methods and LLM-driven models, to analyse network flows holistically. The system will be designed to process at least 100 packets per second and detect threats with a minimum target accuracy of 90%, verified through labelled test datasets. By correlating traffic patterns, protocol behaviours, and flow statistics, the system aims to generate alerts that are not only precise but also enriched with meaningful context, thus providing explainability. This approach is intended to reduce false positives, improve threat interpretation, and support more informed and timely responses by security analysts.

#### **1.3.2 To Automate and Decentralise Threat Response via Smart Contracts**

This project aims to automate and decentralise the threat response process by integrating smart contracts into the IDS architecture. Traditional intrusion detection systems often require manual intervention to respond to threats, which can delay mitigation efforts and increase exposure to ongoing attacks. By leveraging blockchain-based smart contracts, the system will enable predefined, automated response actions such as on-chain alert logging, triggered within 5 seconds upon detection of an intrusion. The smart contract will be deployed and tested on a private blockchain network and will record 100% of the alerts generated by the IDS, ensuring traceability as well as accountability. This decentralised approach ensures that response

mechanisms are executed consistently and without dependence on a central authority, thereby enhancing system resilience, reducing response time, and eliminating single points of failure.

### **1.3.3 To Enable Secure and Verifiable Access to Alert Records**

This project seeks to enable secure and verifiable access to intrusion alert records by integrating a blockchain-based logging mechanism within the IDS framework. Conventional systems often store alerts in centralised or unsecured databases, making them susceptible to tampering, deletion, or unauthorised access. To address this, the proposed system will log 100% of generated alerts onto a private blockchain, ensuring that each record is immutable, timestamped, and transparently verifiable. Access to alert records will be verified through an API interface that retrieves and validates on-chain entries, with a response accuracy of 100% for successfully logged events. This will allow authorised users, such as security analysts or auditors, to access trustworthy alert data for incident response, forensic analysis, and compliance reporting. By providing a decentralised and tamper-resistant record of security events, the system aims to strengthen the overall credibility and reliability of intrusion detection outputs.

## 1.4 Project Scope

This project will focus on developing a prototype of a “Blockchain-Based Intrusion Detection System with Artificial Intelligence”, tailored for use in organisational network environments. The primary deliverable will be a fully functional prototype that demonstrates the feasibility of integrating blockchain and AI to enhance network security. This prototype will act as a proof of concept, showcasing real-time threat detection, contextual analysis, and automated response via smart contracts. While functionally complete for demonstration purposes, the system will not be developed as a market-ready or production-level product.

The project will involve the development of several core components that work together to form a functional prototype of the proposed system. These include AI-based detection modules that combine rule-based techniques with LLM-driven analysis to detect both known and unknown threats in real time. A blockchain logging module will be implemented using smart contracts to ensure the immutability, decentralisation, and automated response of security alerts. The backend system will manage data flow operations, including traffic capture, packet processing, and flow analysis. A user interface will be developed to allow users to monitor system status, view alerts, and control IDS operations in a clear and accessible manner. Finally, integration modules will ensure seamless communication between the detection engine, blockchain layer, and frontend interface, enabling the system to operate as a cohesive and responsive intrusion detection framework.

The project will also include functional as well as non-functional testing using controlled datasets and simulated traffic to verify the performance and functionalities of the system. However, the project will intentionally exclude several areas that fall outside the scope of a prototype-focused development effort. These include full-scale UI/UX refinement beyond essential usability and basic navigation, as the emphasis is on functionality rather than visual design or user experience optimisation. Scalability testing under extreme network loads or deployment across distributed infrastructures will not be conducted, as the system is intended for demonstration in a controlled environment. Integration with enterprise-grade systems such as Security Information and Event Management (SIEM) tools or external APIs is also beyond the scope, as the



project is not aimed at production-level interoperability. Additionally, long-term system maintenance, such as regular update cycles, patch management, and ongoing user support will not be covered, given the academic and proof-of-concept nature of the work.

To summarise, the overall goal is to deliver a working prototype that validates the core ideas of automation, contextual detection, and tamper-proof alert storage using blockchain and AI, within a controlled development environment.

## 1.5 Impact, Significance, and Contribution

This project has the potential to significantly influence the future design of intrusion detection systems by demonstrating how blockchain and artificial intelligence can be synergised to address critical shortcomings in conventional security solutions. By combining real-time detection, contextual analysis, and tamper-proof alert logging, the system offers a modern approach to network security that aligns with the demands of increasingly complex and distributed digital environments, while addressing the critical gaps in current organisational cybersecurity frameworks involving IDSs. The outcome of this project could lead to improved response times, reduced false positives, and enhanced trust in alert data, features that are highly valuable for organisations handling sensitive or large-scale network traffic.

The significance of this project lies in its integration of decentralised technologies and intelligent analytics into a single, cohesive intrusion detection framework. In contrast to traditional IDS models that often operate in isolation with limited interpretability, this system introduces an innovative architecture that ensures alert transparency, contextual relevance, and automation of response. This project addresses multiple pressing cybersecurity challenges, such as threat complexity, alert integrity, and operational latency, making it highly relevant to current academic research and industrial cybersecurity practices. In the long term, the adoption of such technologies could lead to a safer, more secure digital environment, benefiting not just individual organisations, but society as a whole.

This project contributes to the field of cybersecurity by delivering a functional prototype that proves the feasibility of using blockchain and AI in a unified IDS. The novelty of this project lies in the implementation of LLM-based contextual analysis, smart contract-driven automation, and verifiable alert logging, all integrated into a single system. While blockchain-based security solutions have been primarily focused on IoT environments, this project innovatively applies blockchain to the broader and more complex context of organisational cybersecurity. The use of smart contracts to automate incident response processes is another innovative aspect that sets this project apart from traditional approaches with relatively greater dependency on manual interventions.

## 1.6 Background Information

In today's increasingly interconnected digital world, cyber threats have become more sophisticated, frequent, and coordinated, largely driven by the rapid advancement of technologies and the growing use of artificial intelligence, including generative models. This evolution poses serious challenges to conventional cybersecurity strategies, particularly in the areas of threat prevention, detection, and response. Studies have shown that individuals and organisations alike remain highly vulnerable to a wide range of attacks such as Denial of Service (DoS), Distributed Denial of Service (DDoS), phishing, ransomware, malware, SQL injection, and zero-day exploits [6]. Traditional security tools, including firewalls and standalone IDS, often lack the intelligence and adaptability required to identify these modern threats, especially those that do not match known attack signatures.

Intrusion, as defined by Khraisat et al. [7], refers to unauthorised activities that compromise the Confidentiality, Integrity, or Availability (CIA) of an information system. IDSs, whether hardware- or software-based, are designed to monitor network traffic and system behaviour to detect such malicious actions. These systems generally fall into two main categories: Signature-Based Intrusion Detection Systems (SIDSs), which identify known threats by matching patterns from a database, and Anomaly-Based Intrusion Detection Systems (AIDSs), which use statistical or behavioural models to detect deviations from normal network activity [7], [8]. However, both models suffer from limitations: SIDS cannot detect novel or obfuscated attacks, while AIDS often generates high false positive rates and struggles with adaptability.

To overcome these limitations, recent developments have turned to artificial intelligence, particularly machine learning (ML) and deep learning (DL) models. As detailed by Kaur et al. [9], AI plays several key roles in cybersecurity, including automating tasks, identifying threats, preventing attacks, detecting vulnerabilities, responding to incidents, and aiding in recovery efforts. To expand on this, AI algorithms excel at analysing large volumes of data, such as network traffic, to reveal potential threats and vulnerabilities. By detecting patterns and identifying subtle anomalies, AI can reveal new forms of cyber threats that traditional, signature-based systems might miss. Also, these AI systems are able to continuously adapt to emerging

threats through learning algorithms, which not only improve predictive accuracy but also reduce false positives, providing stronger protection against increasingly sophisticated cyberattacks [10]. However, it is important to note that AI-based systems often require significant computational resources.

On a related note, LLMs have emerged as a powerful enhancement to cybersecurity frameworks due to their ability in contextual understanding and text generation. Trained on vast cybersecurity datasets, LLMs can identify contextual patterns and correlations that traditional AI models may overlook due to their limited feature sets. Their ability to understand unstructured or semi-structured flow data allows them to generate more accurate and context-aware threat classifications. Furthermore, LLMs can provide human-readable explanations for each detection decision, delivering natural language insights that increase the transparency, interpretability, and usability of alerts. This level of explainability not only improves the analysts' trust in the system but also supports faster and more confident incident response.

Parallel to AI advancements, blockchain technology has gained attention for its ability to enhance data integrity and trust in distributed systems. Originally developed for cryptocurrencies, blockchain functions as a decentralised, tamper-resistant ledger that records transactions or events immutably across a peer-to-peer network [11]. In the context of cybersecurity, blockchain can be used to log security alerts in a way that prevents deletion or alteration, ensuring forensic traceability. Additionally, the use of smart contracts, which are essentially self-executing scripts triggered by defined conditions, enables automation of key security operations such as real-time alert validation and response without relying on a central authority [12], [13].

The integration of AI, LLMs, and blockchain into IDS design represents a promising new direction in cybersecurity research and practice. As highlighted by recent works [14], combining these technologies can significantly improve detection accuracy, reduce response time, and establish a transparent, decentralised trust layer for security operations. This project aims to build upon these concepts by developing a prototype system that unifies LLM-based contextual detection with blockchain-backed alert logging and smart contract-driven automation, tailored specifically for organisational network environments.

## 1.7 Report Organisation

This report is structured into seven chapters, each logically progressing from the project's background to its evaluation and conclusions. Chapter 1 introduces the project by outlining its background, motivation, problem statement, objectives, scope, and overall structure of the report. Chapter 2 reviews the existing literature, covering IDSs, AI-based threat analysis, blockchain applications in cybersecurity, and related works that informed the system design. Chapter 3 focuses on the methodology adopted, detailing the tools, frameworks, datasets, system architecture, and design principles that guided the development process.

Chapter 4 presents the system implementation in a modular format, describing key components such as packet capture, flow assembly, signature detection, LLM-based reasoning, blockchain integration, and the web interface. Each subsection highlights the core logic and design considerations for its respective module. Chapter 5 explains the system's integration and deployment process, including interface interactions, smart contract deployment, and operational workflow from packet capture to blockchain logging.

Chapter 6 evaluates the system through both functional and non-functional testing. It includes detailed test results, interpretation, limitations encountered during development and testing, and a validity analysis of the evaluation approach. Finally, Chapter 7 concludes the report by summarising the project's achievements, highlighting contributions, and offering recommendations for future improvements.

## **CHAPTER 2 LITERATURE REVIEW**

### **2.1 Review of Relevant Technologies**

This section explores the core technologies that are relevant to The Project, including their functionalities, advantages, and limitations. It provides a foundational understanding of the tools, frameworks, and methodologies employed, establishing the technical context for the system's design and implementation.

#### **2.1.1 Blockchain**

Blockchain is a decentralised, append-only ledger system that facilitates transparent and tamper-evident record-keeping across multiple untrusted nodes. First introduced through the Bitcoin whitepaper in 2008 by Nakamoto, the technology has since evolved into a general-purpose framework supporting various applications beyond digital currencies, including supply chain tracking, identity verification, voting systems, and decentralised finance [15], [16].

At its core, blockchain ensures data integrity and distributed trust through cryptographic linking of blocks, consensus mechanisms, and decentralised replication. Each block contains a list of transactions, a timestamp, and a hash of the previous block, forming an immutable chain. Once added, data in a block cannot be altered without modifying all subsequent blocks across the network, a task that becomes computationally infeasible in properly decentralised systems [15].

Recent academic literature has examined the taxonomy of blockchain systems, categorising them into public, private, and consortium blockchains. Public blockchains, such as Bitcoin and Ethereum, offer full decentralisation and openness, but suffer from performance bottlenecks due to computationally expensive consensus protocols like Proof-of-Work (PoW). In contrast, private and consortium blockchains operate within restricted access groups and adopt lighter consensus mechanisms such as Proof-of-Authority (PoA) or Practical Byzantine Fault Tolerance (PBFT), enabling higher throughput and reduced latency [11], [17].

Several studies have also analysed scalability and performance issues within blockchain systems. According to Xu et al. [18], traditional blockchains face the “blockchain trilemma”, where decentralisation, scalability, and security cannot all be maximised simultaneously. Research efforts have since focused on innovations such as sharding, off-chain computation, and layer-2 protocols to address these constraints. For instance, projects like Lightning Network and Ethereum’s rollups aim to improve transaction throughput without compromising trust-lessness.

Another significant thread in blockchain research involves security and attack surfaces. While blockchain offers immutability, it is not inherently secure from all forms of attack. Smart contracts, for example, have introduced vulnerabilities such as re-entrancy and integer overflow bugs, leading to high-profile exploits. At the protocol level, threats such as 51% attacks and selfish mining highlight potential weaknesses in consensus integrity. Research by Prashanth et al. [19] surveys these vulnerabilities and proposes formal verification tools and hybrid consensus models as partial remedies.

Furthermore, scholars have explored blockchain’s interdisciplinary integration with fields such as Internet of Things (IoT), edge computing, and artificial intelligence (AI). In these domains, blockchain serves as a trust layer for distributed entities lacking central control. For example, in IoT systems, blockchain is used to manage identities, validate sensor data, and audit device behaviour without relying on central gateways [17], [20]. These studies demonstrate the extensibility of blockchain beyond finance, encouraging innovation in domains that require decentralised coordination.

Despite these advances, the integration of blockchain into real-world systems remains an ongoing challenge, particularly in contexts requiring real-time response, lightweight computation, and seamless interoperability. These limitations create opportunities for novel contributions, such as streamlined blockchain integration within high-performance computing environments or time-sensitive systems like intrusion detection.

### 2.1.2 Large Language Model

Large Language Models (LLMs) represent a significant advancement in the field of natural language processing (NLP) and machine learning. These models are characterised by their massive number of parameters which are often in the billions, as well as their ability to generate, summarise, translate, and understand human language with high fluency and coherence. LLMs such as OpenAI's GPT series, Google's Gemini, Meta's LLaMA, and Anthropic's Claude are built upon the Transformer architecture proposed by Vaswani et al. in 2017, which introduced the concept of self-attention and revolutionised sequence modelling [21].

A defining characteristic of LLMs is their ability to generalise across diverse tasks without task-specific training, a property often referred to as zero-shot or few-shot learning. This capability has been attributed to the scale of training data and model parameters. Brown et al. [22] demonstrated that scaling up both leads to emergent behaviours, where LLMs perform competitively on tasks ranging from arithmetic reasoning to code generation, without being explicitly programmed for them.

From a research standpoint, LLMs are seen as more than just predictive models—they are increasingly regarded as probabilistic knowledge bases. Studies have shown that LLMs encode factual information within their parameters, albeit with varying accuracy and robustness. Petroni et al. [23] introduced the concept of “language models as knowledge bases” by probing LLMs for factual recall using cloze-style prompts. Although effective for well-represented knowledge, the models struggle with niche or rarely seen information, revealing the limitations of statistical pattern matching in replacing structured reasoning.

Despite their impressive performance, LLMs face several well-documented limitations. One concern is hallucination, where the model generates factually incorrect but plausible-sounding text. This behaviour poses risks in applications requiring high reliability, such as medical diagnostics, legal advice, or security-sensitive systems. Another challenge is the lack of interpretability, making it difficult to understand or audit the decision-making process behind a given output. Furthermore, the enormous



computational resources required for training and inference raise questions about environmental impact, accessibility, and fairness [24].

Recent research has focused on controlling, fine-tuning, and safely deploying LLMs. Techniques like reinforcement learning with human feedback (RLHF), prompt engineering, and instruction tuning aim to align model outputs with human expectations and ethical standards. These efforts are particularly relevant in domains where the model acts as a decision-support tool or interacts directly with end-users.

In recent years, the application of LLMs in cybersecurity has gained increasing attention, particularly in the areas of threat intelligence analysis, anomaly detection, and automated incident response. LLMs are capable of parsing vast volumes of unstructured security data such as logs, alerts, and technical reports to identify patterns and generate contextual insights [25]. Researchers have explored their potential to assist in tasks like phishing detection, malware description generation, and vulnerability summarisation, due to their strong language understanding capabilities. However, the use of LLMs in real-time or adversarial cybersecurity environments remains limited, primarily due to concerns about accuracy, explainability, and the risk of model manipulation. This gap highlights the need for further research on how LLMs can be safely and reliably integrated into operational security systems.

### **2.1.3 Intrusion Detection System**

Intrusion Detection Systems (IDSs) are a fundamental component in the defence-in-depth approach to cybersecurity. They are designed to monitor network traffic or system activities for signs of malicious behaviour or policy violations. Traditionally, IDS are classified into two primary categories: signature-based and anomaly-based detection systems. Signature-based IDS operate by comparing observed events against a database of known attack patterns or signatures. While efficient and accurate for recognising well-known threats, they are ineffective against novel or zero-day attacks. In contrast, anomaly-based IDS use statistical, behavioural, or machine learning techniques to model normal activity and flag deviations as potential intrusions. This approach is more flexible but also prone to high false positive rates due to the dynamic nature of network behaviour [7], [8].

Over the years, IDS research has evolved significantly to address growing cybersecurity challenges, such as increasing attack sophistication and the proliferation of encrypted traffic. A wide range of algorithms have been employed for detection tasks, including decision trees, support vector machines, clustering techniques, and deep learning models. For example, Moustafa et al. [26] introduced UNSW-NB15, a benchmark dataset and IDS model using statistical flow features, which demonstrated improved accuracy in identifying various attack types compared to older datasets like KDD99. Similarly, deep learning approaches such as convolutional and recurrent neural networks have been adopted to extract temporal and spatial features from traffic flows, showing promising detection rates in controlled settings.

Another stream of research has focused on the deployment environments of IDS. These include Host-Based Intrusion Detection Systems (HIDS), which monitor individual machines, and Network-Based Intrusion Detection Systems (NIDS), which analyse network traffic. NIDS have garnered more attention in recent years due to the shift towards cloud-native and distributed infrastructures. However, the scalability and real-time performance of IDS remain major challenges. High-speed networks, encrypted protocols, and large volumes of traffic necessitate lightweight, efficient, and adaptive detection mechanisms. Researchers have proposed techniques such as feature selection, sampling, and parallelisation to improve processing efficiency [27].

Despite significant progress, several limitations remain in IDS research. First, the quality and availability of realistic datasets continue to hinder reproducibility and generalisability. Many public datasets are outdated, synthetically generated, or fail to represent modern attack techniques. Second, the issue of adversarial attacks against IDS models, particularly those using machine learning, has raised concerns about the robustness of such systems. Attackers may intentionally craft inputs to evade detection or poison training data to mislead the system. Third, IDS often suffer from a lack of interpretability, making it difficult for analysts to understand the rationale behind alerts and to distinguish true positives from benign anomalies [7].

In response to these limitations, recent efforts have begun to explore hybrid and context-aware IDS, combining multiple detection techniques and incorporating external knowledge such as threat intelligence feeds. There is also growing interest in integrating IDS with other security tools like SIEM (Security Information and Event Management) systems, firewalls, and threat hunting platforms to enable a more cohesive security ecosystem [8]. However, achieving seamless integration while maintaining performance, scalability, and reliability remains an open research challenge.

## **2.2 Review of Current Integrations**

This section explores existing attempts to integrate various technologies, such as blockchain, AI, and traditional IDSs to enhance cybersecurity mechanisms. The review focuses on how researchers and developers have combined these components to improve the accuracy, transparency, and reliability of threat detection and response.

### **2.2.1 Integration of AI in Intrusion Detection Systems**

The integration of Artificial Intelligence (AI) into Intrusion Detection Systems (IDSs) has become a major focus in modern cybersecurity research. Traditional IDSs, which rely on static rules and predefined signatures, are increasingly challenged by evolving attack vectors and complex network behaviours. AI techniques, particularly those involving machine learning (ML) and deep learning (DL), offer adaptive and data-driven solutions that can enhance detection accuracy, generalise to unseen threats, and reduce false positives.

Kim et al. [28] introduced an innovative AI-based IDS tailored for real-time web environments. Their hybrid model combined Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks to detect malicious HTTP traffic. CNN was employed to extract spatial features from request data, while LSTM captured temporal patterns in request sequences. A notable feature of their approach was the use of normalised UTF-8 character encoding, which facilitated efficient data processing without the computational cost of entropy calculations. The system was implemented using Docker containers, making it scalable for large-scale deployments.

In a different line of research, Park et al. [29] addressed the common issue of class imbalance in network intrusion datasets, a problem that frequently decreases the performance of ML-based IDSs. They proposed a novel framework that integrates Generative Adversarial Networks (GANs), specifically using the Wasserstein distance, to generate synthetic samples of underrepresented attack types. This synthetic augmentation, coupled with autoencoder-based feature learning, significantly improved detection rates for rare and low-frequency threats. The modular design of the system, comprising data preprocessing, GAN training, autoencoder learning, and predictive

modelling, demonstrated a comprehensive pipeline for building more inclusive and robust IDSs.

Focusing on Internet of Things (IoT) environments, Medjek et al. [30] presented a fault-tolerant IDS aimed at securing RPL (Routing Protocol for Low-Power and Lossy Networks). Their approach involved training lightweight machine learning models such as decision trees, random forests, and k-nearest neighbours on features extracted from simulated RPL-based attacks. While these traditional models achieved high detection accuracy, the study also emphasised the trade-off between performance and computational cost, which is an important consideration for resource-constrained IoT devices. Their work highlighted the need for optimisation strategies to balance detection efficacy with energy and processing limitations.

Across these studies, AI integration in IDSs shows clear promise, particularly in enabling adaptive learning, improving detection of novel threats, and addressing data-related challenges like imbalance and noise. However, common challenges persist, including the need for high-quality labelled datasets, computational overhead in real-time systems, and the risk of adversarial manipulation. These concerns have encouraged further research into hybrid models, feature engineering techniques, and resource-efficient learning algorithms tailored to specific deployment contexts.

### **2.2.2 Integration of Blockchain in Cybersecurity**

Blockchain technology has emerged as a transformative tool in the field of cybersecurity, offering new approaches to securing data, enhancing transparency, and decentralising trust. Unlike traditional centralised systems, blockchain operates on a distributed ledger framework where all transactions are cryptographically linked and stored across multiple nodes. This structure inherently reduces the risk of single points of failure, unauthorised data manipulation, and centralised attacks, making it a promising foundation for secure digital infrastructures.

Kshetri [31] presented a detailed analysis of blockchain's role in strengthening cybersecurity frameworks, particularly in comparison to conventional cloud-based models. The study emphasised that blockchain's decentralised nature eliminates the

dependency on central authorities, thereby reducing susceptibility to data tampering and privilege abuse. Through robust encryption and consensus mechanisms, blockchain ensures secure and verifiable communication between entities. These features are especially beneficial in dynamic and distributed environments like the Internet of Things (IoT), where traditional models often struggle with scalability, trust management, and vulnerability to large-scale attacks.

Extending blockchain's application into intrusion detection, Abubakar et al. [32] proposed a hybrid system that integrates blockchain with multiple IDS algorithms to improve detection accuracy and reduce false positives. Their system employed a weighted voting mechanism to fuse outputs from different AI-based detectors, with blockchain serving as an immutable ledger to store and share alert data. Tested using DARPA 99 and MIT Lincoln Labs datasets, the system demonstrated higher precision in detecting complex intrusion patterns, showcasing blockchain's potential in supporting collaborative and transparent threat intelligence.

Similarly, Babu et al. [33] explored the use of blockchain in securing IoT networks against Distributed Denial of Service (DDoS) attacks. Their approach utilised a permissioned blockchain to facilitate secure device authentication and decentralised alert sharing. A key innovation in their system was the integration of Physically Unclonable Functions (PUFs) to generate tamper-proof cryptographic keys, further enhancing the integrity of device communications. Coupled with ensemble machine learning for intrusion detection, the system significantly reduced false positives while maintaining high detection performance. The decentralised nature of the blockchain allowed alert propagation across all nodes, enabling timely and coordinated response to threats.

While these studies highlight the advantages of blockchain in strengthening cybersecurity systems, several challenges remain. Issues such as network latency, storage scalability, and consensus overhead can hinder the performance of blockchain in high-throughput or real-time environments. Moreover, the integration of blockchain with AI and existing security architectures requires careful consideration of compatibility, privacy, and resource constraints.

### 2.2.3 Integration of AI and Blockchain in Intrusion Detection Systems

The convergence of AI and blockchain technology in the domain of IDSs has opened new avenues for building intelligent, secure, and decentralised cybersecurity solutions. While AI contributes adaptability and learning capability to detect novel or evolving threats, blockchain introduces transparency, data integrity, and decentralised trust. Combined, these technologies aim to overcome the shortcomings of conventional IDSs, such as limited scalability, susceptibility to tampering, and high false positive rates.

Mishra [34] proposed a hybrid security model known as the Hybrid Intrusion Detection Tree (HIDT), which integrates a decision tree algorithm with blockchain to protect smart network environments. The IDS component utilises decision tree classification to detect anomalies in network traffic, while the blockchain layer provides a secure, decentralised reputation system. This system filters and verifies the integrity of incoming data, encrypts verified data using blockchain nodes, and stores them in immutable blocks. The dual-layered approach ensures that compromised devices are identified promptly and that recorded alerts are tamper-proof and verifiable across the network.

In another study, Saveetha and Maragatham [35] introduced a hybrid architecture that leverages DL and blockchain to build a robust and intelligent IDS. Their system combines LSTM networks with Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN) to detect and classify anomalies in network traffic. The blockchain component is employed to record threat detection events in a decentralised ledger, ensuring that once an anomaly is detected and verified, it cannot be altered or deleted. This provides a verifiable audit trail of security incidents, accessible to all participating nodes, and enhances the trustworthiness of the system's response to intrusions.

Expanding this approach to consumer IoT ecosystems, Kumar et al. [36] proposed an integrated framework that combines blockchain-based authentication with an explainable AI (XAI) intrusion detection mechanism. The system addresses vulnerabilities in smart city IoT networks by securing communication channels between IoT devices, fog nodes, and cloud servers using a Proof of Authority (PoA)

blockchain. Simultaneously, a Bidirectional Gated Recurrent Unit (Bi-GRU) model with an attention mechanism and a SoftMax classifier forms the IDS layer, enabling real-time threat detection with interpretable outputs. The inclusion of explainable AI is particularly noteworthy, as it addresses one of the key limitations of black-box deep learning systems by making detection decisions more transparent to security analysts.

These hybrid models illustrate the growing trend of integrating AI and blockchain in IDS design to capitalise on the strengths of both technologies. AI enhances detection capabilities through data-driven learning and adaptability, while blockchain ensures that detection outcomes and critical events are securely recorded and shared without centralised control. However, challenges remain in balancing computational efficiency with security guarantees, particularly in resource-constrained or latency-sensitive environments such as IoT.



## **2.3 Review of Current Applications**

This section reviews existing real-world systems and applications that are similar in scope or function to the proposed project. It evaluates their features, performance, and shortcomings, offering insights into best practices and areas where improvements or innovations are needed.

### **2.3.1 Snort**

Snort is one of the most widely adopted open-source Network Intrusion Detection Systems (NIDS), originally developed by Martin Roesch in 1998. It combines the functionalities of a packet sniffer, protocol analyser, and intrusion detection engine into a lightweight, rule-driven framework. Over the years, it has evolved into a mature and flexible tool, widely deployed in both research and production environments due to its open-source nature, large community support, and regular updates from Cisco Talos [37], [38].

At its core, Snort uses a signature-based detection approach, where predefined rules are used to match specific patterns of malicious traffic. These rules describe various aspects of packets—including source/destination IP addresses, ports, payload content, and protocol fields, to flag known attack signatures. The rules are organised using a structured format and grouped by threat category (e.g., malware, exploits, denial-of-service), allowing for systematic analysis and response. Snort's modular architecture also includes pre-processors for protocol decoding, stream reassembly, and traffic normalisation, enhancing its capability to handle complex and evasive traffic patterns [37], [38].

From a performance and deployment standpoint, Snort is valued for its portability and scalability. It can be run on multiple operating systems and integrated into network monitoring pipelines via tools such as Barnyard2, PulledPork, or SIEM platforms like Splunk. However, Snort's performance is highly dependent on hardware resources and the complexity of its rule sets. As the volume of monitored traffic increases, so does the potential for performance degradation, especially when inspecting deep packet payloads in high-speed networks. This has led to the emergence of high-performance

variants and hardware-accelerated solutions like Snort Inline and Suricata, which offer multithreading support and GPU-based enhancements [39].

While Snort is highly effective against known threats, its primary limitation lies in its inability to detect zero-day attacks or novel intrusion patterns not covered by existing signatures. This results in a reactive security model, where defence is dependent on prior knowledge of the threat landscape. To address this, researchers have experimented with combining Snort's rule-based detection with anomaly-based and machine learning approaches. For example, Gómez et al. [40] discussed the potential of hybrid IDS configurations where anomaly detection modules operate as the first layer of defence for identifying previously unseen patterns, followed by Snort operates as the second layer for detecting known attacks. However, these integrations are not native to Snort and often require external systems and extensive tuning.

### **2.3.2 Suricata**

Suricata is a high-performance, open-source Network Intrusion Detection and Prevention System (NIDPS) developed by the Open Information Security Foundation (OISF). Designed to address the limitations of traditional rule-based IDS, Suricata incorporates multi-threading, deep packet inspection (DPI), and flow-based anomaly detection, allowing it to function not only as an IDS but also as an inline IPS and network security monitoring (NSM) tool. Since its release, it has been recognised for its scalability, flexibility, and modern protocol analysis features.

A key distinction between Suricata and earlier systems like Snort lies in its native support for multi-threaded processing. By taking advantage of multi-core architectures, Suricata can process packets in parallel, enabling it to handle high-bandwidth traffic more efficiently. This architectural design has been widely acknowledged as a significant enhancement for real-time analysis in enterprise environments [39]. In performance evaluations conducted by Gupta and Sharma [41], Suricata consistently outperformed Snort in terms of throughput and detection speed under identical traffic loads.

Despite these advantages, Suricata is not without limitations. One major challenge is its resource consumption. While its performance scales well with hardware, Suricata demands significantly more memory and CPU resources than single-threaded systems, especially in environments with high connection rates or large rule sets. Gupta and Sharma [41] noted that while Suricata's architecture is ideal for large-scale networks, its deployment in constrained or embedded environments requires optimisation.

Another limitation lies in its dependence on signature-based detection for many threat types. Although Suricata supports some anomaly-based detection through statistical flow analysis, this capability remains underdeveloped compared to purpose-built machine learning IDS. Furthermore, similar to other IDS platforms, Suricata struggles with encrypted traffic analysis, relying primarily on metadata inspection and limited TLS fingerprinting, which reduces its visibility into modern HTTPS-based attacks [41].

## 2.4 Summary and Research Gaps

This chapter has presented a detailed review of the core technologies and methodologies relevant to modern Intrusion Detection Systems (IDSs), with a focus on blockchain, large language models (LLMs), artificial intelligence (AI), and widely adopted IDS tools such as Snort and Suricata. The review has also examined the integration of these technologies to enhance detection accuracy, scalability, and system trustworthiness. While many of these technologies have demonstrated strong individual capabilities, their integration into cohesive, real-world systems remains limited and underexplored.

Blockchain technology has proven to be highly effective in ensuring data integrity, decentralisation, and tamper resistance. Its cryptographic and consensus-driven mechanisms provide secure alternatives to centralised logging systems, particularly in distributed environments like IoT. However, existing implementations often suffer from latency, limited scalability, and integration overhead when applied to high-speed or real-time cybersecurity contexts. Similarly, LLMs have shown great promise in semantic analysis, threat interpretation, and contextual understanding. Yet, their real-time adoption in IDS environments is limited due to challenges such as high computational requirements, lack of explainability, and vulnerability to generating inaccurate outputs (hallucinations).

In contrast, traditional IDSs like Snort and Suricata rely on well-established rule-based detection and have maintained popularity due to their open-source nature and community support. Nevertheless, their dependency on predefined signatures makes them ineffective against unknown or evolving threats. AI-based IDSs offer a more adaptive approach through machine learning and deep learning models, but they are often hindered by false positives, unbalanced datasets, and the computational complexity of deployment. While some recent works have explored the integration of AI and blockchain to enhance IDSs, most of these focus on limited aspects such as static dataset evaluation or alert logging, without delivering a complete, real-time, and decentralised detection framework.

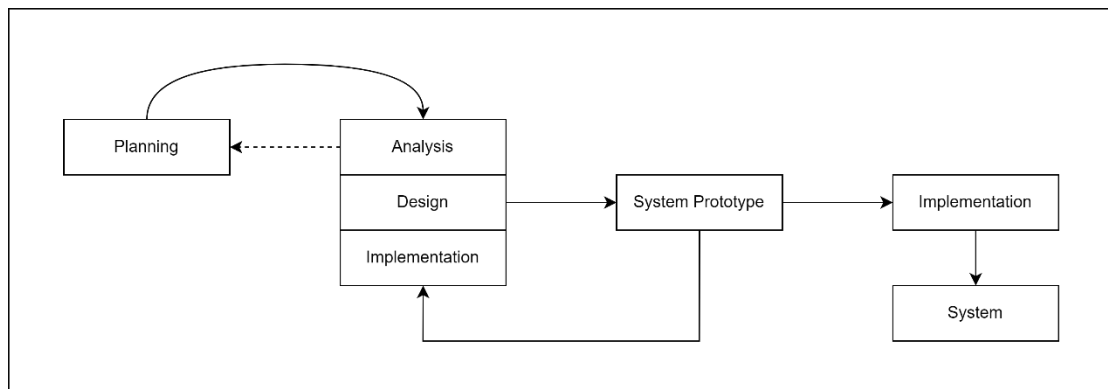
From the literature reviewed, several research gaps have been identified. First, there is a noticeable absence of IDS frameworks that incorporate large language models for real-time, flow-level traffic analysis and behavioural detection. While LLMs offer substantial reasoning and contextual capabilities, they have not been effectively embedded into operational IDS pipelines. Second, although blockchain is used in some IDSs to store alerts, it is rarely employed for full lifecycle alert management, including verification, traceability, and decentralised consensus across detection components. IN addition, blockchain-based IDSs are proposed for IoT environments, leaving the implementation of such systems in organisational networks unstudied. Third, existing systems typically focus on one detection paradigm, often signature- or anomaly-based. There is limited exploration into unified hybrid frameworks that merge rule-based detection, AI-driven anomaly identification, and LLM-based context analysis into a single platform. Finally, many of the proposed AI and blockchain-based IDS architectures are theoretical or tested only in lab environments. Real-world applicability, scalability through containerisation, and robust integration with live network traffic remain largely unaddressed.

## CHAPTER 3 SYSTEM METHODOLOGY

This chapter outlines the structured approach taken in the development of the project. It begins by presenting the system architecture and progresses through each stage of the design process. The chapter includes various diagrams such as the system architecture diagram, use case diagram, and activity diagram to visually represent the system's structure and user interactions. These visual models help clarify the logical flow of data and processes across the different modules in the system. The methodology adopted ensures that each component of the system is well-defined, interoperable, and aligned with the project's objectives. The content of this chapter lays the groundwork for the subsequent design and implementation stages.

### 3.1 Development Methodology

To develop the Blockchain-Based Intrusion Detection System with Artificial Intelligence, this project adopts the System Prototyping methodology (as shown in Figure 3.1.1), a development model under the broader Rapid Application Development (RAD) framework [42]. This approach is particularly well-suited for projects involving emerging technologies such as blockchain and LLM, where system requirements are often dynamic and integration is complex. System Prototyping emphasises the creation of an early functional model, iterative refinement, and regular stakeholder feedback, which represent qualities that are crucial in ensuring the practical success of this innovative system.



**Figure 3.1.1** System Prototyping Methodology

The main advantage of using System Prototyping in this context lies in its capacity to produce a tangible and working version of the system early in the development cycle. This allows for testing of the core modules, including real-time packet capture, signature-based and LLM-based intrusion detection, as well as blockchain-based alert logging before full-scale integration. It enables rapid identification of issues such as detection inconsistencies or performance bottlenecks, which can then be resolved incrementally.

Another reason for adopting this methodology is its support for iterative validation. Each module in the system, for instance, the LLM detection engine, blockchain logger, and frontend dashboard is initially developed independently and tested in isolation. These components are gradually combined into an integrated prototype, allowing for controlled evaluation and adjustment. This approach helps mitigate the risks of module incompatibility or unforeseen system behaviours during integration.

Furthermore, System Prototyping accommodates evolving requirements, which are expected due to the novel nature of combining blockchain and AI technologies. As the system is tested and user feedback is gathered, requirements and configurations, such as the AI model's thresholds or the blockchain's logging frequency, can be modified without requiring a complete redesign. This flexibility ensures the system remains adaptable and functional even as new security threats and detection standards emerge.

The development lifecycle within this methodology involves several overlapping and interactive phases. During the planning phase, the project objectives, scope, and deliverables are defined. The technology stack is selected based on suitability for real-time processing, scalability, and integration, this includes Python, Flask, Solidity, Web3.py, and JavaScript. The planning phase also outlines hardware and network prerequisites to support packet analysis and blockchain operations, along with a timeline and milestone plan.

In the analysis phase, detailed functional and non-functional requirements are gathered. This includes specifying the attack types to be detected, identifying the data elements to be stored on-chain, and determining performance metrics such as detection accuracy

and response time. This phase also evaluates the challenges of deploying AI and blockchain in a real-time intrusion detection setting.

The design phase follows with the creation of system architecture and module interactions. It includes the design of AI modules for traffic anomaly detection, the structure of the smart contract used for immutable alert logging, and the user interface for presenting insights. Design artefacts such as system block diagrams, use case diagrams, and data flow diagrams are produced to guide implementation.

During the implementation phase, each module is built according to its design. The signature-based detection engine, large language model (LLM) detector, blockchain logging handler, and web interface are developed as discrete components. These modules are then gradually integrated, tested, and refined in successive prototypes. Testing is performed throughout the development cycle to ensure each addition does not break existing functionalities and that all modules communicate effectively.

The prototype undergoes multiple iterations of testing and evaluation, where performance, usability, and system robustness are examined. Based on the outcomes of each cycle, the system is fine-tuned to improve efficiency, detection accuracy, and data consistency across components. Eventually, a complete and stable version is deployed. The deployment phase includes setting up the LLM engine for live monitoring, deploying the blockchain across a test network, and launching the user interface for real-time visualisation of alerts and system metrics.

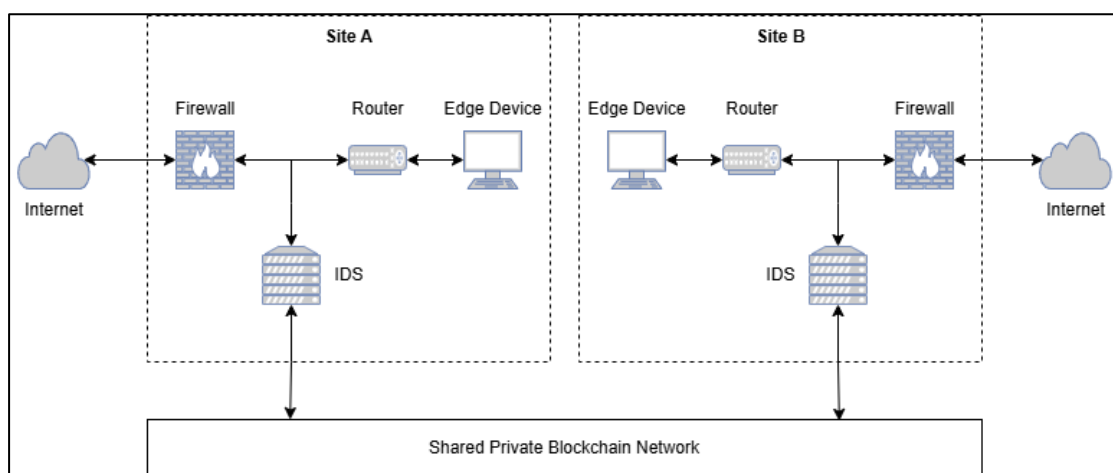
In summary, the System Prototyping methodology is a strategic choice for this project. It supports the rapid development and continuous improvement of the project in a flexible and controlled environment. This approach ensures that both the blockchain and AI technologies are implemented efficiently, tested thoroughly, and adapted dynamically to produce a functional, secure, and innovative intrusion detection system.



### 3.2 System Architecture Overview

This section presents a high-level overview of the system's architecture, illustrating how the various components interact to support the intrusion detection and alert logging processes. It includes visual models to describe the system's structure, user interactions, and workflow. The goal of this section is to provide a clear understanding of the overall system design before delving into the specific technical details in later sections.

#### 3.2.1 System Architecture Diagram



**Figure 3.2.1** System Architecture Diagram

The overall system architecture of the proposed Blockchain-Based Intrusion Detection System with Artificial Intelligence is designed to support secure, decentralised, and intelligent threat detection across multiple networked sites. As shown in Figure 3.1, the architecture is composed of two interconnected network environments, Site A and Site B, each equipped with standard network infrastructure and its own dedicated Intrusion Detection System (IDS). Both sites are connected to a shared private blockchain network, which acts as a decentralised and immutable ledger for logging security alerts. This setup is designed to simulate an organisational network distributed across multiple geographical locations, interconnected via a Wide Area Network (WAN) or Virtual Private Network (VPN).

At each site, the network is protected at the perimeter by a firewall. This component is responsible for enforcing security policies, filtering traffic, and preventing unauthorised access from external sources. Immediately after the firewall, a router manages the routing of internal traffic and connects the firewall to various edge devices or potentially a demilitarised zone (DMZ) within the local network. These edge devices may include user computers, IoT devices, or internal servers that generate and receive network traffic as part of regular operations.

The IDS is strategically positioned within the internal network, connected in such a way that it can observe and analyse network traffic without interfering with the data flow. This is typically achieved through port mirroring or the use of a network tap, enabling the IDS to operate in a passive and non-intrusive manner. The purpose of the IDS is to monitor the internal traffic continuously and identify suspicious behaviours or known attack patterns that may indicate an intrusion.

A defining feature of this architecture is the integration of a shared private blockchain network that connects the IDS nodes at both sites. This blockchain acts as a decentralised and tamper-proof logging mechanism. When either IDS detects a threat or abnormal network activity, it generates an alert and submits it to the blockchain. Each alert is stored as a transaction on the distributed ledger, ensuring that the event is permanently recorded and cannot be altered or deleted.

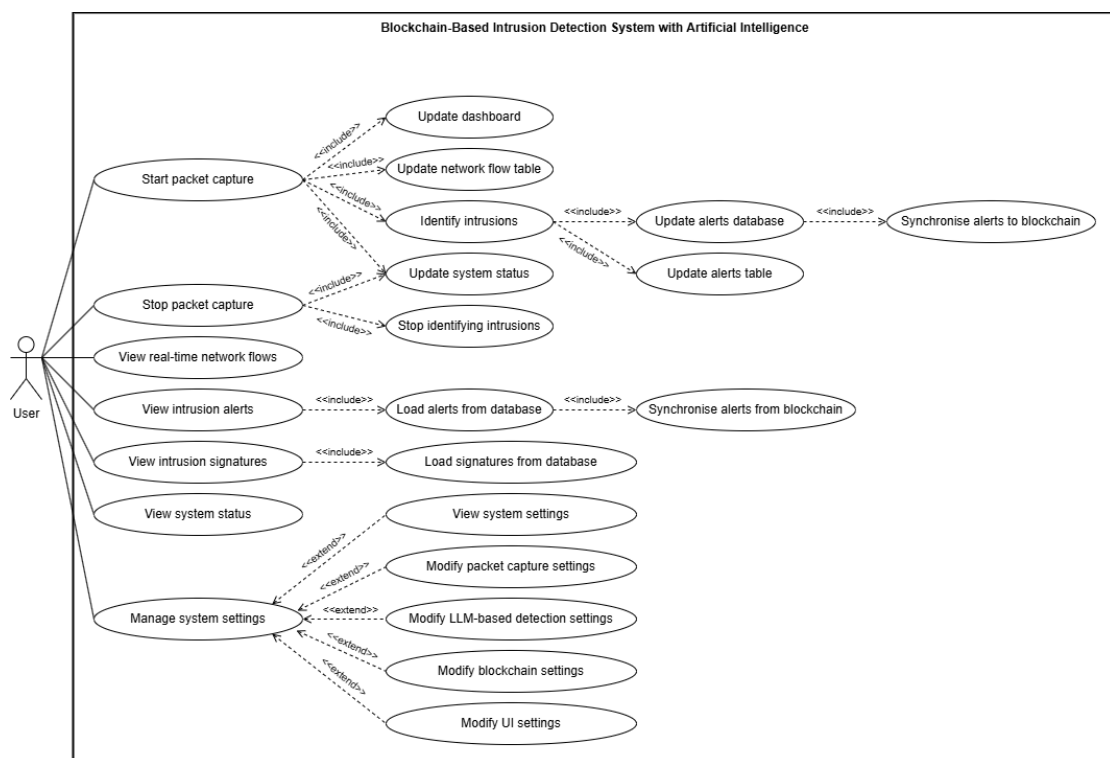
The use of blockchain in this context brings several critical advantages. Firstly, it guarantees data integrity by ensuring that recorded alerts are immutable and cryptographically verifiable. Secondly, it decentralises trust by eliminating the need for a centralised database or storage server, which could become a single point of failure or a target for attackers. Thirdly, it promotes transparency and coordination across multiple sites by providing each IDS with access to a common set of verified alerts, allowing for cross-site threat correlation and response.

The modular nature of this architecture supports scalability and adaptability. Additional sites can be integrated into the system by deploying new IDS nodes and linking them to the shared blockchain network. This ensures that the solution can grow in parallel with the organisation's infrastructure without compromising performance or security.

Moreover, the decentralised design enhances resilience by ensuring that the failure or compromise of a single node does not impact the overall system. In such environments, the unused processing capacity of internal edge devices, such as desktop machines or lightweight servers, can be utilised to support blockchain operations. These devices may be registered as blockchain nodes, contributing to transaction validation, ledger synchronisation, and distributed consensus without the need for dedicated blockchain hardware. This approach not only enhances decentralisation but also improves resource efficiency by utilising existing infrastructure.

In short, the architecture provides a robust foundation for real-time intrusion detection and verifiable alert logging in a multi-site environment. By combining traditional perimeter security with passive traffic monitoring and blockchain-based alert storage, the system ensures both operational effectiveness and long-term data integrity. The architectural design reflects a forward-thinking approach to modern cybersecurity challenges, particularly in distributed or large-scale environments.

### 3.2.2 Use Case Diagram and Description



**Figure 3.2.2** Use Case Diagram

Figure 3.2.2 illustrates the use case diagram of the Blockchain-Based Intrusion Detection System with Artificial Intelligence. It shows how the primary actor, the user, interacts with the core functionalities of the system from a high-level perspective such as initiating packet capture, stopping packet capture, viewing alerts, monitoring flows, managing configurations, monitoring system status, and view signatures. These actions are supported by various internal operations like intrusion identification, system status updates, alert handling, and blockchain synchronisation. The use case model helps to visualise both the functional scope and the modular responsibilities of each component, including how different tasks are interconnected through “include” and “extend” relationships from a use case point of view.

The following use case description tables (Table 3.2.1 to Table 3.2.7) present detailed descriptions of each use case identified in the diagram above. Each use case is documented using a standardised table.

<b>Use Case ID</b>	UC001	<b>Version</b>	1.0
<b>Use Case</b>	Start packet capture		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To initiate packet capturing on a selected network interface for analysis and intrusion detection.		
<b>Actor</b>	User		
<b>Trigger</b>	User selects an interface and clicks the “Start Capture” button.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User <b>Include:</b> Update dashboard, Update network flow table, Identify intrusions, Update system status, Update alerts database, Update alerts table, Synchronise alerts to blockchain		
<b>Precondition</b>	System must be idle and at least one network interface must be available. System must be connected to a network.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User selects a network interface from the dropdown list.	
	2	User clicks the “Start Capture” button.	
	3	System begins capturing packets on the selected interface.	
	4	System updates the network flow table with live traffic.	
	5	System updates the dashboard with real-time stats.	
	6	Intrusion detection processes (signature-based and AI-based) are started.	
	7	Detected intrusions are stored in the alerts database.	
	8	Alerts are synchronised to the blockchain.	
Alternate Flow: No interface specified	2.1	System displays an error message.	
Alternate Flow: Packet capturing already started	3.1	System notifies the user and does not reinitiate capture.	

**Table 3.2.1 UC001: Start Packet Capture**

<b>Use Case ID</b>	UC002	<b>Version</b>	1.0
<b>Use Case</b>	Stop packet capture		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To terminate the ongoing packet capture process and halt intrusion detection.		
<b>Actor</b>	User		
<b>Trigger</b>	User clicks the “Stop Capture” button.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User <b>Include:</b> Update system status, Stop identifying intrusions		
<b>Precondition</b>	Packet capture process must be currently running.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User clicks the “Stop Capture” button.	
	2	System halts packet capturing on the selected interface.	
	3	Intrusion detection processes are stopped.	
	4	System updates the dashboard and system status.	
Alternate Flow: Packet capture is idle	2.1	System displays an error message indicating no active session.	

**Table 3.2.2 UC002: Stop Packet Capture**

<b>Use Case ID</b>	UC003	<b>Version</b>	1.0
<b>Use Case</b>	View real-time network flows		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To allow the user to monitor active network flows with up-to-date statistics.		
<b>Actor</b>	User		
<b>Trigger</b>	User navigates to the “Active Flows” section in the interface.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User		
<b>Precondition</b>	Packet capture must be active, and flows must be available.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User opens the “Active Flows” tab.	
	2	System fetches the latest flow data from memory or cache.	
	3	Active network flows are displayed with source/destination, protocol, port, and statistics.	
	4	Flow table is updated continuously or at fixed intervals.	
	5	User may click “Details” on a flow to view in-depth information.	
	6	User may filter or search for specific flows.	
Alternate Flow: Packet capture is idle	2.1	System displays an empty table with a message indicating that packet capture is not started.	
Alternate Flow: No active flows	3.1	System displays an empty table with a message indicating that there are no active flows.	

**Table 3.2.3 UC003: View Real-Time Network Flows**

<b>Use Case ID</b>	UC004	<b>Version</b>	1.0
<b>Use Case</b>	View intrusion alerts		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To enable the user to review alerts generated by the intrusion detection engines.		
<b>Actor</b>	User		
<b>Trigger</b>	User navigates to the “Alerts” section in the interface, or to the dashboard in the interface.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User <b>Include:</b> Load alerts from database, Synchronise alerts from blockchain		
<b>Precondition</b>	System must be started and accessed via a browser.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User accesses the “Alerts” tab or the alerts table in the dashboard.	
	2	System loads alerts from the local alert database.	
	3	System synchronises additional alerts from the blockchain.	
	4	Alerts are displayed with severity level, timestamp, source/destination, and detection type.	
	5	User may filter, search, or sort alerts as needed.	
Alternate Flow: Empty database	2.1.1	System displays a message, indicating there are no alerts generated.	
Alternate Flow:	2.2.1	System creates a database directory and a database file.	
Missing database	2.2.2	System displays a message, indicating there are no alerts generated.	
Alternate Flow: Failed blockchain connection	3.1	System shows only the local alerts with an error message indicating the failed connection to blockchain.	

**Table 3.2.4 UC004: View Intrusion Alerts**



<b>Use Case ID</b>	UC005	<b>Version</b>	1.0
<b>Use Case</b>	View intrusion signatures		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To allow the user to view the list of loaded intrusion detection signatures.		
<b>Actor</b>	User		
<b>Trigger</b>	User selects the “Intrusion Signatures” section from the dashboard.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User <b>Include:</b> Load signatures from database		
<b>Precondition</b>	Signature file must be successfully loaded during system initialisation.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User navigates to the “Intrusion Signatures” tab.	
	2	System loads the list of signatures from memory or from the local database.	
	3	The signatures are displayed, including fields like name, pattern, protocol, and detection criteria.	
	4	User can filter or search for specific signatures.	
Alternate Flow: Signature file corrupts or missing	2.1	System displays an error message.	
Alternate Flow: Empty signature file	3.1	System displays a message indicating no signatures are available.	

**Table 3.2.5 UC005: View Intrusion Signatures**

<b>Use Case ID</b>	UC006	<b>Version</b>	1.0
<b>Use Case</b>	View system status		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To allow the user to monitor the current operational state of the system components.		
<b>Actor</b>	User		
<b>Trigger</b>	User opens the system status section in the interface.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User		
<b>Precondition</b>	The backend services must be running and able to return status data.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User accesses the “System Status” tab.	
	2	System fetches the status of packet capture, LLM detection engine, blockchain, and system resources.	
	3	Real-time statistics for each component are displayed.	
	4	System health indicators (e.g. running/stopped, connected/disconnected) are updated periodically.	
Alternate Flow: System fails	2.1	System displays an error message.	

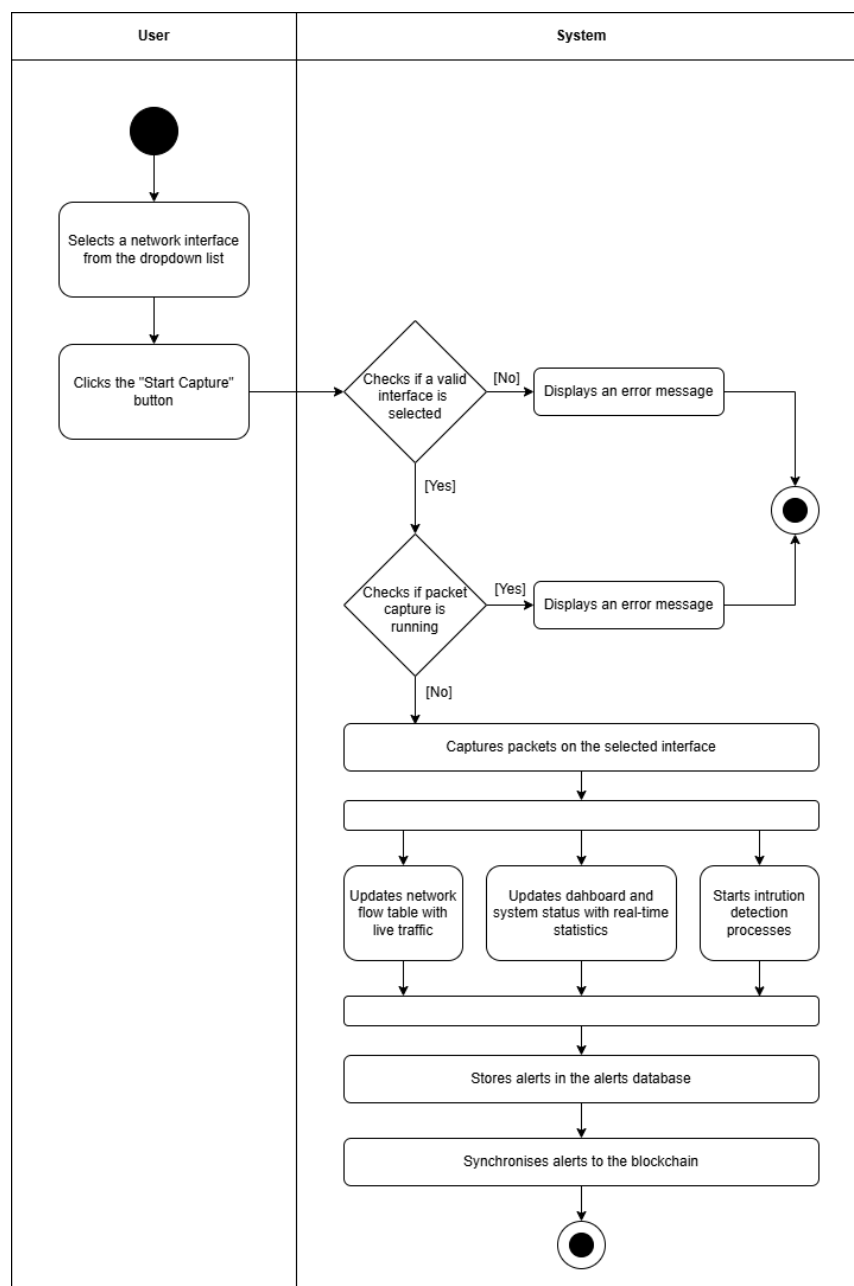
*Table 3.2.6 UC006: View System Status*

<b>Use Case ID</b>	UC007	<b>Version</b>	1.0
<b>Use Case</b>	Manage system settings		
<b>Use Case Type</b>	Primary		
<b>Stakeholder</b>	User		
<b>Purpose</b>	To allow the user to configure and update system parameters for detection, AI models, and blockchain.		
<b>Actor</b>	User		
<b>Trigger</b>	User opens the settings or configuration panel in the web interface.		
<b>Trigger Type</b>	External		
<b>Relationship</b>	<b>Association:</b> User <b>Extend:</b> View system settings, Modify packet capture settings, Modify LLM-based detection settings, Modify blockchain settings, Modify UI settings		
<b>Precondition</b>	System must be running and accessed from a browser.		
<b>Scenario Name</b>	<b>Step</b>	<b>Action</b>	
Main Flow	1	User navigates to the “Settings” tab.	
	2	User adjusts available options, such as: Network interface, BPF filter, Packet log file, LLM model, LLM analysis batch size, Start/stop LLM, Test LLM, Blockchain URL, Contract address, Sync Interval, Force sync, Theme, Data refresh interval.	
	3	System validates and applies new settings.	
	4	System confirms changes with a success message.	
Alternate Flow: Invalid input	3.1	System displays an error message and does not apply the change.	

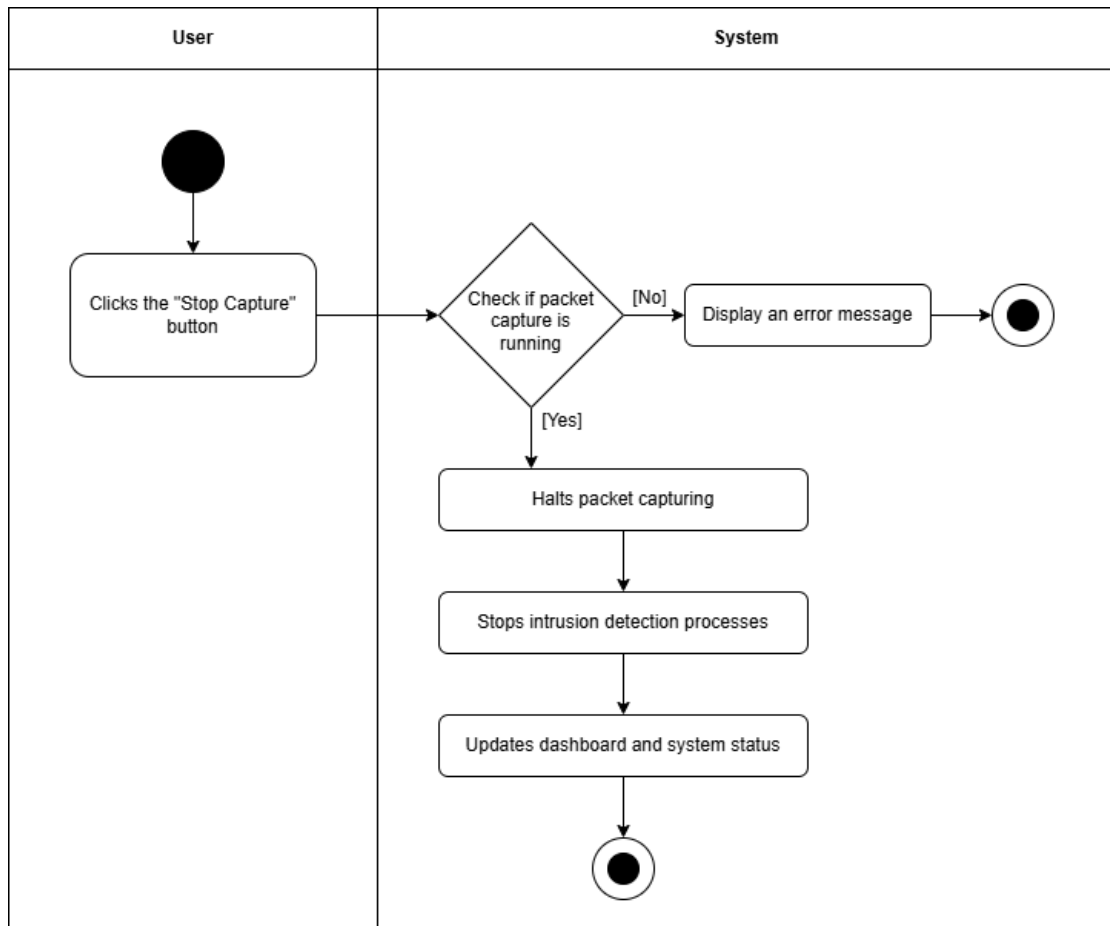
*Table 3.2.7 UC007: Manage System Settings*

### 3.2.3 Activity Diagram

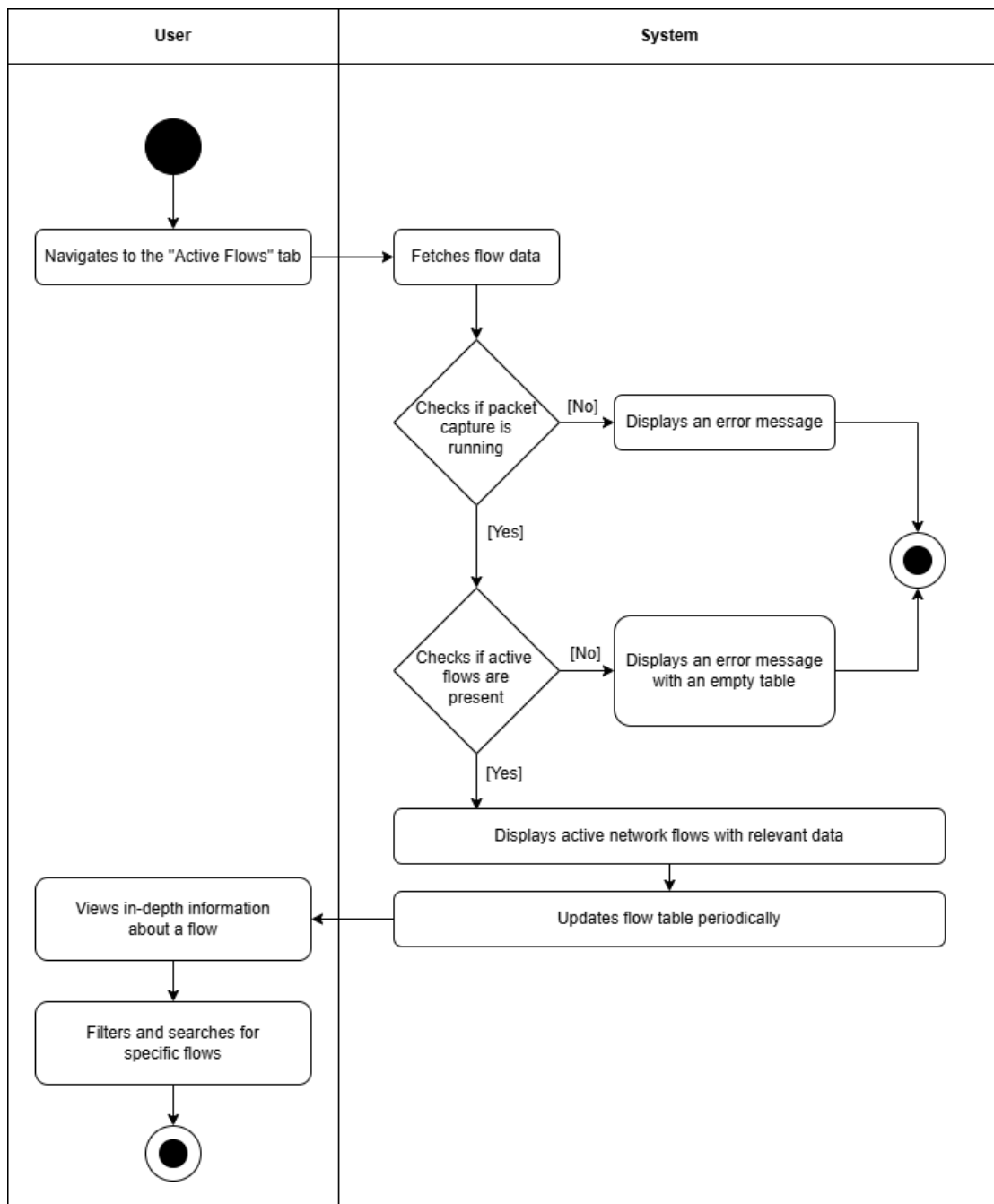
This section presents the activity diagrams for selected use cases to visualise the flow of actions within the system. Each diagram (Figure 3.2.3 to Figure 3.2.9) illustrates the step-by-step process involved in carrying out a use case, highlighting the decision points, actions performed by the user, and system responses, as detailed in the use case descriptions in the previous section. These diagrams help to clarify the logic and sequence of operations within the system.



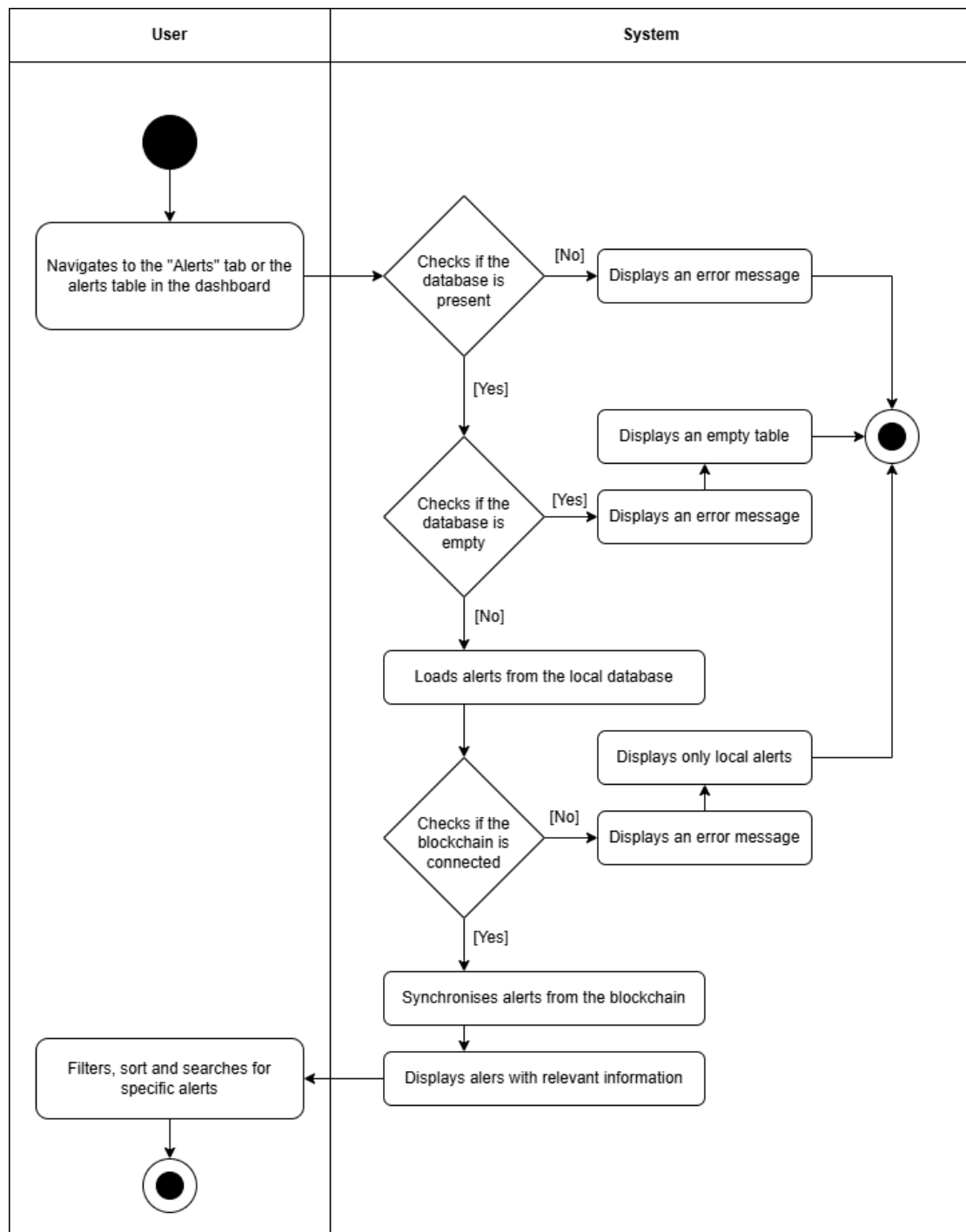
**Figure 3.2.3** Activity Diagram: Start Packet Capture



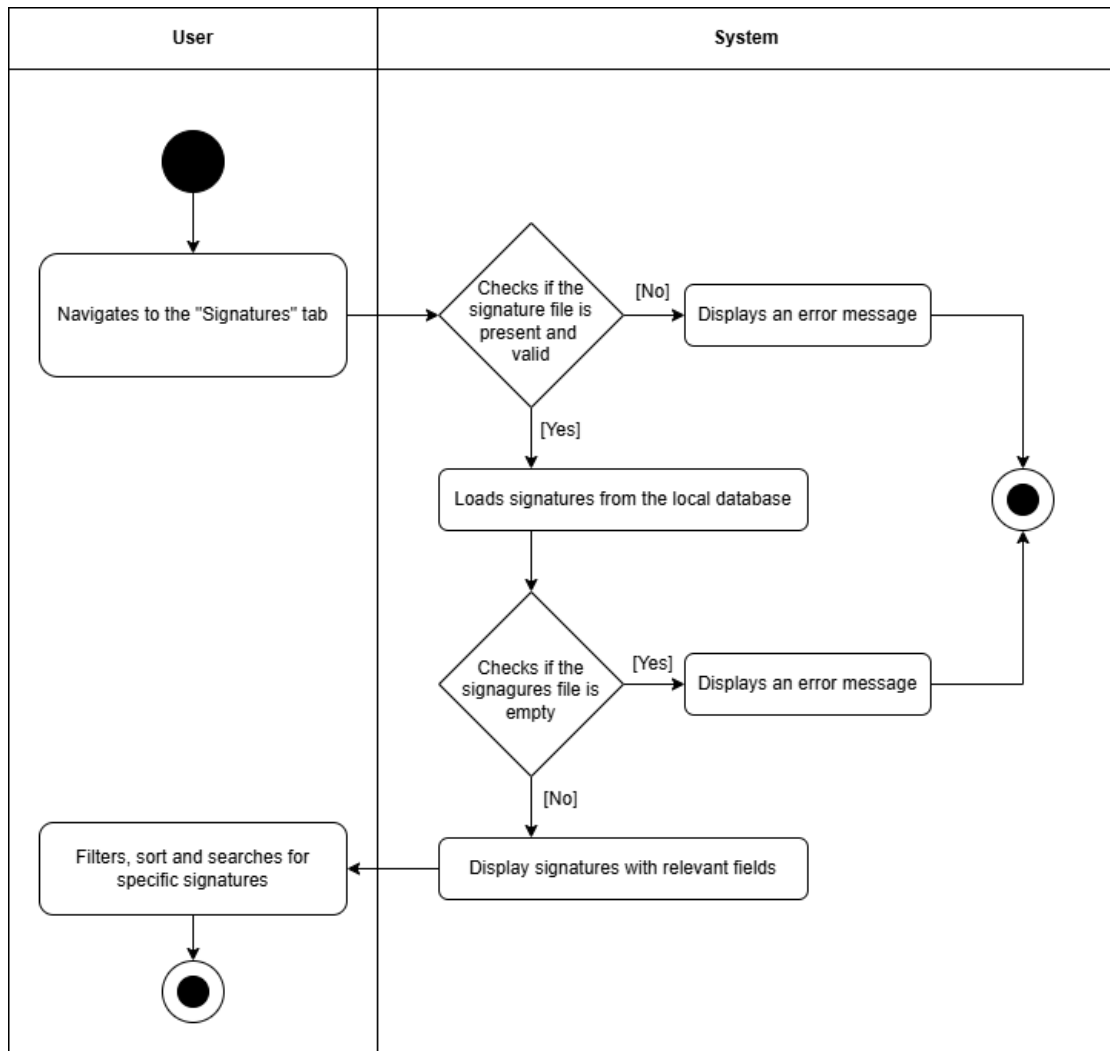
**Figure 3.2.4** Activity Diagram: Stop Packet Capture



**Figure 3.2.5** Activity Diagram: View Real-Time Network Flows

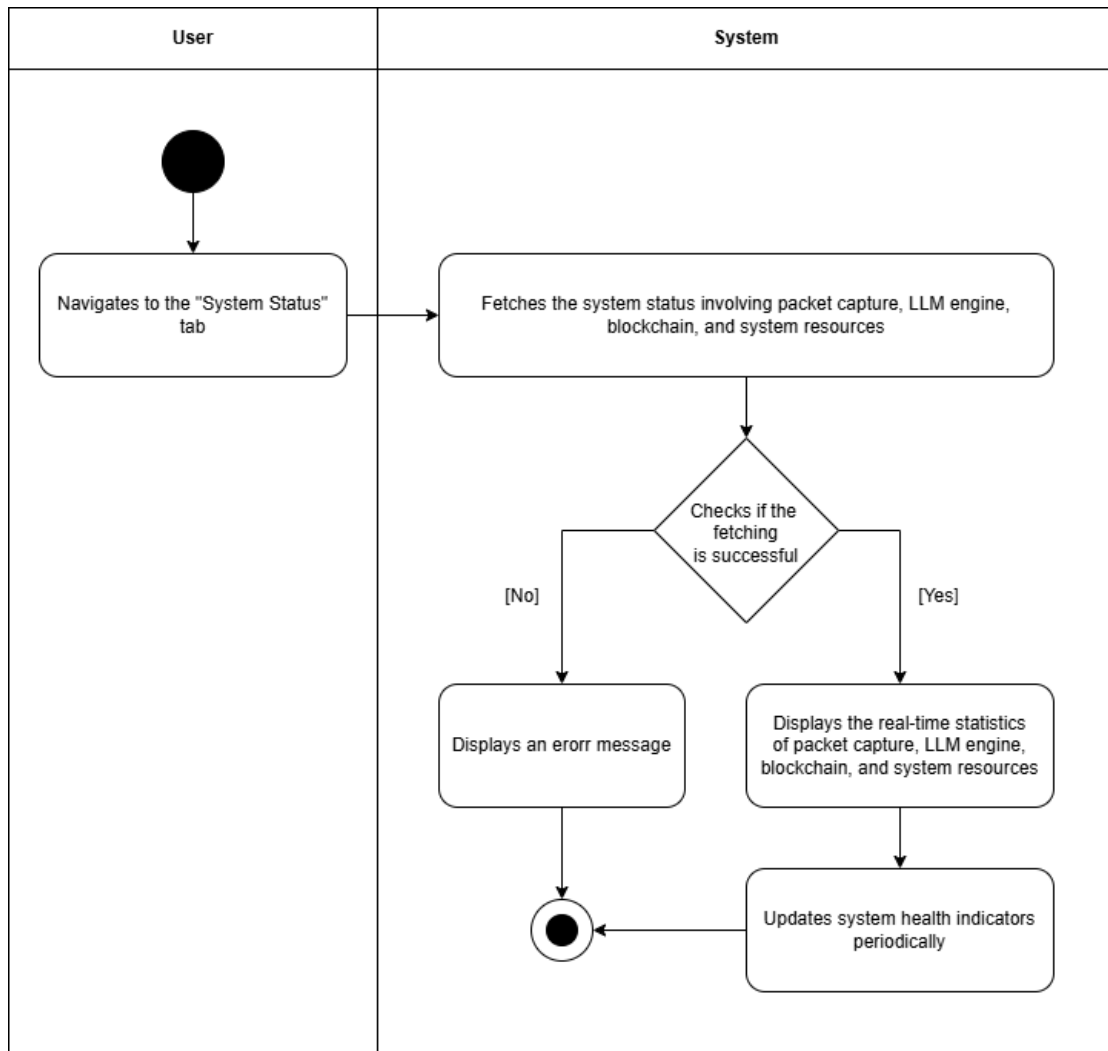


**Figure 3.2.6** Activity Diagram: View Intrusion Alerts

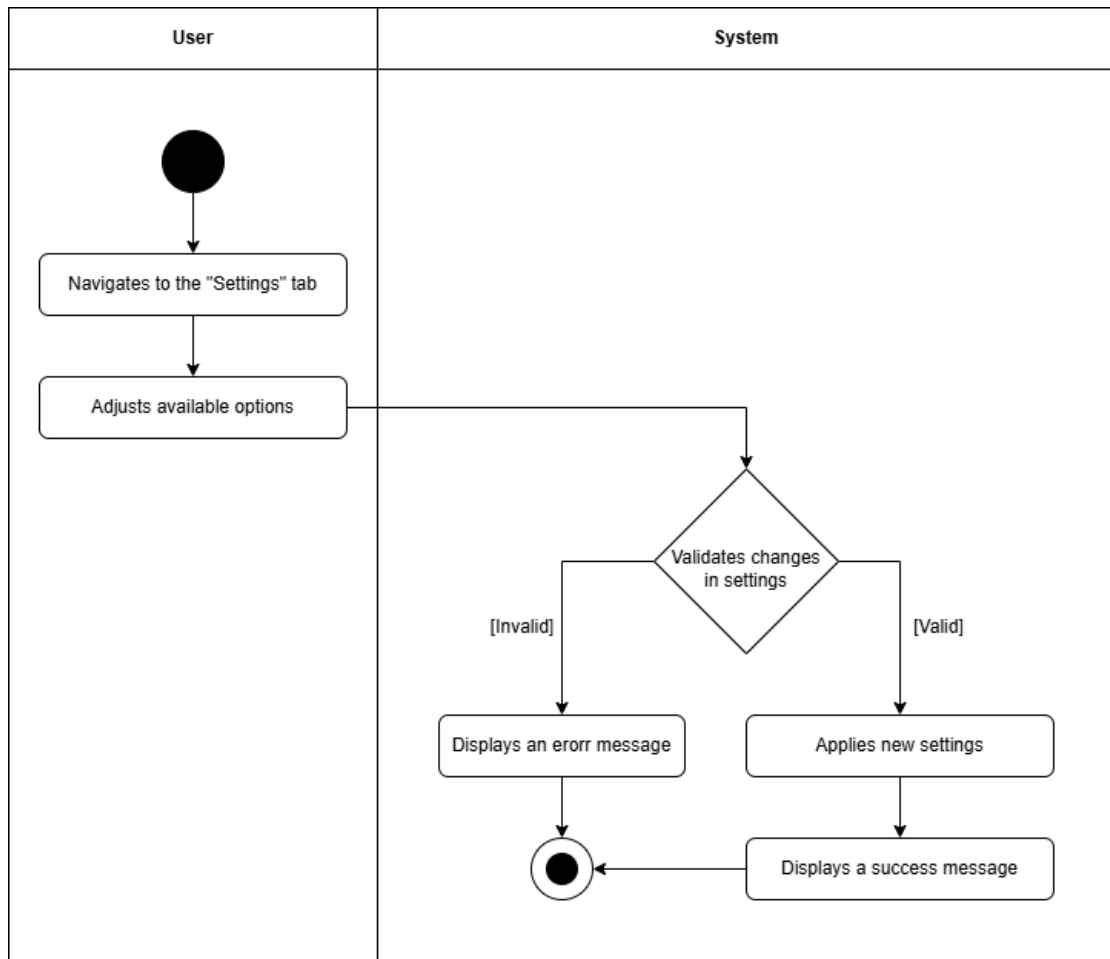


**Figure 3.2.7** Activity Diagram: View Intrusion Signatures





**Figure 3.2.8** Activity Diagram: View System Status



**Figure 3.2.9** Activity Diagram: View System Settings

### 3.3 Design Specifications

This section defines the essential technical and operational requirements needed to develop and deploy the system effectively. It outlines the software and hardware components necessary to support system functionality. The section also details the functional and non-functional requirements that describe what the system must do and how it should perform under various conditions. Additionally, it highlights any design constraints that influence the implementation, such as technical limitations, compatibility concerns, or performance boundaries.

#### 3.3.1 Software Requirements

The software requirements were carefully selected to support the functionality of real-time packet capture, flow analysis, signature-based and AI-based intrusion detection, blockchain logging, and a responsive web-based user interface. This section focuses on the conceptual and development-time software dependencies. Table 3.3.1 summarises the software requirements for the project.

Component	Purpose	Type
<b>Programming Languages</b>		
Python 3.10+	Main programming language for backend services and detection engines	Backend Technologies
HTML, CSS, JavaScript	Structure and behaviour of the web-based frontend	Frontend Technologies
Solidity	Language used to develop the smart contract	Smart Contract Language
<b>Frameworks and Libraries</b>		
Flask	Lightweight web server framework for serving API endpoints	Python Framework
scapy	Packet manipulation and network traffic parsing.	Python Library
threading, queue, time	Standard libraries for concurrency and packet processing queues	Python Built-in Modules
dotenv	Environment configuration management	Python Utility Module
Web3.py	Interacts with Ethereum blockchain to store alerts	Blockchain SDK
Node.js + npm	Supports frontend asset bundling and library management	JavaScript Runtime

Chart.js / Recharts	Visualisation of alert distribution and network activity	JavaScript Library
FontAwesome	Icons for user interface design	UI Library
Requests	Python HTTP library used for REST API communication	Python Library
psutil	To monitor system resource usage such as CPU and memory	Python Library
<b>LLM Software</b>		
Ollama	A local LLM model runtime for serving and managing Gemma 3B:1 with minimal resource overhead	LLM Hosting Runtime
Gemma3:1b	A 1-billion parameter language model used to analyse and classify network flows	AI Detection Model
<b>Blockchain Tools</b>		
Ganache	Local Ethereum blockchain emulator for testing blockchain transactions	Blockchain Emulator
<b>Database / Storage</b>		
SQLite	Lightweight database for temporary alert storage before blockchain sync	Local Database
JSON	Data format for configuration files and inter-process communication	Data Format
<b>Development Tools</b>		
Git	Version control system for collaborative development	Version Control Tool
VS Code / PyCharm	IDEs used to write and manage Python and JavaScript code	Development Environment
Chrome	Web-browser for viewing and testing the frontend dashboard	Testing Tool / UI Layer
<b>OS and Runtime</b>		
Windows 11 (64-bit)	Operating system used to host this system	Operating System

**Table 3.3.1** Software Requirements

The project is developed and tested on Windows 11 (64-bit). All software components, including Python, Flask, Scapy, and Ollama, are compatible with the Windows environment. However, certain libraries such as scapy and psutil may require elevated permissions or administrative access to interact with network interfaces and system-level resources.

To ensure smooth operation of the system on Windows 11 (64-bit), it is recommended to run Windows PowerShell or Command Prompt with administrator privileges when performing packet capture. Python virtual environments should be used to avoid conflicts with system packages. The Ollama runtime and the Gemma 3B:1 model can be executed locally, provided the system has at least 8GB of RAM and preferably a dedicated GPU such as an NVIDIA card to accelerate inference. Ganache must be installed and run either as a standalone application or through the command line to emulate the local Ethereum blockchain.

### **3.3.2 Hardware Requirements**

This section outlines the essential hardware specifications needed to support the development, execution, and evaluation of the project. The hardware requirements here refer to the minimum and recommended capabilities of a development machine or deployment server. They are identified based on the system's need to perform real-time packet capture, process network flow data, analyse traffic using a local LLM, and synchronise alerts with the blockchain. Table 3.3.2 summarises the hardware requirements for this project.

Component	Minimum Requirement	Recommended Specification	Purpose
Processor (CPU)	Intel Core i5 (4 cores)	Intel Core i7 / AMD Ryzen 7 (6+ cores)	Handles flow analysis, API requests, LLM pre-processing
Memory (RAM)	8 GB	16 GB or more	Required for LLM inference, blockchain sync, and buffers
Storage	256 GB SSD	512 GB SSD or more	Stores packet logs, alerts, contract data, and models
Graphics (GPU)	Integrated GPU	NVIDIA GPU (4 GB VRAM or more, e.g. GTX 1650 or better)	Accelerates LLM inference for faster flow analysis
Network Interface Card	Standard Ethernet or Wi-Fi (Monitor Mode optional)	Gigabit Ethernet or USB NIC with monitor mode support	Captures live traffic and supports flow-level inspection
Display & Peripherals	Basic HD display, mouse, keyboard	Dual monitor setup for parallel monitoring	Supports debugging, dashboard interaction
Power Supply	65W	90W+	Ensures stable power during model inference or capture
Cooling System	Standard	Enhanced cooling (especially if using GPU inference)	Prevents overheating during sustained processing loads

**Table 3.3.2 Hardware Requirements**

These hardware resources are critical to ensuring that the system functions efficiently. Real-time packet capture must be continuous and lossless to avoid missing malicious activity. Signature-based and AI-based detections must operate concurrently to detect known and novel threats in real time. Additionally, blockchain logging of alerts must occur promptly to maintain the integrity and traceability of security events. Lastly, the frontend dashboard should remain responsive, enabling seamless interaction for monitoring and administrative tasks.

Among all components, CPU, RAM, and disk I/O performance have the most direct impact on real-time detection and data processing. The inclusion of a dedicated GPU is especially important for accelerating LLM inference, particularly when running a model like Gemma3:1b locally, as it significantly reduces analysis latency.

Although the system can run on lower-end hardware by reducing packet processing rates or disabling GPU inference, this compromises responsiveness and detection speed. Therefore, for development, evaluation, and production-like testing, the use of the recommended hardware specifications is strongly encouraged.

### 3.3.3 Functional Requirements

Functional requirements for this project define the essential capabilities that the system must possess to effectively detect and respond to security threats. These requirements outline the specific actions the system must perform, hence ensuring the system meets its core objectives. The functional requirements of this system are outlined in Table 3.3.3 according to the requirement category.

Category	Requirement ID	Requirement Description
Packet Capture	FR1	The system shall capture live network packets from a specified interface.
	FR2	The system shall allow users to select a network interface and apply optional filters.
Flow Analysis	FR3	The system shall process captured packets into network flows.
	FR4	The system shall compute flow statistics including packet count, byte size, and duration.
Signature Detection	FR5	The system shall match flows against predefined attack signatures stored in a JSON file.
	FR6	The system shall generate alerts when a flow matches a known signature.
AI Detection (LLM)	FR7	The system shall analyse suspicious flows using a local LLM hosted via Ollama.
	FR8	The system shall generate AI-based alerts with severity levels based on LLM output.

Blockchain Logging	FR9	The system shall log alerts to the Ethereum blockchain via a smart contract.
	FR10	The system shall synchronise unsent alerts in batches to the blockchain.
Web Dashboard	FR11	The system shall provide a web interface to view active flows, alerts, and system status.
	FR12	The system shall allow users to view detailed information for each alert and flow.
	FR13	The system shall provide controls to start and stop packet capture.
System Status & Control	FR14	The system shall display real-time statistics such as packet rate, flow count, and alerts.

*Table 3.3.3 Functional Requirements*

### 3.3.4 Non-Functional Requirements

Non-functional requirements define the quality attributes and operational constraints of the project. Non-functional requirements describe how the system should perform under various conditions. These include performance expectations, reliability, usability, scalability, and security standards that ensure the system remains effective, efficient, and user-friendly throughout its lifecycle. The non-functional requirements of this system are outlined in Table 3.3.3 according to the requirement category.



Category	Requirement ID	Requirement Description
Performance	NFR1	The system shall process and analyse network packets in real time with minimal latency of not more than 5 seconds.
	NFR2	The system shall support concurrent detection using both signature and AI-based engines.
	NFR3	The system shall capture and process packets efficiently with a maximum 2% rate of dropping packets.
	NFR4	The system shall initialise all components and become ready for use within 10 seconds.
	NFR5	The dashboard shall update displayed data (flows, alerts, stats) every 1–5 seconds in real time.
	NFR6	The system shall batch and synchronise alerts to the blockchain every 60 seconds or less.
Usability	NFR7	The system shall provide a user-friendly web interface with clear visual indicators.
	NFR8	The system shall display meaningful messages for errors and alerts.
Maintainability	NFR9	The system shall match flows against predefined attack signatures stored in a JSON file.
Accuracy	NFR10	The system shall have a detection accuracy of at least 90%.
	NFR11	The system should have a false positive rate of less than or equals 5%.

***Table 3.3.4 Non-Functional Requirements***

### 3.3.5 Design Constraints

Before the development of the project, several design-stage constraints shaped how the system could be conceptualised, structured, and planned. These constraints influenced architectural decisions, technology choices, and the balance between functional requirements and practical feasibility. This section outlines the key limitations identified during the design phase.

Firstly, the requirement for near real-time intrusion detection imposed a constraint on the design of data flow and processing mechanisms. The system had to be capable of handling large volumes of traffic quickly, which ruled out complex or computationally

expensive preprocessing during initial capture. Therefore, the architecture was designed to prioritise lightweight filtering and staged analysis.

To ensure maintainability and future scalability, a modular design was preferred. However, modularity often introduces performance overhead, particularly in inter-process communication and data handoff between detection components. This required early design compromises. Some tightly coupled functions (e.g. packet parsing and flow analysis) were grouped within the same module to reduce latency.

The design of the user interface was bound by the need to display dynamic data such as packet counts, flow tables, and alert logs. This required a frontend architecture that could refresh frequently without degrading browser performance. It led to the decision to use polling-based updates (via JavaScript fetch APIs) rather than real-time websockets, which simplified frontend design at the cost of minor update delay.

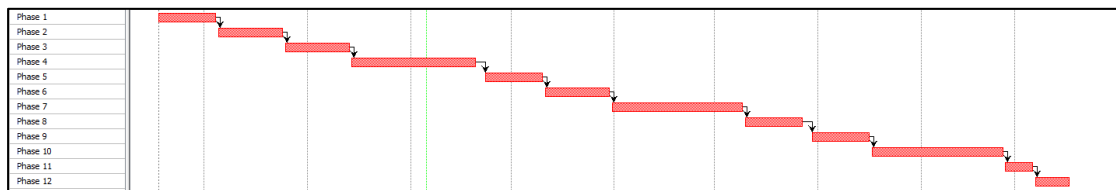
Because the system was expected to run on standard user machines or student lab computers, design assumptions had to account for limited CPU, RAM, and storage. This meant avoiding heavyweight detection frameworks, such as full deep learning models or stream processing engines. Lightweight, rule-based filtering and statistical detection methods were prioritised in the early architecture.

Furthermore, the system required access to network interfaces for packet capture, which is restricted on some operating systems without administrator or root privileges. This constraint influenced the design by limiting the supported platforms and requiring fallbacks in case access to packet capture was denied.

Artificial Intelligence components were intended to detect complex or evolving threats, but during the design phase, the availability of realistic and diverse network traffic datasets was identified as a major constraint. This limited the ability to design and fine-tune models from scratch. As a result, the LLM integration was scoped to use zero-shot or few-shot detection techniques rather than fully fine-tuned models.

### 3.4 Development Timeline

The timeline for the “Blockchain-Based Intrusion Detection System with Artificial Intelligence” is structured to ensure a systematic and efficient development process, following the system prototyping methodology. This approach allows for iterative feedback and refinement, ensuring that each phase of the project aligns with user requirements and technical objectives over the span of 28 weeks (2 trimesters).



**Figure 3.4.1** Project Development Timeline

Figure 3.5.1 shows the Gantt Chart outlining the project development timeline. The development process is broken down into 12 phases, each with its distinct activities and milestones to achieve. The phased breakdown of the development plan is shown in Table 3.5.1 below.

Phase	Description	Duration
Phase 1: Planning	Identify user requirements, define project scope, gather initial functional and non-functional requirements.	Week 1 to Week 2
Phase 2: Analysis	Perform analysis on the existing systems and establish project foundations.	Week 3 to Week 4
Phase 3: First Prototype Design	Design basic system architecture, draft initial diagrams, and outline core components.	Week 5 to Week 6
Phase 4: First Prototype Implementation	Develop basic versions of key components, including the data preprocessing for datasets, initial AI model training, network traffic capturing, and blockchain network configuration.	Week 7 to Week 10
Phase 5: Testing and Refinement	Conduct initial tests on the first prototype. Refine requirements based on the testing results.	Week 11 to Week 12

Phase 6: Second Prototype Design	Design detailed system architecture and refine the system diagrams based on the testing results and the refined requirements.	Week 13 to Week 14
Phase 7: Second Prototype Implementation	Develop enhanced versions of key components. Optimise the performance of the components and fix issues.	Week 15 to Week 18
Phase 8: Testing and Refinement	Conduct unit, integration, performance and user acceptance tests to evaluate the prototype. Refine the scope and requirements based on the results.	Week 19 to Week 20
Phase 9: Final Prototype Design	Design the final versions of the modules with fully functional components.	Week 21 to Week 22
Phase 10: Final Prototype Implementation	Develop final versions of the modules with LLM, enhanced blockchain logging, and full functionality for real-time alerts and response automation.	Week 23 to Week 26
Phase 11: Final Testing and Refinement	Conduct unit, integration, performance and user acceptance tests to evaluate the performance metrics of the prototype. Fix final issues if they were to emerge.	Week 27
Phase 12: Deployment	Deploy the system and ensure the system is able to function optimally.	Week 28

***Table 3.4.1 Phased Breakdown of the Development Plan***

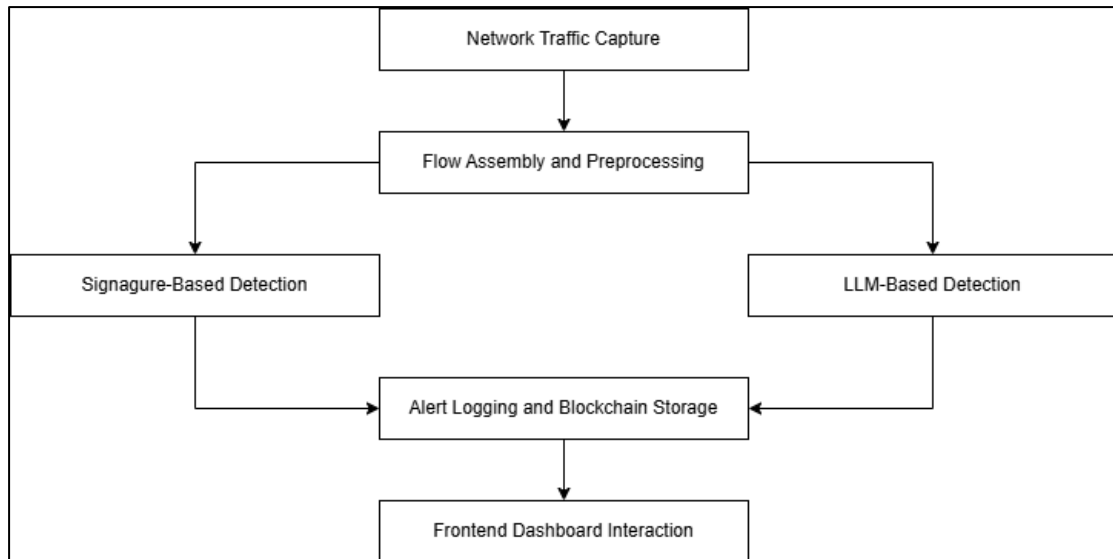
## **CHAPTER 4 SYSTEM DESIGN**

### **4.1 System Architecture**

This chapter presents the detailed design of the proposed system. It begins by outlining the overall system architecture, including the high-level system flow, block diagram, and data flow representation. The chapter then breaks down the design into individual components, covering their interactions, internal logic, and how they contribute to the system as a whole. It also discusses the structure of the database and storage mechanisms, the design of the smart contract used for alert logging on the blockchain, and the interfaces facilitating communication between modules. Additionally, the chapter describes the compilation and setup process, including required tools, dependencies, and project configuration, ensuring the system can be reliably built and deployed.

#### **4.1.1 High-Level System Flow**

The high-level system flow illustrates the sequential process by which the Blockchain-Based Intrusion Detection System with Artificial Intelligence captures, analyses, and responds to network traffic. This flow provides an overview of how data moves through various subsystems, from initial packet capture to final alert storage on the blockchain. Figure 4.1.1 outlines the high-level system flow for this project.



**Figure 4.1.1** High Level System Flow

The process begins with network traffic capture, where the system listens to a selected network interface using the scapy packet manipulation library. Users can define optional Berkeley Packet Filter (BPF) rules to refine the captured data. Incoming packets are collected and added to a queue for subsequent processing, ensuring they are not lost even under high traffic volume.

Next, the packets are processed through flow assembly and preprocessing. At this stage, packets are grouped into flows based on attributes such as source and destination IP addresses, ports, and protocol types. The system calculates important statistical features for each flow, including packet count, byte size, duration, and TCP flag patterns. These features are necessary for both signature-based and AI-based threat evaluation.

The signature-based detection engine inspects each flow against a library of predefined attack signatures stored in a structured JSON file. If a match is detected, an alert is generated and queued for blockchain logging. This method is effective for identifying known threats with well-defined patterns.

For threats that do not match any known signature but still appear anomalous, the system employs AI-based detection using a local Large Language Model (LLM). Suspicious flows are passed to the Gemma3:1b model via the Ollama runtime. The LLM performs zero-shot classification and returns an output indicating whether the

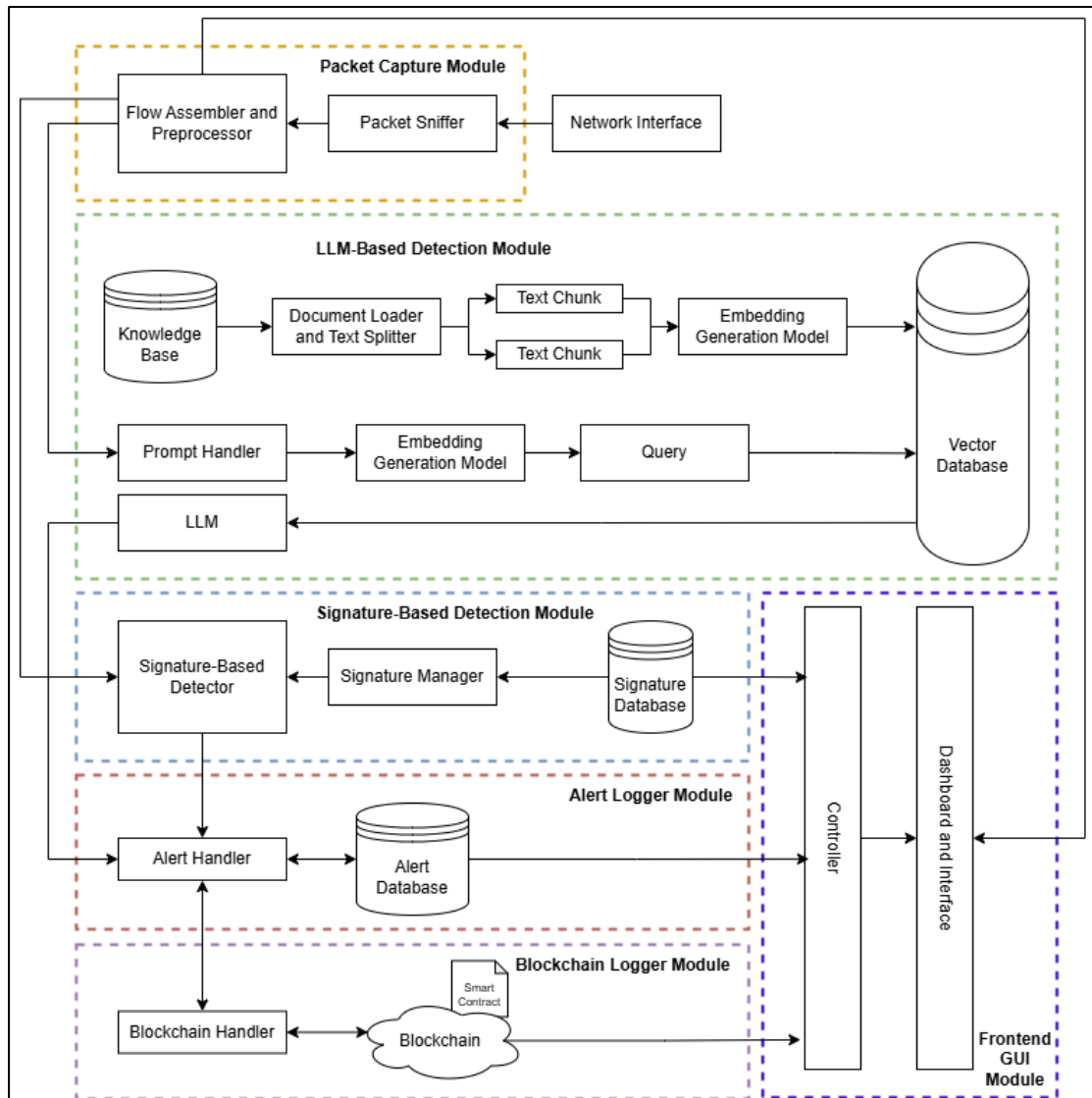
flow is benign or malicious, along with a severity rating. AI-generated alerts are also added to the system's alert history.

Following detection, alerts, whether signature-based or LLM-derived, are handled by the alert logging and blockchain storage module. Alerts are first stored temporarily in a local SQLite database. At defined intervals or in real-time (as defined by the user), they are batched and sent to the Ethereum blockchain (emulated by Ganache) using Web3.py. This ensures that alerts are securely stored, time-stamped, and tamper-proof.

Finally, the frontend dashboard provides users with a real-time interface to monitor system activity. The dashboard displays live traffic flows, alerts, system status, and blockchain synchronisation logs. Users can also control packet capture, view detailed flow and alert information, and toggle both the AI engine and blockchain logger as needed.

#### **4.1.2 System Block Diagram and Data Flow**

The system block diagram provides a high-level overview of the core components in the Blockchain-Based Intrusion Detection System with Artificial Intelligence and illustrates how data flows between these components. However, it presents a more detailed view of the entire system as compared to the high-level system flow in the previous section. In this diagram, each block represents a modular subsystem responsible for a specific task in the process of intrusion detection and alert management. The system block diagram for this project is depicted in Figure 4.1.2 below.



**Figure 4.1.2** System Block Diagram

### Packet Capture Module

The process begins at the Packet Capture Module, which is responsible for real-time monitoring of network traffic. The Network Interface listens to incoming and outgoing packets on the system. These packets are intercepted by the Packet Sniffer, which captures raw packet data. The data is then processed by the Flow Assembler and Preprocessor, where packets are grouped into flows based on common attributes (such as IP addresses, ports, and protocols). These flows are enriched with statistical features and metadata before being forwarded simultaneously to both the LLM-Based Detection Module and the Signature-Based Detection Module which run on separate threads for optimisation.



### **LLM-Based Detection Module**

The flows from the packet capture module are ingested into the LLM-based detection engine, which implements a Retrieval-Augmented Generation (RAG) framework. This module performs semantic analysis using AI. Initially, a Knowledge Base provides context, consisting of network security documentation, threat reports, and other domain knowledge. This content is divided using a Document Loader and Text Splitter, then encoded into numerical vectors using an Embedding Generation Model. These vectors are stored in a Vector Database for fast retrieval.

When a new flow arrives, it is converted into a structured query by the Prompt Handler. This query is used to retrieve relevant knowledge chunks from the vector database. These retrieved chunks, combined with the live flow data, are passed to the Large Language Model (LLM). The LLM evaluates the flow in context and decides if it is malicious. If an anomaly is detected, the LLM generates an alert, which is passed to the Alert Logger Module for further processing.

### **Signature-Based Detection Module**

In parallel with the LLM analysis, the Signature-Based Detection Module applies a deterministic approach. The module receives the same pre-processed flow from the Packet Capture Module. It checks this data against a Signature Database using the Signature-Based Detector. The Signature Manager maintains the database by handling retrieval operations for attack patterns and updating the system with new threat signatures. If a match is found, an alert is immediately generated and passed to the Alert Logger Module.

### **Alert Logger Module**

Both detection modules feed alerts into the Alert Logger Module, where the Alert Handler standardises the alert data. Each alert is enriched with metadata such as timestamp, source, severity level, detection method (LLM or signature), and relevant flow details. The processed alerts are saved in the Alert Database. This ensures persistence, supports retrospective analysis, and facilitates interaction with both the blockchain logger and the frontend interface.

### **Blockchain Logger Module**

To ensure integrity and auditability, alerts are further processed by the Blockchain Logger Module. The Alert Handler forwards validated alerts to the Blockchain Handler, which transforms alert metadata into a structured format suitable for on-chain storage. A Smart Contract deployed on the blockchain receives the alert hash and relevant metadata. This ensures that the detection record is immutable and verifiable. This module periodically synchronises with the local database, batching alerts and updating on-chain state to reduce cost and congestion.

### **Frontend GUI Module**

Finally, the Frontend GUI Module serves as the main interface for system users. It connects to all backend modules through a central Controller and displays data using the Dashboard and Interface. Users can monitor:

- Real-time packet capture status.
- Ongoing flow analysis and statistics.
- Alerts from both LLM and signature detectors.
- All loaded signatures.
- System health and status
- Blockchain sync state and transactions.

Users can also interact with the system by toggling detection engines, adjusting capture filters, syncing alerts to blockchain manually, and viewing flow-level detail. The interface promotes transparency, usability, and control in a single web-based dashboard, accessible by any authorised node within the network due to its hosted backend services.

## **4.2 Component-Level Design**

This section provides a detailed breakdown of each major component within the system, describing their specific roles, internal structures, and interactions with other modules. It explains how the components are designed to perform their tasks from a technical and low-level point of view, as compared to the previous section. This section also outlines the design principles behind each module.

### **4.2.1 Packet Capture Module**

#### **Overview**

The Packet Capture Module is the foundational component responsible for acquiring real-time network traffic data, transforming it into structured flows, and forwarding it for threat analysis. Implemented in Python using the Scapy library, this module features multithreaded execution, configurable filtering, and integrated alert logging. It serves as the critical entry point for the entire detection pipeline, dictating the quality and granularity of data available to the detection engines.

#### **Configurable Capture Options**

This module operates by first configuring a network interface for live packet capture. The selected interface may be dynamically obtained through environment variables or set manually by the user via the system's graphical interface. To accommodate diverse monitoring requirements, the module supports the use of Berkeley Packet Filter (BPF) syntax. This allows users to define highly specific capture rules, such as tcp port 80 or not port 53, thereby reducing processing load and narrowing the focus to relevant traffic. The sniffing process itself is designed for robustness and responsiveness, using a time-bounded polling loop with a two-second sniffing timeout (configurable in the environment variables in .env) to prevent indefinite blocking.

#### **Robust Packet Statistics Tracking**

Captured packets are handed off to a central processing routine (process\_captured\_packet) which updates multiple runtime statistics. These include packet count, byte count, dropped packets, and packet timestamps. The module

intelligently calculates packet size using a tiered strategy, falling back from Scapy's `len()` method to raw layer estimates and minimum frame assumptions when necessary. This redundancy ensures accuracy in environments with incomplete packet metadata.

### **Asynchronous Analysis**

To support asynchronous analysis, packets are inserted into a bounded FIFO queue for batch processing. The queue is carefully monitored to avoid overflow; once it reaches 90% of its maximum capacity, packet capture is automatically throttled to prevent system instability. In scenarios where the queue is full, packets are dropped, and a running count of such events is maintained to evaluate packet loss during high-volume sessions.

### **Network Flow Assembly**

The packet processing thread consumes packets in configurable batches, defaulting to around 50 at a time, which balances throughput and responsiveness. For each packet, it invokes the Traffic Analyser, a separate component responsible for flow assembly, payload extraction, and protocol dissection. The analyser returns flow-level summaries that are then used for statistical updates and alert generation. Alerts, if generated, are stored with timestamps and packet summaries for later retrieval and can be logged in a separate alert log file.

### **Configurable Network Data Logging**

Logging functionality is embedded and configurable. When enabled, the module records packet summaries and flow analysis results into structured log files. It also generates a secondary alert log, isolating intrusion-relevant data for quicker review. The logging mechanism respects disk space limitations by using buffered writes and optional file rotation mechanisms.

### **DoS Tracking Subsystem**

To improve detection of distributed or fragmented attacks, the module includes a lightweight DoS Tracker. This subsystem monitors flow volume per IP address over configurable time windows and evaluates protocol-specific thresholds. If a host exceeds traffic thresholds within the defined interval, it is flagged as suspicious, and a high-confidence alert is generated even before reaching the LLM stage. This helps ensure

responsiveness to flooding attacks and rate-based anomalies that may otherwise go undetected.

### **Runtime Control and Visibility**

The component further exposes interfaces for runtime control, including methods to start or stop packet capture, enable or disable logging, and query active capture statistics. These statistics include packets per second, bytes per second, dropped packet count, and queue utilisation rate—all of which are computed with high-resolution timestamps and running counters. This diagnostic data is not only useful for system tuning but also provides valuable context during system evaluation and performance benchmarking.

### **Design Principles**

From a software engineering perspective, the Packet Capture Module demonstrates modularity, concurrency control, and resiliency. Its thread-safe design ensures continuous packet ingestion and processing under variable traffic loads. It features exception-safe operations at every critical point, from packet dissection to queue operations, to maintain robustness even under malformed or unexpected traffic conditions.

## **4.2.2 LLM-Based Detection Module**

### **Overview**

The LLM-Based Detection Module provides advanced, context-aware intrusion detection using a locally hosted large language model (LLM). It integrates with the Ollama framework to perform semantic analysis of network flows that may not trigger conventional signature-based alerts. This module enhances the system's ability to detect complex, evolving, or zero-day threats by reasoning over flow-level metadata in natural language format. It operates asynchronously to ensure system responsiveness and scalability, even under high throughput conditions.

### **Architecture and Integration**

At its core, the detection engine is implemented as a Python class that runs in a dedicated processing thread, separate from the main packet capture and analysis pipeline. Incoming flow data is added to a bounded queue and processed either periodically or whenever a batch size threshold is met. The component connects to the Ollama server via REST API, using endpoints to verify server availability, list supported models, and submit prompt-based queries for real-time analysis.

Before processing flows, the system ensures the specified LLM model (e.g., mistral, llama, gemma) is available on the server. The module dynamically checks the health and capabilities of the Ollama instance at runtime and gracefully degrades if resources are unavailable. These operational checks allow the system to function autonomously in both online and offline modes, ensuring fault tolerance.

### **Flow Analysis Workflow**

The LLM Detection Engine continuously collects and batches flow records from the real-time traffic analyser. Each flow record includes protocol-level details such as source/destination IPs, ports, packet and byte counts, duration, and, for TCP flows – flag statistics. These records are formatted into a structured prompt, which is sent to the LLM for evaluation.

The prompt instructs the LLM to focus on identifying explicit malicious patterns such as DoS attempts, port scanning, brute-force login trials, and suspicious payload anomalies. The LLM is instructed to ignore benign anomalies and to only return alerts with high confidence ( $\geq 0.7$ ). This strict filtering reduces false positives and aligns the module with practical security response needs.

### **RAG Integration**

RAG enhances this module by retrieving contextual information, such as CVE descriptions, historical attack reports, and threat intelligence feeds from a vector store or knowledge base prior to LLM evaluation. This retrieved data would be appended to the prompt dynamically, allowing the LLM to reason over both the live flow data and relevant background knowledge. Such integration would enable more informed,

context-sensitive decisions, particularly for zero-day or polymorphic threats that lack signatures.

### **Alert Generation and Post-Processing**

Upon receiving the LLM's response, the system parses the returned data into structured alert objects. Each alert includes a unique signature ID ( $\geq 9000$ ), the protocol involved, a threat description, severity level, confidence score, and metadata about the implicated flow. Alerts that fall below the confidence threshold are automatically discarded. All validated alerts are dispatched to registered callback functions, which forward them to the Alert Logger and Blockchain Logger for persistent storage and verification.

Additionally, the LLM Detection Module contributes to system statistics, maintaining counters for flows analysed, API calls made, alerts generated, and errors encountered. These metrics are accessible via the system's web dashboard and are essential for monitoring the health and efficiency of the AI pipeline.

### **Design Principles**

The design of this module is guided by several key software engineering principles: modularity, asynchronous processing, fault tolerance, and context-awareness. Modularity ensures the component operates independently from the rest of the system, allowing for easy updates, testing, and future integration with alternative LLM providers or Retrieval-Augmented Generation (RAG) backends. Asynchronous processing is achieved through multithreaded execution and queue-based batch handling, enabling the system to analyse traffic in near real time without bottlenecking the main detection pipeline. Fault tolerance is embedded through regular health checks, dynamic model verification, and robust error handling for API communication, ensuring the module can gracefully degrade during failures. Finally, context-awareness is a core design goal, with prompts structured to encourage the LLM to evaluate patterns based on semantic relationships, flow statistics, and potential threat intelligence, hence making it capable of detecting sophisticated, low-signature attacks that traditional engines may overlook.

### 4.2.3 Signature-Based Detection Module

#### Overview

The Signature-Based Detection Module is a rule-driven engine responsible for identifying known attack patterns within network flows. It complements the LLM-Based Detection Module by providing deterministic and fast detection capabilities using predefined signatures. This dual-layered detection approach ensures that the system can detect both conventional threats with high precision and novel or ambiguous ones through AI-based reasoning. The signature engine is particularly effective in recognising attacks such as SQL injection, cross-site scripting (XSS), brute-force logins, and port scans, where known payload characteristics or behavioural patterns are present.

#### Architecture and Implementation

The core of the signature engine is built around a modular detection pipeline consisting of three major components: the Signature Manager, the Signature Database, and the Detection Engine. The Signature Manager is responsible for loading, parsing, updating, and maintaining the rule set stored in a JSON-formatted signature database (signatures.json). Each rule includes attributes such as a unique signature id, protocol, matching conditions (e.g., port number, TCP flags, rate limit, time window, or payload keywords), threat category, severity level, CVEs list, action, and metadata.

During packet processing, flows extracted from the traffic analyser are passed through the Detection Engine. Each flow is compared against the active rule set in real time. Matching conditions are evaluated using both packet metadata and decoded payloads when available. The matching process is optimised to run within milliseconds per flow, ensuring minimal performance overhead even when dealing with hundreds of concurrent flows.

The signature matching algorithm is implemented as a conditional matcher that inspects various components of each flow, including source and destination ports, IP addresses, protocol types, and embedded payload content. Payload matching supports both exact string comparisons and substring detection, enabling flexible rule definitions. For



example, a rule may trigger if a TCP packet on port 80 contains the string “/admin/login”, indicating a possible brute-force login attempt.

In addition to simple pattern matching, the engine supports compound conditions, such as flag combinations (e.g., SYN without ACK for SYN floods) and flow statistics (e.g., packet frequency or byte size thresholds). These conditions are combined using logical conjunctions to define richer detection criteria. The design ensures that rule evaluation remains both transparent and explainable, which is important for forensic analysis and threat validation.

### **Alert Generation and Formatting**

Upon detecting a signature match, the module generates an alert object containing structured metadata. This includes the signature ID, a descriptive name, threat category (e.g., intrusion, reconnaissance, malware), severity rating, protocol type, CVEs list, action, and metadata. The alert also includes contextual flow information such as IP addresses, ports, and timestamps. Alerts are then forwarded to the Alert Logger and Blockchain Logger modules for storage and immutability. This standardised format allows seamless integration with the rest of the system, including the GUI dashboard and blockchain contract for audit logging.

Each alert is also assigned a human-readable description to aid in incident response. For instance, a rule that matches excessive failed SSH login attempts would produce an alert with the message: “Multiple SSH login failures from a single source detected within a short time frame”, as defined in the description of the rule. Such contextual tagging helps administrators understand the nature of the threat quickly and take appropriate actions.

### **Rule Management and Extensibility**

The Signature Manager offers an interface for dynamically updating rules without restarting the system. New signatures can be added at runtime, and outdated ones can be removed or modified via the GUI or backend API. The signature database is structured for readability and extensibility, allowing administrators or researchers to define new rules using clear JSON fields.

To maintain performance, the engine enforces constraints such as rule indexing and protocol-based filtering prior to evaluation. This reduces the number of comparisons required per packet, especially in high-throughput scenarios. Additionally, built-in validation ensures that malformed rules are rejected during loading, maintaining system stability.

### **Design Principles**

The design of the Signature-Based Detection Module is also guided by the principles of efficiency, modularity, clarity, and extensibility. Efficiency is achieved through fast, rule-based matching that enables real-time detection without introducing system lag, making it suitable for high-throughput environments. The module is developed in a modular manner, separating rule management, detection logic, and alert handling, which simplifies maintenance and allows independent upgrades. Clarity is prioritised in the rule definition format, which uses readable JSON structures to ensure that signatures are understandable and easy to audit. Finally, extensibility is embedded into the architecture by allowing dynamic rule updates at runtime and supporting a wide range of match conditions, including payload strings, port numbers, and protocol-specific flags, thus making it adaptable to evolving threat patterns and new network protocols.

#### **4.2.4 Alert Logger Module**

##### **Overview**

The Alert Logger Module is responsible for the structured recording of all security alerts generated by the detection components, namely the Signature-Based Detection Module and the LLM-Based Detection Module. It acts as a centralised repository for security events, ensuring that every detected anomaly is documented with detailed metadata for further analysis, auditing, and forensic investigation. This module plays a critical role in maintaining visibility into system activity and acts as the bridge between detection engines and persistent or immutable storage such as the blockchain.

### **Architecture and Functionality**

The module is designed to operate as an event-driven component within the intrusion detection system. Alerts are passed to it in real time via callback functions or direct method invocations from the detection modules. Each alert is expected to follow a structured format containing a signature ID, description, category, severity level, confidence score (for LLM alerts), protocol type, and flow-related metadata (source/destination IP, ports, timestamps). Upon receiving an alert, the module serialises the data into JSON format and appends it to an in-memory or file-based alert log.

The logging system is extensible to support multiple output formats. Currently, alerts are logged to structured .log files, and optionally, printed to the console for debugging or real-time monitoring. Logs are timestamped and include identifiers that facilitate filtering and cross-referencing. This ensures that alerts can later be analysed individually or in aggregate, enabling pattern recognition, incident correlation, and historical trend analysis.

### **Alert Classification and Management**

The Alert Logger distinguishes between alerts based on their source (LLM vs Signature) and category (e.g., reconnaissance, intrusion, DoS, anomaly). This classification allows alerts to be grouped and prioritised. The module also tracks alert statistics, such as the number of alerts received per source and category, which can be visualised via the GUI dashboard.

To prevent duplicate logging, the module includes basic de-duplication logic based on timestamp, signature ID, and flow metadata. Additionally, log rotation or size-based splitting can be implemented to avoid excessive disk usage, especially during high alert volumes. This ensures the system remains performant and storage-efficient over extended monitoring periods.

### **Interoperability with Other Modules**

The Alert Logger interfaces directly with both the Blockchain Logger Module and the Frontend GUI Module. It provides the blockchain component with verified and formatted alerts ready for hashing and on-chain submission. Simultaneously, it pushes

alert data to the GUI controller to enable real-time display and user notifications. These interactions are handled using lightweight message passing or shared memory structures, ensuring minimal latency and maximum responsiveness.

The module is also designed to be compatible with future extensions, such as sending alerts to an external SIEM (Security Information and Event Management) system, exporting to CSV for offline analysis, or triggering automated incident response scripts. Its core structure provides a flexible foundation for integration into larger security infrastructures.

### **Design Principles**

The design of the Alert Logger Module is grounded in the principles of reliability, transparency, interoperability, and scalability. Reliability is ensured through structured data handling, fault-tolerant writing operations, and fallback mechanisms. Transparency is reflected in the readable and standardised alert format, which supports traceability and auditability. Interoperability is achieved by adhering to common data exchange formats and exposing well-defined interfaces for external modules like the blockchain handler and frontend controller. Lastly, the module is built for scalability, capable of handling high alert volumes through queuing mechanisms and batch processing if required.

#### **4.2.5 Blockchain Logger Module**

##### **Overview**

The Blockchain Logger Module is designed to provide tamper-proof, decentralised storage of critical security alerts generated by the intrusion detection system. By leveraging smart contracts deployed on a blockchain platform, this module ensures that high-severity alerts are immutably recorded, enabling transparent auditing, verifiable logging, and long-term accountability. Its inclusion marks a significant advancement in IDS architecture, introducing a layer of trust and integrity that conventional storage methods cannot offer.

### **Architecture and Smart Contract Interaction**

The module operates as a backend service that interacts with an Ethereum-compatible blockchain via Web3.py. A smart contract, written in Solidity and deployed to a local blockchain network (e.g., Ganache or a testnet), defines a public function to store alert hashes along with associated metadata such as timestamp, severity level, and alert ID. This contract acts as a distributed ledger, allowing any stakeholder to verify that an alert was recorded without the possibility of retroactive modification.

Alerts are received from the Alert Logger Module in structured JSON format. Before submission, each alert is serialised and hashed using a secure hashing algorithm (typically SHA-256). The hash is then sent to the smart contract along with key fields like signature id, protocol, and severity. The module uses a configured Ethereum account and private key to sign and submit transactions, ensuring cryptographic authenticity. The module also synchronises the local database with the blockchain network to ensure a reliable storage and sharing of alerts across the WAN of an organisational network.

### **Transaction Handling and Error Management**

The Blockchain Logger handles blockchain transactions asynchronously to avoid blocking the main detection flow. It queues incoming alerts and processes them in batches or on a rolling basis, depending on network load and configuration. Each transaction includes a gas estimate to ensure successful execution without exceeding block limits.

To maintain resilience, the module includes comprehensive error handling. If a transaction fails due to insufficient gas, nonce issues, or network disconnection, the alert is requeued with exponential backoff. Critical errors are logged with detailed diagnostics, and alerts that repeatedly fail submission are backed up locally for manual review. This ensures that no high-priority alert is lost, even in cases of blockchain failure or instability.

### **On-Chain Alert Verification**

Once an alert hash is stored on-chain, it becomes publicly verifiable. Any party with access to the contract address and blockchain explorer can confirm the existence, time,

and content of a submitted alert. This feature provides strong guarantees against data tampering and supports regulatory compliance in contexts where audit trails and incident forensics are legally required.

To verify alerts, users can compare the hash of a local alert record with those stored on the blockchain. If the hashes match, the alert is confirmed to be authentic and untampered. This mechanism ensures that the blockchain functions as a single source of truth for critical events, fostering trust between system operators and external stakeholders.

### **Integration with Other Modules**

The Blockchain Logger integrates seamlessly with the Alert Logger Module via a standardised interface. It receives alerts that are either flagged as critical or manually selected by the system operator for on-chain logging. It also interacts with the GUI Module, exposing on-chain status indicators such as transaction confirmation, gas usage, and contract sync status. These updates provide users with real-time visibility into blockchain operations and system integrity.

Moreover, the module is capable of exporting alert hashes to the GUI for verification purposes. This enhances user trust and transparency, especially in collaborative or multi-user environments where accountability is vital. Integration with the Web3.py framework ensures compatibility with a wide range of Ethereum tools and networks, facilitating future migration to production-grade blockchains.

### **Design Principles**

The Blockchain Logger Module is built on the principles of immutability, trustlessness, resilience, and accountability. Immutability is achieved by anchoring alerts to a blockchain ledger, preventing any form of retroactive alteration. Trustlessness removes the need for third-party verification by allowing cryptographic proof of alert authenticity through on-chain hashes. Resilience is enforced through asynchronous queuing, retry logic, and local backup of failed submissions, ensuring system reliability even under network faults. Finally, accountability is embedded through transparent transaction tracking, hash verification, and contract-based access to alert history.

#### **4.2.6 Frontend GUI Module**

##### **Overview**

The Frontend Graphical User Interface (GUI) Module serves as the visual interface for administrators and users to monitor, interact with, and control the Blockchain-Based Intrusion Detection System. Designed as a web-based dashboard, it provides real-time visibility into network activity, detected alerts, system statistics, and blockchain status. The GUI bridges the gap between complex backend logic and human usability, allowing even non-technical users to effectively interpret and manage security data.

##### **Architecture and Technologies Used**

The frontend is implemented using standard web development technologies – HTML, CSS, and JavaScript, with additional styling handled via a dedicated stylesheet (styles.css) for responsive and user-friendly layouts. Dynamic content rendering is performed by app.js, which handles real-time updates from the Flask backend using asynchronous HTTP requests (AJAX). This separation of structure (HTML), style (CSS), and behaviour (JS) follows modern frontend design best practices and supports maintainability and scalability.

The GUI layout is structured into distinct sections, including a Live Network Traffic Table, Alerts Feed, Blockchain Status Panel, and System Controls. These areas are clearly delineated to minimise cognitive load and maximise situational awareness. Colour-coded severity levels, icons, and tooltips further enhance usability by allowing quick visual parsing of critical information.

##### **Real-Time Data Visualisation and Interaction**

One of the key features of the GUI is its ability to display live traffic flows and alerts in real time. Using polling mechanisms, the frontend periodically queries the backend Flask application for updates to active flows, system status, and alerts. These are then rendered into HTML tables and visual components without requiring a full page reload, creating a smooth and responsive user experience.

For each flow, the GUI shows protocol type, source and destination IPs and ports, packet and byte counts, and connection duration. Alert entries include the detection

method (LLM or signature), severity, threat description, and timestamp. Users can click on individual entries to expand further details, such as TCP flag history or LLM-generated explanations, thereby supporting deeper investigation when needed.

### **System Control Features**

Beyond passive monitoring, the GUI offers interactive system control elements. Users can start or stop packet capture, clear active flows, enable or disable blockchain logging, and adjust filter settings. These actions are sent to the backend using HTTP requests and handled securely to prevent unintended system disruption.

Additionally, a dedicated panel displays the status of key subsystems, including the LLM Detection Engine, Signature Engine, Alert Logger, and Blockchain Logger. Metrics such as queue sizes, flow counts, active threads, and blockchain sync status are updated periodically, providing a comprehensive operational overview.

### **User Experience and Accessibility**

Special care has been taken to ensure the GUI is intuitive, responsive, and accessible across different devices and screen sizes. The design employs a clean, dark-themed aesthetic to reduce eye strain during prolonged use. Font sizes, spacing, and layout responsiveness have been optimised using CSS media queries and flexible grid layouts. Furthermore, error messages, system warnings, and success notifications are shown as toast alerts or embedded banners, improving feedback without interrupting user flow.

### **Design Principles**

The Frontend GUI Module is designed based on the principles of usability, clarity, responsiveness, and separation of concerns. Usability ensures that all features are easily accessible and understandable, even to non-expert users. Clarity is achieved through a clean layout, logical grouping of components, and consistent visual language. Responsiveness guarantees that the interface adapts fluidly to various devices and network speeds, maintaining reliability in diverse environments. Finally, the strict separation of structure (HTML), presentation (CSS), and behaviour (JavaScript) adheres to best practices in frontend engineering, enabling maintainability, modularity, and future extensibility. These principles make the GUI an essential, user-friendly interface that empowers effective monitoring and control of the entire detection system.



### **4.3 Database and Storage Design**

The database and storage design of the Blockchain-Based Intrusion Detection System is centred around lightweight, high-performance mechanisms that support real-time data flow, efficient alert recording, and secure integration with both local and decentralised storage. Given the system's hybrid architecture, consisting of live traffic capture, AI-driven analysis, and blockchain logging, the storage model is designed for speed, modularity, and extensibility. Other than relying on a traditional relational database, the system also uses in-memory data structures, flat-file logging, and smart contract-based blockchain records to manage different categories of data. This section aims to outline the database and storage design implemented in the system to achieve such requirements.

#### **4.3.1 Network Flow Data Handling**

Network flows, as generated by the Traffic Analyser, are not permanently stored in a central database. Instead, they are temporarily held in memory using Python dictionaries and lists. This decision is based on the high throughput and volatile nature of flow data, which is primarily used for transient processing by detection engines. Flow objects are indexed using hash-based keys derived from their 5-tuple identifiers (source IP, destination IP, source port, destination port, protocol). Each flow entry maintains its own statistical profile, such as byte count, packet frequency, duration, and TCP flag distribution, stored locally within the NetworkFlowStatistics object. These data structures are periodically pruned to manage memory usage and flow expiration based on timeout policies.

#### **4.3.2 Alert Database and File-Based Log**

When a detection engine, either signature-based or LLM-based identifies a suspicious or malicious network flow, the corresponding alert is forwarded to the Alert Logger Module. This module then performs dual persistence by recording the alert into both a SQLite relational database and a structured flat .log file. The SQLite database functions as the primary structured store, designed to support querying, filtering, and future integration with analytics tools or external dashboards. Each alert entry is stored as a

row in the database, containing fields such as id, name, category, severity, flow\_hash, timestamp, details, synced\_to\_chain, tx\_hash, created\_at (timestamp). The relational model enables fast retrieval of specific alerts based on criteria like date range, detection method, or severity level. The database scheme for alert storage is shown in Table 4.3.1 below.

Field Name	Data Type	Constraints	Description
id	INTEGER	PRIMARY KEY AUTOINCREMENT	Unique identifier for each alert.
name	TEXT	NOT NULL	Name or title of the alert (e.g., “LLM: TCP Flood”).
category	TEXT	NOT NULL	Type of detection category (e.g., dos, intrusion, llm-detection).
severity	TEXT	NOT NULL	Threat level of the alert (e.g., low, medium, high).
flow_hash	TEXT	NOT NULL	SHA-256 hash of the flow data for integrity and traceability.
timestamp	INTEGER	NOT NULL	UNIX timestamp (in seconds) indicating when the alert was generated.
details	TEXT	NONE	Additional JSON-encoded flow or alert metadata for context (optional).
synced_to_chain	INTEGER	DEFAULT 0	Indicates whether the alert was synced to blockchain (0 = no, 1 = yes).
tx_hash	TEXT	UNIQUE	Blockchain transaction hash if the alert was recorded on-chain.
created_at	INTEGER	DEFAULT (strftime('%s', 'now'))	UNIX timestamp when the record was created.

**Table 4.3.1** Database Schema for Alert Storage

In parallel, alerts are also serialised into human-readable JSON format and appended to a flat log file using buffered I/O. This secondary file-based logging approach ensures data redundancy and provides a portable, platform-independent snapshot of system activity. It is especially useful in scenarios where lightweight external review or quick diagnostics are needed without database access. These log files preserve chronological ordering and can be archived, rotated, or exported for audit purposes.

### 4.3.3 Blockchain-Based Immutable Storage

the system incorporates blockchain-based storage via the Blockchain Logger Module. This acts as a secondary, immutable data store, designed to guarantee the integrity and authenticity of critical security events. Alerts selected for on-chain recording are first hashed using SHA-256 to produce a fixed-length digest. The hash, along with selected metadata (e.g., severity, protocol, timestamp), is then submitted to a deployed smart contract on an Ethereum-compatible blockchain.

This smart contract maintains a verifiable, append-only record of alerts that cannot be tampered with or erased. Any third party can audit the on-chain data by comparing locally stored logs with the blockchain entries. This hybrid model combining off-chain file-based logs and local database (SQLite) for alert storage with on-chain decentralised hashes ensures both operational efficiency and forensic-grade data integrity.

### 4.3.4 Temporary Queues and Runtime Storage

In addition to persistent logging, the system employs several in-memory queues and buffers to coordinate real-time operations. For instance, the LLM Detection Module uses a thread-safe FIFO queue to batch flow records prior to submission to the LLM. Similarly, packet capture statistics and DoS tracker data are stored in local runtime variables and updated continuously. These ephemeral stores are essential for maintaining low-latency operations but are cleared periodically to avoid memory saturation.

System configuration data, such as interface selection, threshold values, and model settings, are loaded from .env files or predefined JSON configuration files. This approach simplifies deployment and allows non-developers to modify system parameters without editing source code.

## 4.4 Smart Contract Design

### Overview

The smart contract design in this system introduces a tamper-proof, decentralised storage mechanism for critical security alerts, enabling immutable logging on a blockchain. The smart contract is written in Solidity and deployed on an Ethereum-compatible blockchain (e.g., Ganache for testing). It forms the core of the Blockchain Logger Module, which securely logs a cryptographic summary of selected alerts. By recording high-severity events on-chain, this design ensures data integrity, traceability, and accountability—key requirements in modern intrusion detection systems where auditability is critical.

### Contract Structure and Key Functions

The smart contract is implemented under the name `AlertsContract.sol` and manages a dynamic array of alert records. Each record contains five primary fields: name, category, severity, flowHash, and timestamp. These fields are submitted by the backend via a `logAlert()` function, which is publicly accessible and non-payable, meaning it does not require Ether to execute.

A corresponding `AlertLogged` event is emitted each time a new alert is recorded, allowing external listeners, such as blockchain explorers or backend systems, to monitor blockchain activity in real time. To retrieve stored alerts, the contract provides two view functions: `getAlertCount()` returns the total number of alerts stored, while `getAlert(index)` returns the details of a specific alert by index. These functions ensure transparency and allow verifiers or auditors to independently access stored alerts without altering the blockchain state.

### Functionality and Usage

The `logAlert()` function is the primary entry point for recording alerts on-chain. It requires all core alert details to be passed in as arguments. Upon execution, the alert is appended to the internal array and indexed automatically. The function includes parameters for:

- name: A short title for the alert (e.g., “TCP Flood”),
- category: The classification (e.g., “dos”, “intrusion”),
- severity: The assessed threat level (e.g., “high”),
- flowHash: A SHA-256 hash uniquely identifying the flow,
- timestamp: UNIX time when the alert was detected.

Each of these values is stored immutably, ensuring that once written, alerts cannot be altered or deleted. This guarantees the audit trail’s credibility and creates a permanent record for compliance, investigation, or legal evidence.

### **Events and Data Transparency**

The AlertLogged event acts as a broadcast mechanism for external observers. It is triggered every time logAlert() is called and includes indexed fields (name, category, severity) for fast filtering, along with the flowHash and timestamp. This supports use cases such as real-time alert tracking, blockchain monitoring dashboards, and automated threat response systems that react to new on-chain events.

By using events, the contract reduces the need for full on-chain querying and allows alert data to be streamed efficiently to off-chain components via Web3 interfaces. This makes the contract practical for integration in both decentralised and hybrid security systems.

### **Design Principles**

The smart contract design is based on principles of immutability, minimalism, transparency, and cost-efficiency. Immutability is guaranteed by the append-only structure of the alert array, where past entries cannot be modified or removed. Minimalism is maintained by limiting storage to essential fields and avoiding excessive on-chain data, reducing gas costs and improving performance. Transparency is achieved through publicly accessible functions and indexed events, allowing any party to verify the system’s behaviour without needing permission or trust in a central authority. Finally, cost-efficiency is addressed by avoiding unnecessary state changes and implementing read-only functions for external use, ensuring the contract remains practical in both test and production blockchain environments.

## 4.5 Communication Interface Design

### Overview

The communication interface design of the Blockchain-Based Intrusion Detection System establishes how data and control signals are exchanged between internal modules, the frontend GUI, and external systems such as the blockchain network and LLM server. The system follows a modular, microservice-oriented architecture where each component communicates through well-defined, loosely coupled interfaces. This design ensures extensibility, simplifies debugging, and enables real-time data flow without compromising performance or maintainability.

### Internal Module Interactions

Internal communication between system modules is primarily achieved through function calls, thread-safe queues, and callback mechanisms. For instance, the Packet Capture Module and Traffic Analyser push structured flow objects into queues consumed by both the Signature-Based Detection Engine and the LLM Detection Engine. These engines, in turn, send alerts asynchronously to the Alert Logger via registered callback functions. This event-driven design allows modules to operate independently, while ensuring synchronised data processing and responsive behaviour.

The system also employs shared in-memory structures (e.g. Python dictionaries and counters) for live metric tracking, such as packets processed, alerts generated, and queue sizes. These shared variables are exposed to the GUI and backend status APIs for real-time monitoring.

### Backend-to-Frontend Communication

The interface between the backend (Flask) and the frontend GUI is handled using asynchronous HTTP (AJAX) requests. JavaScript scripts periodically poll the backend through REST-like endpoints (e.g. /get\_flows, /get\_alerts, /status) to fetch the latest data. The responses are returned in JSON format, ensuring lightweight transmission and easy parsing in the browser.

User actions on the frontend, such as starting or stopping packet capture, resetting flow tables, or toggling blockchain logging are sent to the backend via POST requests. These

actions are routed through dedicated Flask routes that trigger the corresponding backend methods. This request-response model ensures a smooth and responsive user experience while maintaining control integrity.

### **Backend-to-Blockchain Communication**

Communication between the backend and the blockchain is facilitated via the Web3.py library, which interacts with an Ethereum-compatible node (e.g. Ganache). When an alert is selected for on-chain logging, the backend constructs a transaction by encoding the relevant fields—name, category, severity, flowHash, and timestamp—and submits it to the smart contract via the `logAlert()` function.

The backend also monitors transaction status, gas usage, and contract state (e.g. total alert count) using Web3.py's query functions. Confirmation receipts and transaction hashes are returned to the Flask application and logged into the SQLite alert database, linking on-chain records with local logs.

### **Backend-to-LLM Communication**

The LLM Detection Module communicates with a locally hosted Ollama server via HTTP POST requests. When a batch of flows is ready for analysis, the module generates a structured prompt and sends it to the Ollama API at the `/api/generate` endpoint. The server returns a response containing the LLM's analysis, which is then parsed and converted into structured alert objects.

The LLM communication interface is resilient and supports timeout management, error detection, and retries. If the server is unreachable or a response is malformed, the detection engine gracefully logs the error and continues processing new flows. This ensures robustness in unpredictable runtime conditions.

### **Logging and Monitoring Interfaces**

To support observability, the system provides interfaces for log storage (file-based and SQLite) and status monitoring. Key runtime metrics such as the number of flows analysed, queue sizes, LLM request count, and blockchain sync status are exposed via a `/status` endpoint, which the frontend queries periodically. These metrics allow

administrators to monitor system health, diagnose performance bottlenecks, and verify component readiness in real time.

### **Design Principles**

The communication interface design is grounded in the principles of loose coupling, clarity, scalability, and fault tolerance. Loose coupling ensures that modules remain independently testable and maintainable, as they interact only through clearly defined interfaces. Clarity is achieved by using standard protocols (HTTP, JSON) and consistent API endpoints. Scalability is supported by asynchronous, non-blocking communication patterns and queue-based buffering between processing components. Fault tolerance is embedded through retry logic, timeout handling, and structured error reporting, allowing the system to maintain operational integrity even in the presence of failures or transient network issues. This communication model enables the system to function reliably in real-time environments, while remaining modular and extensible for future upgrades.



## 4.6 Compilation and Setup Design

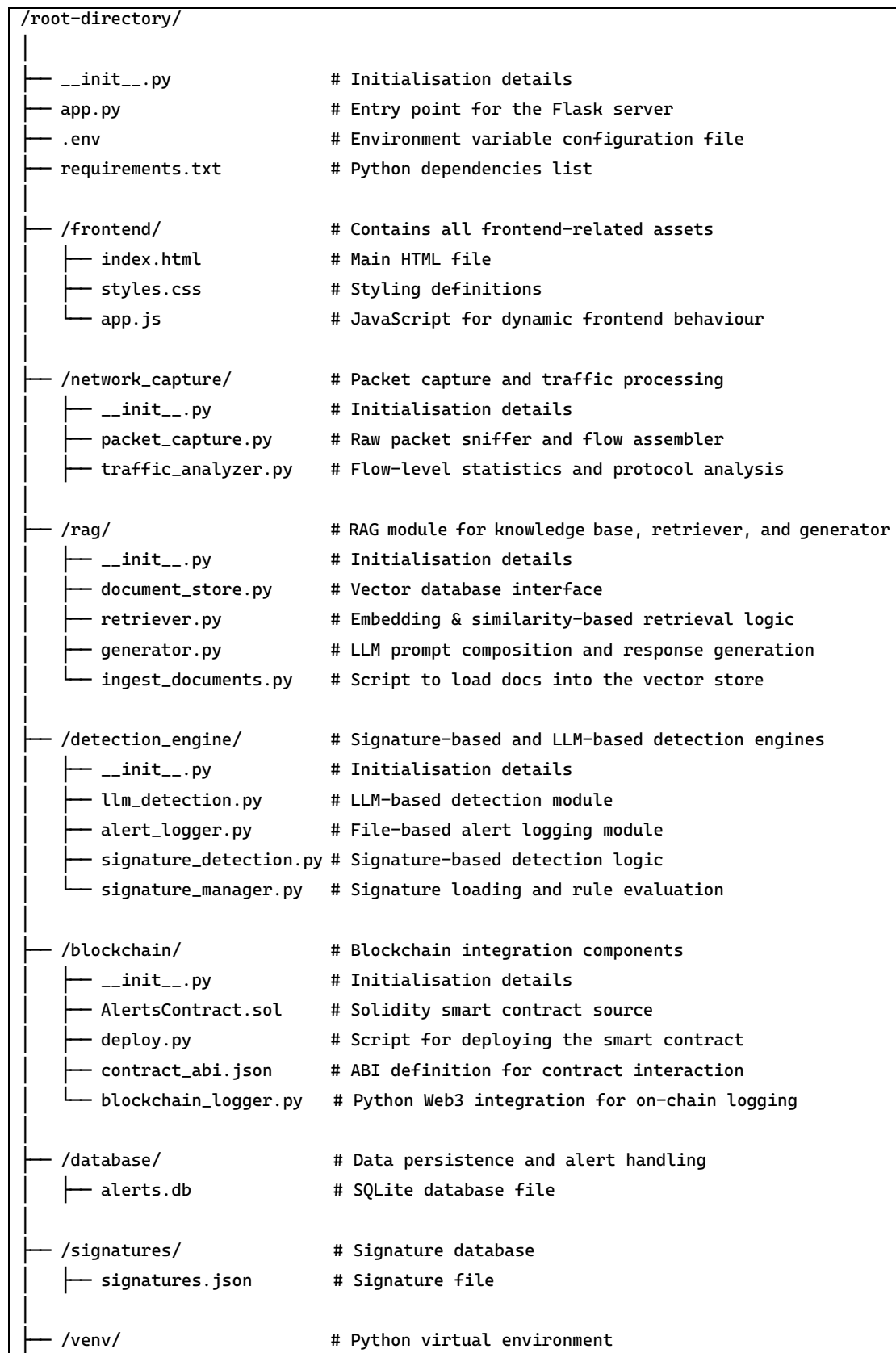
The compilation and setup design of the Blockchain-Based Intrusion Detection System is structured to promote ease of deployment, clarity of codebase organisation, and modular component interaction. A clearly defined folder hierarchy and a set of configuration files govern how the system is installed, initialised, and maintained. This structure ensures that each part of the system, backend, frontend, AI engine, smart contract, and storage, can be independently understood, configured, and extended without introducing ambiguity or dependency conflicts. The folder structure of the project is shown in Figure 4.6.1 below.

Besides, the system's configuration is governed by a set of external files that enable flexibility, portability, and environment-specific customisation. These files define system behaviour, dependencies, data sources, and external integration settings, allowing the software to adapt without the need to modify the source code directly. By separating configuration from logic, the design simplifies deployment and improves maintainability.

The primary configuration file is the `.env` file, which contains environment variables used across the entire system. It defines key operational parameters and URLs for all the modules. These values are dynamically loaded at runtime using the `python-dotenv` package, enabling seamless changes between testing, development, and production environments.

Another important configuration file is `requirements.txt`, which lists all Python libraries required to run the system. This includes modules such as `flask`, `scapy`, `web3`, and `sqlite3`. The python virtual environment can be set up easily with a single command, ensuring consistency across different machines and deployments.

For blockchain integration, `contract_abi.json` file contains the Application Binary Interface (ABI) of the deployed smart contract. This ABI defines the functions, events, and data structures exposed by the contract, allowing the backend to encode transactions, decode responses, and interact with the blockchain securely and correctly.

**Figure 4.6.1** Folder Structure

## **CHAPTER 5 SYSTEM IMPLEMENTATION**

This chapter outlines the practical steps taken to build and deploy the Blockchain-Based Intrusion Detection System. It begins with the setup of the development environment, covering both hardware and software requirements. The chapter then describes the implementation of the backend logic, blockchain integration, and frontend interface. Each section highlights how individual modules were configured to form a fully functional system. It also explains how the system operates in real time and discusses the challenges encountered during implementation, along with the solutions applied.

### **5.1 Environment and Tools Setup**

This section describes the initial setup required to support the development and execution of the system. It covers the hardware and software environments, including the development platforms, libraries, frameworks, and external tools used. The purpose is to ensure a stable and compatible environment that supports all components of the system throughout the implementation phase.

#### **5.1.1 Hardware Setup**

The successful development and deployment of the project require a powerful and adaptable hardware setup that is capable of handling various demanding tasks. In this project, the entire system will be implemented using a single high-performance laptop. The laptop will serve multiple critical functions within the system, hosting of both LLM and blockchain testnet, and multithreaded environments. A detailed overview of the hardware specifications of the laptop used is shown in Table 5.1.1 below.

Hardware Component	Specification
Model	Lenovo IdeaPad Gaming 3 82K2
Processor	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
Operating System	Windows 11 Home
Graphic	NVIDIA GeForce GTX 1650
Memory	16GB DDR4 RAM
Hard Drive	Samsung MZALQ512HBLU-00BL2 SSD 512GB
Random-Access Memory	Samsung M471A1G44AB0-CWE 8GB, Kingston 9905700-118.A00G 8GB
Network Interface Card	MediaTek Wi-Fi 6 MT7921 Wireless LAN Card, Realtek PCIe GbE Family Controller

**Figure 5.1.1** Laptop Specifications

### CPU

The AMD Ryzen 7 5800H is an octa-core, 16-thread processor with a base clock speed of 3.20 GHz. Its multithreaded architecture is crucial for parallel task execution, such as processing captured network packets while simultaneously running intrusion detection models and syncing alerts to the blockchain. The high core and thread count reduces latency in the data pipeline and ensures real-time responsiveness of the system, particularly under high network traffic loads.

### RAM

To support data-heavy operations, the system is equipped with 16GB of DDR4 RAM. This memory capacity is sufficient for LLM-based detection module and the real-time signature-matching engine with is memory-intensive. The RAM also accommodates the simulation of blockchain testnet and the storage of temporary packet buffers, ensuring smooth multitasking and minimal memory bottlenecks.

### Hard Drive

Storage requirements are addressed by a 512GB NVMe SSD, which offers fast read/write speeds. This is vital for both the persistent storage of alert data and the rapid read/write of captured packet as well as system logs. The SSD significantly reduces input/output latency during the preprocessing and batch analysis of network traffic data, enabling timely decision-making and blockchain syncing of detected alerts.

## GPU

Graphical processing requirements for AI-based detection are met using the NVIDIA GeForce GTX 1650 GPU. This GPU, although entry-level by workstation standards, features CUDA cores that accelerate computations, hence improving the LLM performance. The GPU also supports real-time visual analytics rendered through the web interface, allowing system users to monitor the flow status and alert trends efficiently.

## NIC

Lastly, reliable connectivity is enabled through dual interfaces: MediaTek Wi-Fi 6 and Realtek PCIe GbE Ethernet. Wi-Fi 6 provides high-speed wireless communication useful during testing in environments with multiple devices, while the Ethernet port ensures stable and secure data transmission when syncing alerts across blockchain nodes or capturing high-throughput packet streams.

### 5.1.2 Software Setup

#### Programming Languages

The development of the project requires multiple programming languages, each chosen for its unique strengths and suitability to the specific demands of the project. The programming languages selected and a summary of their roles in this project are outlined in Table 3.1.3.1 below.

Programming Language	Version	Role
Python	3.12.5	Powers the backend system
HTML	ECMAScript 6+	Structures the web interface
JavaScript	20.10.0, ECMAScript 6+	Handles dynamic frontend behaviour
CSS	ECMAScript 6+	Styles the web interface
Solidity	0.8.0	Implements the smart contract

**Table 5.1.1** *Programming Language Requirements*

Moving on, various libraries and frameworks are used in developing, testing, and deploying this project. These tools are selected based on their ability to handle specific tasks within the system. The main libraries selected and a summary of their roles in this project are outlined in Table 5.1.2 below.

Library/Framework	Version	Role
Flask	3.1.0	Serves as the backend web framework for building API endpoints
flask_cor	5.0.1	Enables Cross-Origin Resource Sharing (CORS) to allow frontend-backend communication
py_solc_x	2.0.3	Compiles and interacts with Solidity smart contracts.
python-dotenv	1.1.0	Loads environment variables from a .env file for secure configuration
Requests	2.32.3	Handles HTTP requests, mainly used for external API communication
scapy	2.6.1	Captures and processes raw network packets for analysis
numpy	2.2.5	Used for packet statistics and data arrays
pandas	2.2.3	Manages structured data like flow logs and alert histories
seaborn	0.13.2	Creates clean, high-level statistical visualisations
matplotlib	3.10.1	Plots graphs and charts for network activity and flow analysis
web3	7.11.0	Interfaces with the Ethereum blockchain to read/write smart contract data

**Table 5.1.2** *Library and Framework Requirements*

Furthermore, the development and deployment of this project involve a range of tools and platforms that facilitate various stages of the project. These tools and platforms are selected for their ability to streamline development, ensure efficient deployment, and achieve the main objectives of the system. The main tools and platforms selected as well as a summary of their roles in this project are outlined in Table 5.1.3 below.

Tool/Platform	Version	Role
Ollama	0.6.6	Hosts and serves the LLM locally
Gemma3:1b	-	A lightweight LLM model
Ganache	2.7.1	Simulates a local Ethereum blockchain
SQLite	3.42.0	Stores intrusion alerts locally
Git	2.45.2.windows.1	Manages version control for source code
VS Code	1.99.3	Provides an integrated development environment
Chrome	135.0.7049.116 (Official Build) (64-bit)	Used to access and test the web-based user interface

**Table 5.1.3** *Tool and Platform Requirements*

In addition, the operating system plays a fundamental role in supporting the execution of all development tools, libraries, and runtime environments. This project is developed and deployed on Windows 11 Home, which provides a stable and user-friendly environment for multitasking, blockchain simulation, LLM hosting, and network management. Its compatibility with essential tools, programming languages, and virtual networking interfaces makes it a practical choice for building and testing an AI-powered, blockchain-integrated intrusion detection system. The main OS selected for this project is described in Table 5.1.4 below.

OS	Version	Role
Windows 11 Home	23H2	Operating system used to host this system

**Table 5.1.4** *Operating System Requirements*

## 5.2 Blockchain Implementation

The blockchain component of this system ensures that all intrusion alerts generated by the detection engines are logged in a tamper-proof, decentralised ledger. This guarantees the integrity and immutability of critical security events, which is essential in environments where trust, transparency, and auditability are priorities. To support this requirement, a smart contract is deployed on a private Ethereum blockchain using Ganache. The backend system interacts with this smart contract through the Web3.py and py-solc-x libraries. This section outlines the implementation steps involved in deploying and integrating the blockchain component with the intrusion detection system.

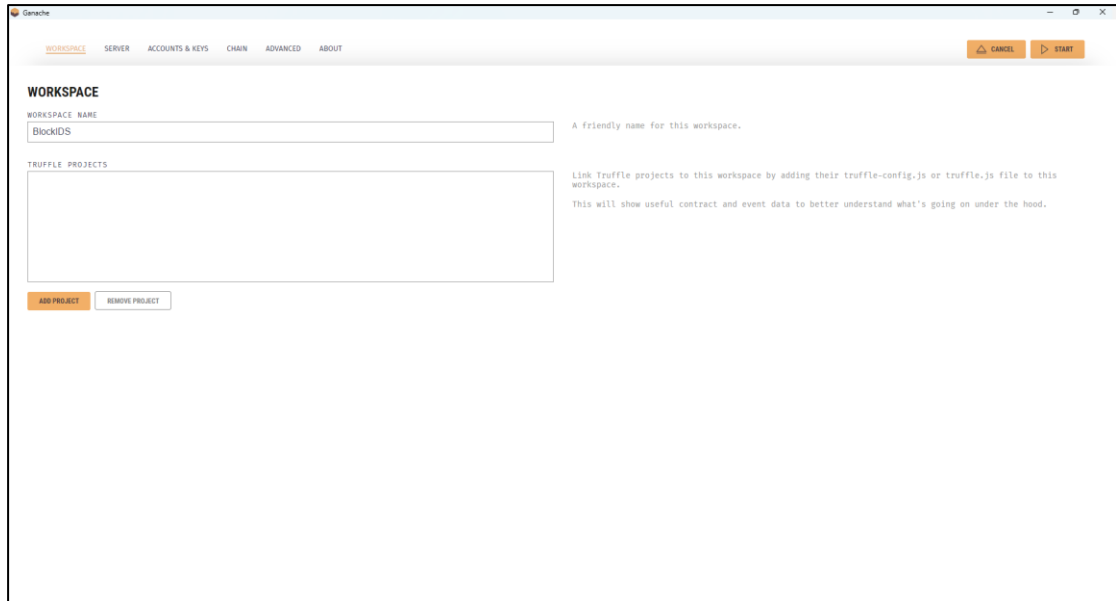
### 5.2.1 Installing and Running Ganache

Ganache is a personal blockchain simulator developed by Truffle, designed for Ethereum smart contract development. It runs a local blockchain on the user's machine, allowing developers to deploy, test, and debug smart contracts in a controlled environment. Ganache provides complete visibility over blockchain operations, including blocks, transactions, and account balances, which is ideal for rapid prototyping and testing without relying on public testnets.

In this project, Ganache is used to simulate a decentralised environment where all intrusion alerts can be logged securely. This ensures that even during the development phase, the system can demonstrate blockchain logging and verification functionality.

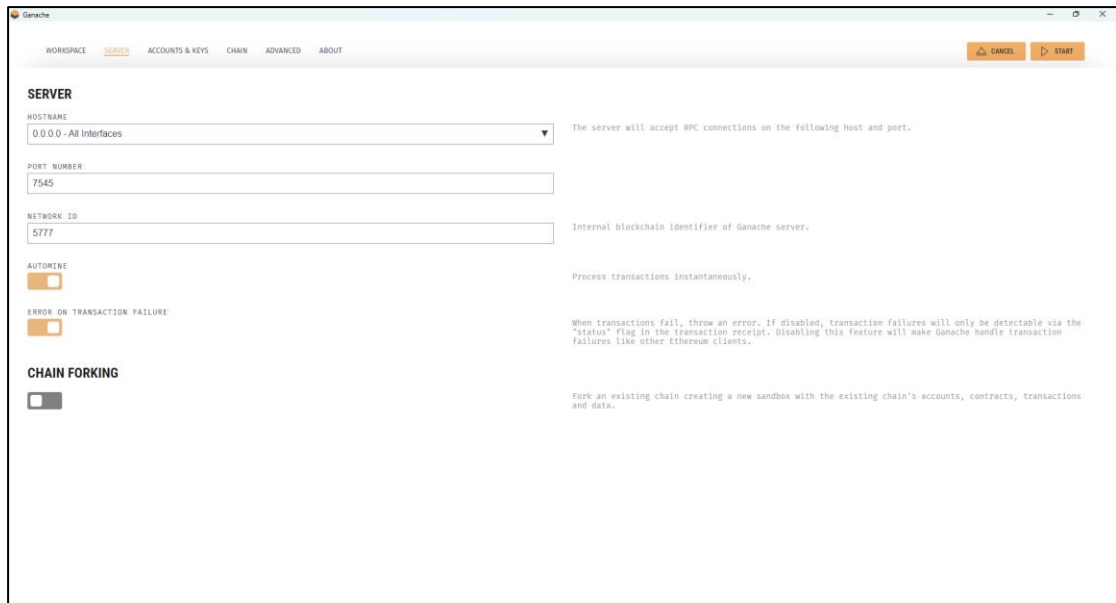
To begin, the Ganache application is downloaded and installed from the official Truffle Suite website at <https://archive.trufflesuite.com/ganache/>. Once launched, the initial step involves configuring a new workspace. As shown in Figure 5.2.1, the workspace is named BlockIDS, which serves as a container for managing the project's smart contracts and blockchain state. Ganache allows for the addition of Truffle projects directly into the workspace for better integration and management; however, in this project, smart contract deployment is managed independently through Python scripts.





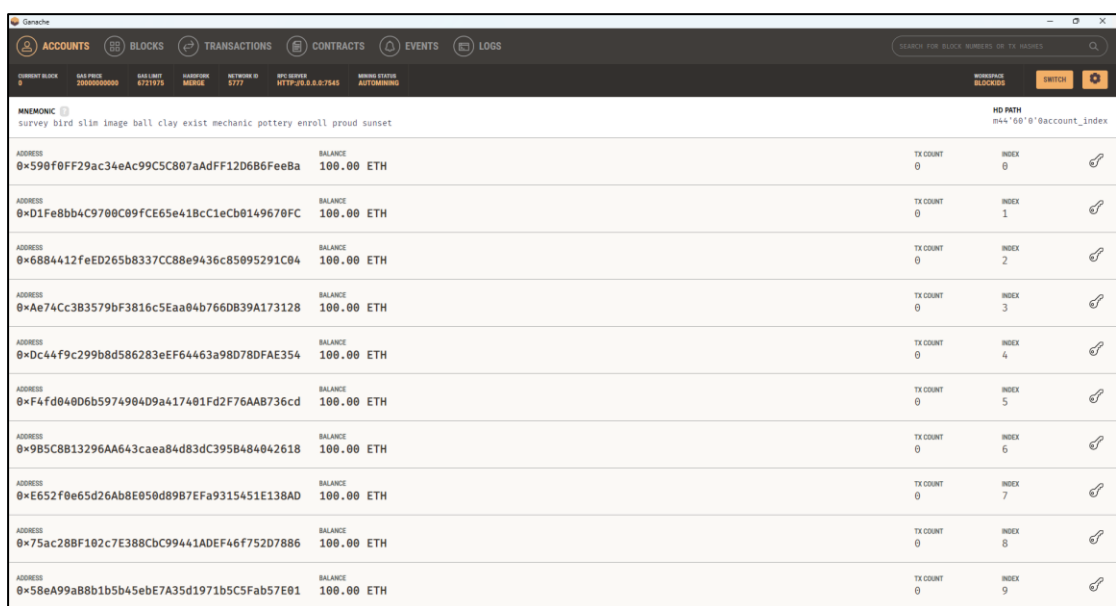
**Figure 5.2.1** *Ganache Workspace Configuration*

Next, the blockchain server settings are configured. As shown in Figure 5.2.2, the hostname is set to 0.0.0.0, representing “All Interfaces”, enabling Ganache to accept RPC connections from any network interface. The default RPC port is set to 7545, which is the endpoint used by the backend system to connect to the blockchain. The network ID is defined as 5777, a common default for local Ethereum networks. Other options such as “Automine” and “Error on Transaction Failure” are enabled to ensure that transactions are processed instantly and any issues are explicitly flagged, which helps with debugging and consistency during smart contract interaction.



**Figure 5.2.2 Ganache Server Configuration**

After the configuration is complete, the “Start” button is clicked to launch the blockchain. Once started, Ganache provides a local Ethereum environment with several pre-funded accounts. These accounts can be used for deploying smart contracts and initiating transactions without requiring real Ether. The use of Ganache in this project eliminates the complexities of public network deployment while preserving all core blockchain features such as transaction signing, contract execution, and block mining. Figure 5.2.3 shows the running Ganache environment after a successful setup.



**Figure 5.2.3 Ganache Environment**

The consistent and user-friendly interface of Ganache simplifies blockchain management, allowing focus to remain on the development and integration of smart contract features. With the environment running and properly configured, the system is ready to compile, deploy, and interact with the smart contract for intrusion alert logging.

### **5.2.2 Setting Up the Python Environment**

To enable Python to interact with the Ethereum blockchain, a set of blockchain-related libraries must be installed. These include Web3.py, a Python library that allows communication with Ethereum nodes via JSON-RPC, and py-solc-x, which acts as a Python interface for compiling Solidity smart contracts. Additionally, python-dotenv is used to load configuration values, such as RPC URLs and contract addresses, from an environment file (.env), which simplifies environment management and enhances security by avoiding hard-coded credentials. This file includes variables such as the Ganache URL, contract ABI path, database path, and contract address.

Installing these libraries in a virtual environment ensures consistency, isolation, and easier dependency tracking during development. By separating configuration from code, the system becomes more maintainable and adaptable to changes. For instance, switching between test and production environments requires no code modification, only a change in the .env file. This also enhances security, as sensitive data is not exposed directly in the source code.

### **5.2.3 Installing and Configuring the Solidity Compiler**

Solidity is the primary programming language for writing smart contracts on the Ethereum blockchain. To compile the smart contract within the Python environment, the appropriate version of the Solidity compiler must be installed using py-solc-x. In this project, version 0.8.0 is selected for compatibility with the contract code. Installing the compiler ensures that the contract can be compiled locally before deployment, converting it into a format that can be understood and executed by the Ethereum Virtual Machine (EVM). This process also generates the ABI (Application Binary Interface), which acts as a bridge between the contract and the backend system.

### 5.2.4 Compiling and Deploying the Smart Contract

Once the smart contract is written (refer to Chapter 4.4 Smart Contract Design), it is compiled within the Python environment using py-solc-x. This step generates the bytecode and ABI, both of which are required for deployment. The backend connects to the Ganache blockchain via the RPC endpoint and uses one of the pre-funded test accounts to deploy the contract. Deployment is performed as a transaction, and the resulting transaction receipt contains the deployed contract address. This address is then saved to the .env file and referenced by the backend system whenever it needs to call the contract. This deployment process ensures the contract is fully registered on the blockchain and ready to receive alerts. The deployment process and outcomes are shown in Figure 5.2.4 and Figure 5.2.5 below.

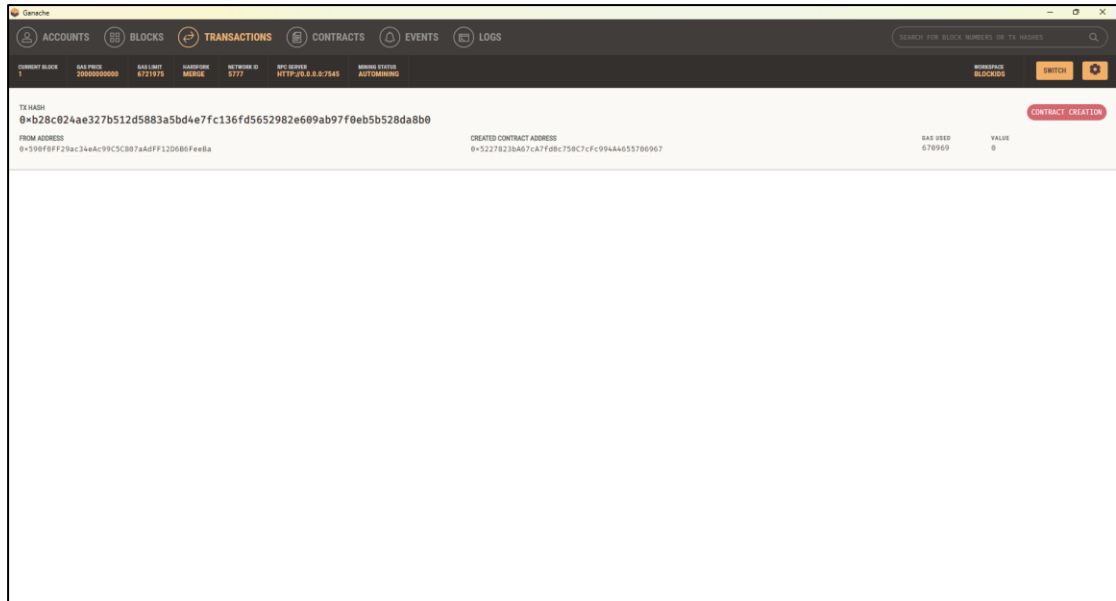
```
(venv) PS C:\dev\Blockchain-Based-IDS-with-AI> python -m blockchain.deploy
INFO: No GANACHE_PRIVATE_KEY found in .env. Using default account from node (suitable for local Ganache).
INFO: Connecting to blockchain: http://127.0.0.1:7545
INFO: Connected successfully.
INFO: Using default account from node: 0x590f0FF29ac34eAc99C5C807aAdFF12D686FeeBa
INFO: Compiling C:\dev\Blockchain-Based-IDS-with-AI\blockchain\AlertsContract.sol using solc version 0.8.0...
INFO: Checking/installing Solidity compiler v0.8.0...
INFO: solc 0.8.0 already installed at: C:\Users\wwenk\.solcx\solc-v0.8.0
INFO: Compilation successful.
Deploying contract...
Building transaction...
Transaction sent with hash: b28c024ae327b512d5883a5bd4e7fc136fd5652982e609ab97f0eb5b528da8b0
Waiting for transaction receipt...
Transaction confirmed.
Contract deployed at address: 0x5227823bA67cA7fd0c750C7cFc994A4655706967
Contract ABI saved to C:\dev\Blockchain-Based-IDS-with-AI\blockchain\blockchain\contract_abi.json

----- Deployment Successful -----
Contract 'AlertsContract' deployed successfully.
Address: 0x5227823bA67cA7fd0c750C7cFc994A4655706967
ABI saved to: C:\dev\Blockchain-Based-IDS-with-AI\blockchain\blockchain\contract_abi.json

Action Required: Update your .env file with the new contract address:
CONTRACT_ADDRESS=0x5227823bA67cA7fd0c750C7cFc994A4655706967

Ensure other services (like the main app) use this updated address.
(venv) PS C:\dev\Blockchain-Based-IDS-with-AI>
```

*Figure 5.2.4 Smart Contract Deployment*



*Figure 5.2.5 Contract Creation on Blockchain*

### 5.2.5 Saving Contract Metadata

The ABI generated during compilation and the address obtained after deployment are critical for contract interaction. The ABI is stored in a separate JSON file (contract\_abi.json), and the address is recorded in the environment file (.env). These two elements together allow the backend system to create an interface to the smart contract, enabling the logging and retrieval of alerts. Managing this metadata externally ensures that if the contract is redeployed (for example, after modification), only these references need to be updated, avoiding the need to alter the core logic of the backend code.

### 5.2.6 Initialising the System with Blockchain Support

Before launching the intrusion detection system, it is essential to verify that Ganache is running and accessible. The backend will then load the contract metadata from the environment and ABI files. If a contract has not yet been deployed, the deployment script can be run manually to compile and deploy it. Once the system is started via the main application script, the backend automatically connects to the blockchain, initialises the logger, and begins syncing alerts. At this stage, the system is fully operational with end-to-end blockchain integration.

### 5.3 LLM Implementation

In this project, a lightweight LLM model is hosted locally using Ollama, which serves as the backend inference engine for AI-driven threat detection. This section describes the implementation steps required to set up and integrate the LLM component with the system.

#### 5.3.1 Installing Ollama

Ollama is an open-source platform designed to run large language models locally with minimal setup. It supports various model architectures, including LLaMA, Mistral, and Gemma, and offers an HTTP API for interaction. For this system, Ollama hosts the gemma3:1b model, a compact and efficient LLM optimised for lightweight inferencing on edge devices or development machines. Its relevance in this project lies in its ability to analyse sequences of network flows and detect nuanced threats such as stealthy scans or early-stage intrusions that may not trigger rule-based alerts. By operating entirely offline, Ollama ensures data privacy and low-latency inference without relying on cloud-based APIs.

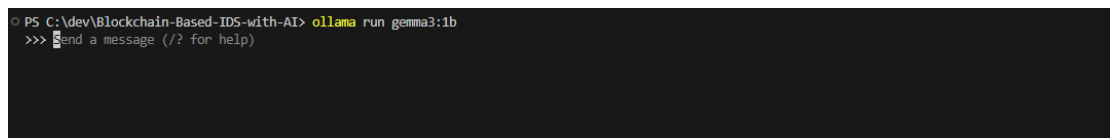
To begin, Ollama is installed on the development machine using its official package for Windows downloadable at <https://ollama.com/>. Once installed, the Ollama service can be started using Command Prompt or Windows Powershell with the “ollama serve” command. The Ollama server runs as a local server accessible via <http://localhost:11434>, as shown in Figure 5.3.1 below.

```
PS C:\dev\Blockchain-Based-IDS-with-AI> ollama serve
2025/05/08 16:30:55 routes.go:1232: INFO server config env="map[CUDA_VISIBLE_DEVICES: GPU_DEVICE_ORDINAL: HIP_VISIBLE_DEVICES: HSA_OVERRIDE_GFX_VERSION:
HTTPS_PROXY: HTTP_PROXY: NO_PROXY: OLLAMA_CONTEXT_LENGTH:2048 OLLAMA_DEBUG:false OLLAMA_FLASH_ATTENTION:false OLLAMA_GPU_OVERHEAD:0 OLLAMA_HOST:http://
127.0.0.1:11434 OLLAMA_INTEL_GPU:false OLLAMA_KEEP_ALIVE:5m0s OLLAMA_KV_CACHE_TYPE: OLLAMA_LLM_LIBRARY: OLLAMA_LOAD_TIMEOUT:5m0s OLLAMA_MAX_LOADED_MODEL
S:0 OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:C:\\Users\\wweenk\\ollama\\models OLLAMA_MULTIUSER_CACHE:false OLLAMA_NEW_ENGINE:false OLLAMA_NOHISTORY:false OLL
AMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:0 OLLAMA_ORIGINS:[http://localhost https://localhost http://localhost:* https://localhost:* http://127.0.0.1 https
://127.0.0.1 http://127.0.0.1:* https://127.0.0.1:* http://0.0.0.0 https://0.0.0.0 http://0.0.0.0:* https://0.0.0.0:* app://* file://* tauri://* vscode-
webview://* vscode-file://*] OLLAMA_SCHED_SPREAD:false ROCR_VISIBLE_DEVICES:]"
time=2025-05-08T16:30:55.452+08:00 level=INFO source=images.go:458 msg="total blobs: 10"
time=2025-05-08T16:30:55.453+08:00 level=INFO source=images.go:465 msg="total unused blobs removed: 0"
time=2025-05-08T16:30:55.455+08:00 level=INFO source=routes.go:1299 msg="Listening on 127.0.0.1:11434 (version 0.6.6)"
time=2025-05-08T16:30:55.455+08:00 level=INFO source=gpu.go:217 msg="looking for compatible GPUs"
time=2025-05-08T16:30:55.455+08:00 level=INFO source=gpu_windows.go:167 msg="packages count=1"
time=2025-05-08T16:30:55.456+08:00 level=INFO source=gpu_windows.go:214 msg="" package=0 cores=8 efficiency=0 threads=16
time=2025-05-08T16:30:56.202+08:00 level=INFO source=gpu.go:319 msg="detected OS VRAM overhead" id=GPU-e4896052-84aa-b28b-c799-3eb0ca609c7d library=cuda
compute=7.5 driver=12.8 name="NVIDIA GeForce GTX 1650" overhead="638.7 MiB"
time=2025-05-08T16:30:56.205+08:00 level=INFO source=types.go:130 msg="inference compute" id=GPU-e4896052-84aa-b28b-c799-3eb0ca609c7d library=cuda varia
nt=v12 compute=7.5 driver=12.8 name="NVIDIA GeForce GTX 1650" total="4.0 GiB" available="3.2 GiB"
```

*Figure 5.3.1 Ollama Local Server*

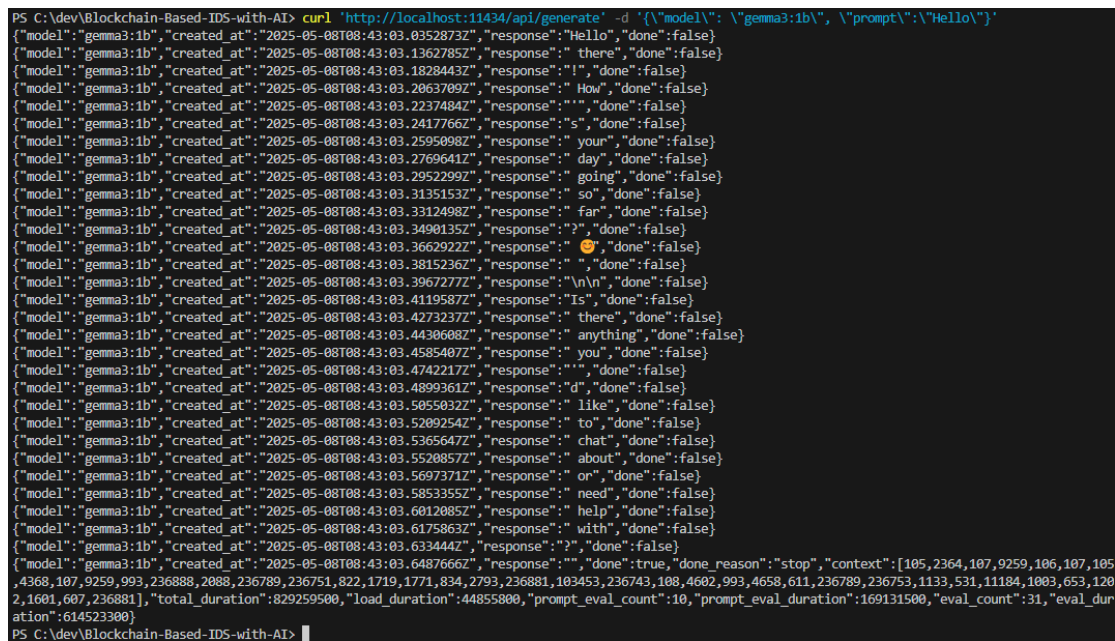
### 5.3.2 Pulling the Required Model

The next step involves pulling the desired model. In this case, the gemma:1b model is downloaded using Ollama's CLI. This model is selected for its balance between performance and resource usage, making it suitable for real-time traffic analysis in environments with limited hardware. After installation, the model is loaded into memory and ready to respond to inference requests sent via HTTP as depicted in Figure 5.3.2 and Figure 5.3.3.



```
PS C:\dev\Blockchain-Based-IDS-with-AI> ollama run gemma3:1b
>>> Send a message (/? for help)
```

*Figure 5.3.2 Running LLM Hosted via Ollama*



```
PS C:\dev\Blockchain-Based-IDS-with-AI> curl 'http://localhost:11434/api/generate' -d '{"model": "gemma3:1b", "prompt": "Hello"}'
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.0352873Z", "response": "Hello", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.1362785Z", "response": " there", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.1828443Z", "response": "I", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2063709Z", "response": " How", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2237484Z", "response": "", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2417766Z", "response": "s", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2595088Z", "response": " your", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2769641Z", "response": " day", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.2952299Z", "response": " going", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3135153Z", "response": " so", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3312498Z", "response": " far", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3490135Z", "response": "?", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3662922Z", "response": " 😊", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3815236Z", "response": " ", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.3967277Z", "response": "\n\n", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4119587Z", "response": "Is", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4273237Z", "response": " there", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4430608Z", "response": " anything", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4585407Z", "response": " you", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4742217Z", "response": "", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.4899361Z", "response": "d", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5055032Z", "response": " like", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5209254Z", "response": " to", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5365647Z", "response": " chat", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5520857Z", "response": " about", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5697371Z", "response": " or", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.5853355Z", "response": " need", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.6012085Z", "response": " help", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.6175863Z", "response": " with", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.633444Z", "response": "?", "done": false}
{"model": "gemma3:1b", "created_at": "2025-05-08T08:43:03.6487666Z", "response": "", "done": true, "done_reason": "stop", "context": [105, 2364, 107, 9259, 106, 107, 105, 4368, 107, 9259, 993, 236888, 2888, 236789, 236751, 822, 1719, 1771, 834, 2793, 236881, 103453, 236743, 108, 4602, 993, 4658, 611, 236789, 236753, 1133, 531, 11184, 1003, 653, 120, 2, 1601, 607, 236881], "total_duration": 829259500, "load_duration": 44855800, "prompt_eval_count": 10, "prompt_eval_duration": 169131500, "eval_count": 31, "eval_duration": 614523300}
PS C:\dev\Blockchain-Based-IDS-with-AI>
```

*Figure 5.3.3 Interaction with LLM via HTTP*

### 5.3.3 Configuring the Environment for LLM

To enable smooth integration between the LLM engine and the backend system, several environment variables must be defined and loaded at runtime. These values are stored in a .env file located in the project directory, similar to the other modules.

The key variables include:

- `LLM_MODEL_NAME`: Specifies the model identifier (gemma:1b).
- `OLLAMA_BASE_URL`: URL of the Ollama server (`http://localhost:11434`).
- `LLM_BATCH_SIZE`: Number of flows per analysis batch.
- `LLM_MAX_QUEUE_SIZE`: Maximum number of flows in the queue.
- `LLM_PROCESSING_INTERVAL`: Interval (in seconds) for checking and processing flow batches.

These variables are loaded at application startup using the `python-dotenv` library. Proper configuration ensures that the system communicates correctly with the Ollama server and that flow data is processed efficiently in batches.

### **5.3.4 Initialising the LLM Detection Engine**

The backend includes a module named LLM Detection Engine (defined in `llm_detection.py`), which is responsible for managing LLM-related operations. Upon system startup, this engine is initialised using the values loaded from the `.env` file. It creates a queue for incoming flow data, starts a separate processing thread, and continuously monitors the queue size and timing conditions. When either the batch size is reached or the interval expires, the engine prepares the data and sends a structured prompt to the LLM.

### **5.3.5 Implementing the RAG Mechanism**

To improve the model's factual grounding and decision-making, Retrieval-Augmented Generation (RAG) is introduced. RAG allows the LLM to retrieve relevant documents or facts from an external knowledge base before generating a response. In this system, a lightweight vector store is used to index a curated set of known threat descriptions, signatures, and tactics (e.g. DoS, brute force, reconnaissance), as well as normal network behaviours to prevent false positives.



When a batch of flows is submitted for LLM analysis, the system extracts key tokens, which are used to perform a similarity search against the local knowledge base using cosine similarity or other embedding techniques. The top-matching documents are appended to the prompt sent to the LLM. This enriches the context of the query and guides the model toward more accurate, explainable, and relevant outputs.

### **5.3.6 Formatting and Sending Network Flow Data**

The system captures and summarises network flow data through the traffic analyser module. This summarised data is then formatted into a natural language prompt, which includes key flow features such as IP addresses, ports, protocol type, packet count, byte volume, and duration. This prompt is sent to Ollama via an HTTP POST request to the `/api/generate` endpoint. The model analyses the flows in context and returns a JSON-like list of potential security alerts with confidence scores and descriptions.

### **5.3.7 Handling and Logging AI-Generated Alerts**

The system processes the model's response, filtering out low-confidence or malformed entries. Valid alerts are converted into the standard format used across the system and passed to the alert management pipeline. These alerts are recorded in the local database and also synchronised to the blockchain via the BlockchainLogger (in `blockchain_logger.py`). This ensures that AI-generated detections are traceable, tamper-proof, and visible to the frontend dashboard.

Similar to the other modules, this module adopts a `.env` file to store all environment-specific variables. This approach allows the application to access key configuration values dynamically without hardcoding them into the source code. Environment variables related to the LLM include the model name to be used, the base URL of the Ollama server, the batch size for LLM processing, the maximum size of the flow queue, the interval for batch processing.

Specifically, the variable `LLM_MODEL_NAME` is set to define which model Ollama should serve for inference. In this system, the value is set as `gemma:1b`, indicating the use of the lightweight model, which balances accuracy and processing speed. The

OLLAMA\_BASE\_URL variable typically points to `http://localhost:11434`, which is the default local address where the Ollama server listens for API requests. This ensures that all LLM operations are performed locally without any external network dependency, improving both performance and data security.

The LLM\_BATCH\_SIZE and LLM\_MAX\_QUEUE\_SIZE variables define how many network flows should be collected before they are sent to the LLM for analysis, and how many flows can be queued at any time, respectively. These parameters are crucial for balancing responsiveness with efficiency, especially during periods of high traffic. The LLM\_PROCESSING\_INTERVAL defines how often the system checks whether the queue has enough data to trigger a batch analysis, allowing timely detection without overwhelming system resources.

## **5.4 Backend Services Implementation**

The backend services form the core logic and orchestration layer of the intrusion detection system, managing packet capture, flow analysis, signature-based detection, alert handling, and frontend communication. Built using Python 3.12.5, the backend is designed to be modular, multi-threaded, and capable of handling real-time network data. This section details the implementation steps and essential components required to set up the backend services for the system to operate effectively.

### **5.4.1 Setting Up the Flask Web Framework**

The entire backend is structured around the Flask microframework, a lightweight web framework written in Python. Flask provides the RESTful API interface that allows the frontend to interact with the system, including issuing commands like starting or stopping packet capture, fetching flow and alert data, and monitoring system status. Flask is selected for its simplicity, scalability, and ease of integration with Python-based services. It supports modular routing and JSON response handling, which are critical for maintaining a clean interface between backend logic and the web frontend.

### **5.4.2 Setting Up the Flask Web Framework**

To set up Flask, the application structure is defined in a single entry point (`app.py`), where routes are declared, and controllers are mapped to their respective services. Flask's built-in development server is sufficient for the system's local use case, and the server is configured to run on all network interfaces (`0.0.0.0`) to support remote frontend access within a controlled environment.

### **5.4.3 Initialising Packet Capture with Scapy**

At the foundation of the intrusion detection system is the packet capture engine, responsible for capturing live network traffic from a specified interface. This functionality is implemented using Scapy, a powerful Python library for packet manipulation and analysis. Scapy supports low-level packet sniffing with the ability to filter, dissect, and decode network protocols.

To initialise packet capture, the system first queries all available network interfaces and allows the user to select one. Once selected, a dedicated thread is started to continuously sniff packets using Berkeley Packet Filter (BPF) syntax to reduce system load by capturing only relevant traffic (e.g. TCP, HTTP, or SSH). Each packet is timestamped and enqueued for further processing. Scapy's direct access to the data link layer makes it suitable for real-time, low-latency capture, which is essential for timely intrusion detection.

#### **5.4.4 Configuring the Packet Processing Queue**

To manage the flow of incoming packets, a multi-threaded queuing system is implemented using Python's queue and threading modules. Captured packets are added to a thread-safe queue with a defined maximum size to prevent memory overflow. Another dedicated thread retrieves packets from the queue in configurable batches and forwards them to the traffic analyser for further inspection.

This batching mechanism is critical for maintaining system responsiveness, especially under high traffic conditions. It decouples packet acquisition from processing, preventing packet loss due to temporary analysis delays. The backend also monitors the queue status to track dropped packets and logs warnings when the queue approaches its maximum capacity.

#### **5.4.5 Implementing Flow-Based Traffic Analysis**

The traffic analysis engine serves to aggregate individual packets into network flows, which are groups of packets sharing the same five-tuple: source IP, destination IP, source port, destination port, and protocol. The flow-based design allows the system to evaluate behavioural patterns rather than individual packets, increasing detection accuracy.

The analyser keeps a dictionary of active flows, updating their statistics, such as total packets, total bytes, session duration, and TCP flag distributions, with each new packet. It also handles flow timeout logic to remove inactive sessions, maintaining a manageable memory footprint. This component is essential for supporting both rule-

based and AI-based detection engines, as they rely on complete flow summaries rather than isolated packets.

#### **5.4.6 Integrating the Signature-Based Detection Engine**

The signature-based engine is responsible for detecting known attack patterns using predefined rules. This component is implemented through a combination of a Signature Manager and a Signature Detection module. The rules are stored in a structured JSON file (`signatures.json`) and loaded during system initialisation. Each rule specifies matching criteria such as IP addresses, ports, protocol types, payload content, TCP flag combinations, and rate limits.

When a new flow is analysed, it is compared against the loaded signatures using the Signature Detection engine. Matching flows trigger alerts, which are then passed to the alert management pipeline. This module offers fast and deterministic detection of common threats like SYN floods, SSH brute force attempts, and SQL injections, ensuring the system can respond to known vulnerabilities with high confidence.

#### **5.4.7 Managing Alerts and Local Storage**

Once an alert is triggered, either by the signature engine or other modules, it is stored locally in an SQLite database. SQLite is chosen for its simplicity, portability, and seamless integration with Python. The alert table schema captures all relevant metadata, including timestamp, severity, category, flow identifier, and detection source. This data is not only used for blockchain synchronisation but also displayed on the frontend dashboard for real-time monitoring.

In addition to storing alerts, the system maintains an alert history in memory, enabling fast access to recent alerts without the overhead of repeated database queries. This hybrid model of in-memory caching and persistent storage ensures a balance between performance and durability.

#### **5.4.8 Configuring the Environment for Backend Services**

To maintain a clean separation between code and configuration, the system uses the `python-dotenv` library to load environment variables from a `.env` file. This file contains critical settings such as interface names, logging options, packet filter strings, and various module-specific parameters. At runtime, these variables are imported and applied to the relevant system components, allowing flexible reconfiguration without modifying source files.

#### **5.4.9 Enabling Cross-Origin Requests and Frontend Communication**

To allow the web-based frontend to interact with the backend services, Cross-Origin Resource Sharing (CORS) is enabled using the `flask_cors` extension. This is necessary because browsers enforce the same-origin policy, which blocks frontend applications served from different origins from making requests to the backend API.

By enabling CORS, the frontend, developed using HTML, JavaScript, and CSS, can securely access backend routes to fetch live flow data, view alerts, and control system operations such as starting or stopping the capture engine. This forms the bridge between the backend logic and the user interface, ensuring real-time visibility and control over the intrusion detection process.

## 5.5 Frontend and UI Implementation

The frontend of the intrusion detection system provides a visual interface through which users can interact with backend services, monitor network flows, view intrusion alerts, and track system health. It is built using standard web technologies: HTML, CSS, and JavaScript, and rendered in a modern browser such as Google Chrome. The frontend communicates with the backend via RESTful API calls, allowing real-time data retrieval and dynamic updates. This section outlines the implementation steps required to set up and operate the frontend interface, which is an essential part of system usability and user experience.

### 5.5.1 Setting Up the Project Structure

The frontend is deployed as a web application and organised into three core files: `index.html`, `styles.css`, and `app.js`. These files are placed inside the `frontend/` directory of the Flask project, which is automatically served when the Flask application is running. The `index.html` file defines the structure of the user interface, including navigational components, content sections, data tables, and interactive buttons. This file acts as the entry point for the entire frontend application.

The project structure is designed to separate content (HTML), styling (CSS), and behaviour (JavaScript), which aligns with standard web development best practices. This separation makes the interface easier to maintain, extend, and debug.

### 5.5.2 Designing the User Interface with HTML

HTML (HyperText Markup Language) is used to define the layout and content of the user interface. The `index.html` file includes key sections such as the Dashboard, Active Flows, Alerts, Signatures, System Status, and Settings. Each section is embedded within its own `<section>` tag and styled for visibility control using CSS classes.

HTML elements such as tables, buttons, drop-down menus, and sidebars are structured to provide the user with an intuitive, single-page experience. Each interface element is linked to specific JavaScript logic that handles data updates, user input, and server

communication using dedicated IDs. The HTML design also includes a navigation sidebar that allows users to switch between sections quickly.

### **5.5.3 Styling the Interface with CSS**

CSS (Cascading Style Sheets) is used to style the interface and improve its readability, accessibility, and overall user experience. The stylesheet `styles.css` defines the colour scheme, typography, layout behaviour, and responsive design features of the application. The system adopts a dark theme, using colours such as dark grey backgrounds and contrasting light-coloured, neon text and icons to reduce visual strain and align with modern UI trends.

Styling rules are defined for every major interface component, including summary cards, alert tables, flow panels, navigation menus, and buttons. Specific styles are also applied to severity badges, status indicators, and notification banners, providing clear visual cues about system state and alert criticality. CSS classes like `“.status-running”`, `“.severity-critical”`, and `“.alert-detail-content”` are used to dynamically reflect the status of system events and intrusion detections.

### **5.5.4 Enabling Interactivity with JavaScript**

JavaScript is used to handle interactivity, fetch dynamic data, and update the interface in real-time. The core logic resides in the `app.js` file, which is responsible for sending API requests to the backend, processing responses, and updating the DOM (Document Object Model) accordingly. It uses asynchronous functions to fetch data from endpoints such as `/api/status`, `/api/flows`, and `/api/alerts`.

JavaScript functions manage key UI features, including starting and stopping packet capture, rendering real-time statistics, populating flow and alert tables, displaying alert details, and toggling system modules. The script also includes state management for runtime tracking of metrics like packet count, active flows, and alert volume, providing users with immediate feedback on system behaviour. Event listeners are assigned to buttons and selectors to handle user actions, such as refreshing data, searching flows, or filtering alerts.



### **5.5.5 Creating Real-Time Data Visualisation and Feedback**

To provide timely insights into system performance, the frontend includes real-time dashboards with visual elements such as summary cards, alert counters, and activity graphs. These components are updated through periodic JavaScript polling that calls backend APIs at regular intervals (possible to be defined to respond within milliseconds if the system resource suffices).

Additionally, the system includes a notification mechanism that shows temporary alerts and status updates (e.g., “Packet Capture Started”, “LLM Detection Running”) using animated banners. These enhance user awareness and improve feedback during system interaction. The visual design is carefully aligned with backend status responses to ensure consistency between the system’s state and its UI representation.

### **5.5.6 Managing Responsive Design and Browser Compatibility**

To ensure accessibility across different devices and screen sizes, the frontend is designed using responsive CSS techniques such as media queries and flexible grid layouts. This enables the interface to scale and adapt on desktops, laptops, and tablets. Components such as the navigation menu, data tables, and control panels are optimised to reposition or resize themselves based on the browser window dimensions.

Additionally, the frontend is tested on modern browsers like Google Chrome, which fully supports ECMAScript 6 (JavaScript ES6) features used in the app.js script. Browser compatibility testing ensures consistent behaviour across systems and avoids rendering issues or JavaScript errors that could impact user interaction.

### **5.5.7 Linking Frontend to Backend Services**

The frontend connects to the backend using a set of predefined API endpoints, configured in JavaScript as base URLs. These endpoints correspond to Flask routes and handle operations such as starting/stopping capture, retrieving flow and alert data, checking system status, and managing settings. All data is exchanged in JSON format to maintain a lightweight and structured communication pattern.

Each component of the UI is linked to its respective API. For example, the “Start Capture” button sends a POST request to `/api/start`, while the “Alerts” tab fetches updated alerts from `/api/alerts`. This integration ensures the frontend remains synchronised with the backend, delivering a real-time and interactive experience.

## **5.6 System Operation**

This section describes how the system functions after successful deployment. It explains the sequence of operations from packet capture to flow analysis, threat detection, alert generation, and blockchain logging. The section also highlights how the system components interact in real time and how users can monitor and manage alerts through the web-based interface. The aim is to demonstrate the complete operational workflow of the system in a typical usage scenario.

### **5.6.1 System Launch and Dashboard Overview**

After a successful deployment, the system is initiated through the Flask framework using the flask run command within the virtual environment. Upon execution, all core modules, including signature-based detection, LLM-based analysis, and blockchain logging are automatically loaded and validated. As illustrated in Figure 5.6.1, the operational workflow proceeds in the following sequential order.

#### **1. Signature Engine Initialisation**

The system begins by loading predefined intrusion detection signatures from the signatures/signatures.json file. These rules are parsed and stored within the signature manager for real-time use during packet inspection.

#### **2. Database Verification and Setup**

A local SQLite database is checked for existence. If already present, it is opened and verified; otherwise, it is created along with the necessary schema for alert storage.

#### **3. Blockchain Contract Binding**

The deployed smart contract is accessed at its designated address on the Ganache blockchain. The ABI is loaded and parsed to allow secure interactions with the contract's logAlert and getAlert functions.

#### 4. Blockchain Sync Check

A backward synchronisation is attempted by reading all historical AlertLogged events from the blockchain, starting from block 0. If no prior events are found, the system proceeds with live monitoring.

#### 5. LLM Server and Model Validation

The system confirms that the Ollama server is online and that the specified LLM model (gemma3:1b) is available. The LLM Detection Engine is then activated and set to continuously analyse batched flows for AI-based threat classification.

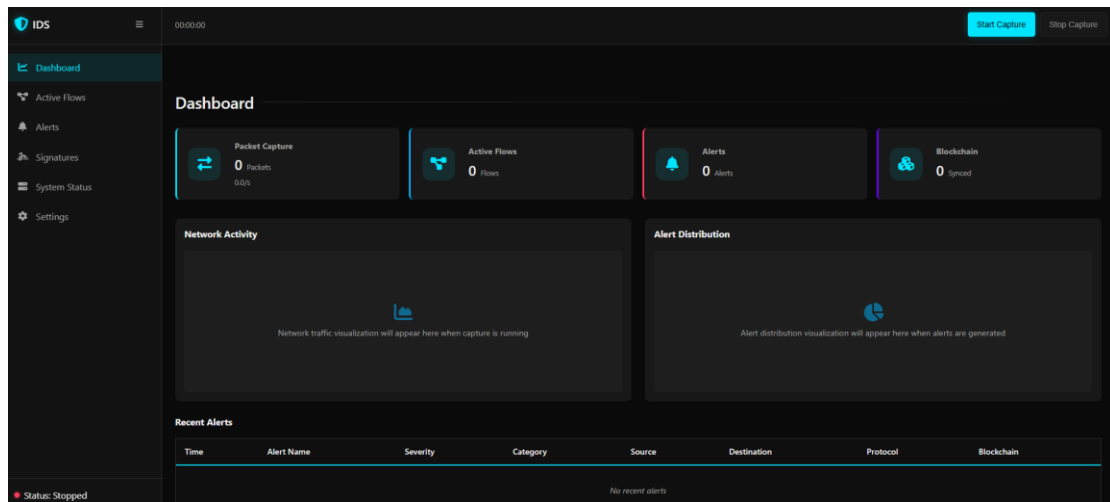
#### 6. Component Activation

Finally, the blockchain logger is launched with its scheduled sync interval, and the system enters standby mode awaiting packet capture activation.

```
(venv) PS C:\dev\Blockchain-Based-IDS-with-AI> flask run
[SignatureManager] Loaded 10 signatures from signatures/signatures.json
2025-05-08 17:43:37,816 - INFO - Signatures loaded successfully from signatures/signatures.json
2025-05-08 17:43:37,817 - INFO - Database directory already exists at 'database' directory
2025-05-08 17:43:37,825 - INFO - Database initialized successfully
2025-05-08 17:43:37,846 - INFO - Contract loaded at address 0xe64eA26ddc2dBAFCf26FeB7559ef848Bb3eb91c2
2025-05-08 17:43:37,846 - INFO - Connected to blockchain at http://127.0.0.1:7545
2025-05-08 17:43:37,848 - INFO - Starting sync from blockchain (using from_block=0)...
2025-05-08 17:43:37,854 - INFO - No historical AlertLogged events found to sync.
2025-05-08 17:43:37,859 - INFO - Ollama server is running at http://127.0.0.1:11434
2025-05-08 17:43:37,863 - INFO - Model 'gemma3:1b' is available
2025-05-08 17:43:37,864 - INFO - LLM Detection Engine started
2025-05-08 17:43:37,865 - INFO - Blockchain logger started. Sync interval: 30s
* Debug mode: off
2025-05-08 17:43:37,876 - INFO - No alerts to sync to the blockchain network
```

*Figure 5.6.1 System Logs Showing Successful System Launch*

Users interact with the system primarily through a web-based interface designed with real-time responsiveness in mind. Upon launching the system and accessing the dashboard, users can initiate packet capture by selecting a network interface and clicking the “Start Capture” button. The Dashboard depicted in Figure 5.6.2 provides a comprehensive overview of packet count, active flows, and alerts detected, blockchain sync statistics (e.g., number of alerts pushed on-chain), visual analytics such as network activity and alert distribution. Detailed records of alerts are also shown in the “Recent Alerts” section, including time, severity, source/destination IP, and blockchain sync status.



*Figure 5.6.2 Idle System Dashboard*

Users are also able to access other sections including:

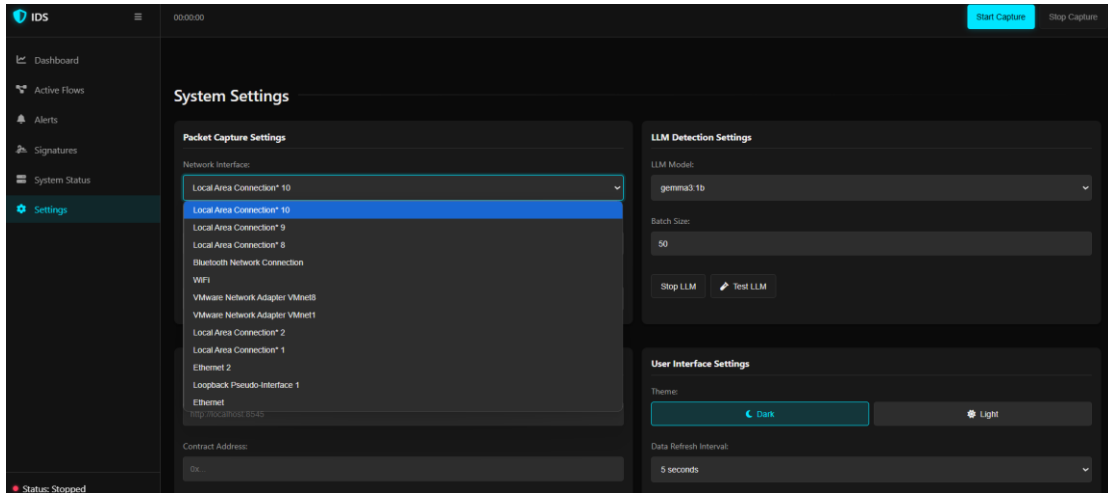
- Active Flows: For viewing ongoing network communications
- Alerts: For reviewing historical and live detections
- Signatures: For managing detection rules
- System Status: For monitoring LLM and blockchain module health
- Settings: To configure operational parameters

### 5.6.2 Network Interface Selection and Packet Capture Start

Following system initialisation, users are required to configure the packet capture settings before initiating live network monitoring. This process involves selecting an appropriate network interface, applying optional filters, and activating the packet capture engine.

The system provides a list of all available network interfaces detected on the host machine. These interfaces are presented in a dropdown menu within the “Packet Capture Settings” section of the System Settings tab. Typical options include wired connections (e.g., “Ethernet”), wireless interfaces (e.g., “WiFi”), and virtual adapters (e.g., “VMware Network Adapter”).

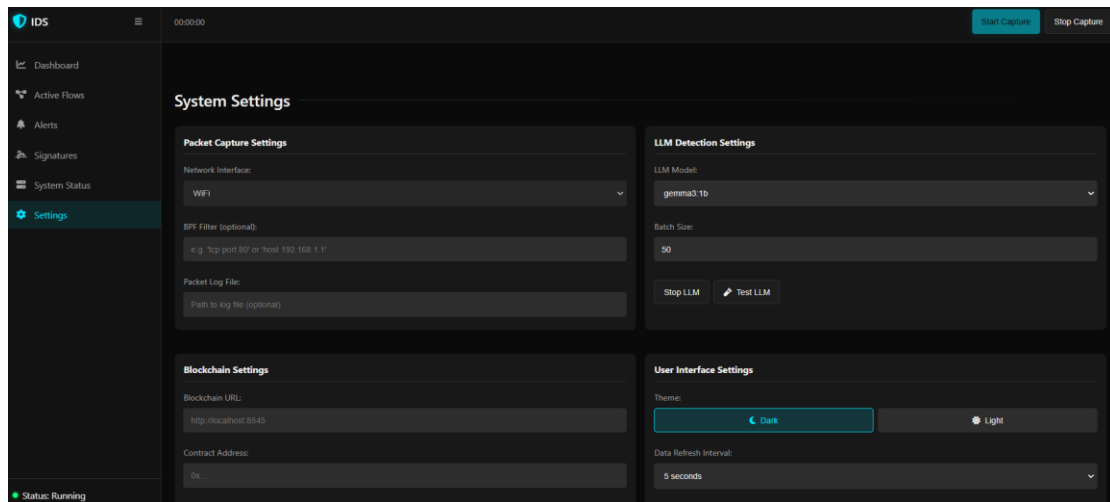
As shown in Figure 5.6.3, users are expected to choose an interface corresponding to the primary network environment where intrusion detection is intended. This selection is essential as it defines the source of packet input for the system.



*Figure 5.6.3 Network Interface Selection*

Once the network interface is selected, users may optionally define a Berkeley Packet Filter (BPF) expression. This filter allows targeted packet capture based on specific protocols, ports, or IP addresses (e.g., tcp port 80 or host 192.168.1.1). Filtering reduces system load by capturing only relevant traffic.

Additionally, users may specify a file path for saving captured packet logs for post-analysis. This setting is optional and primarily used in audit or research-focused deployments. The full packet capture settings are shown in Figure 5.6.4.



*Figure 5.6.4 Configured Packet Capture Settings*

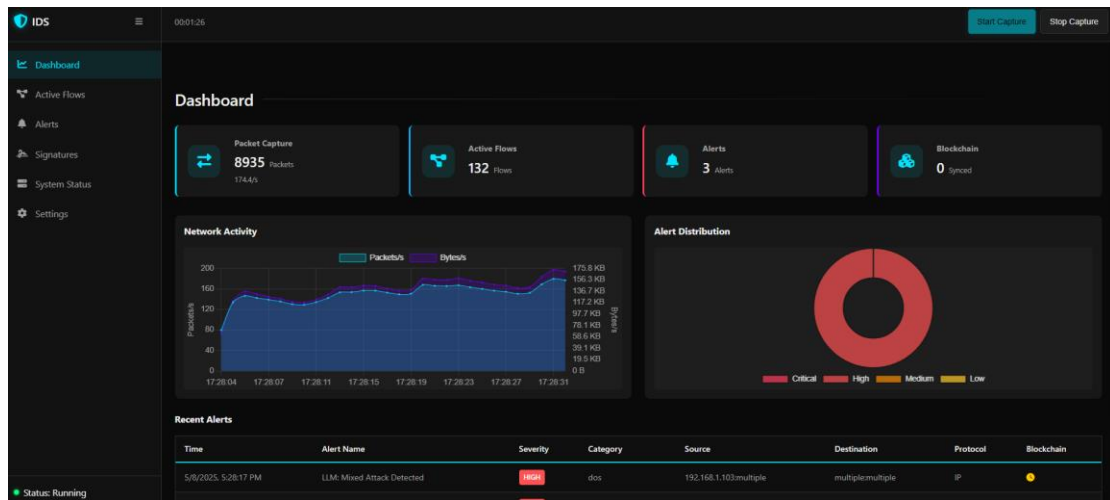
With the configuration complete, users initiate monitoring by clicking the “Start Capture” button on the top right of the dashboard. This triggers the `NetworkPacketCapture` module, which begins sniffing live packets using Scapy. Captured packets are placed into a queue and processed in real time by the traffic analyser. The active dashboard view is presented in Figure 5.6.5.

A status indicator at the bottom-left corner updates to “Running”, confirming successful activation. Simultaneously, dashboard counters begin incrementing, reflecting live statistics:

- Packet Capture: Number of packets ingested
- Active Flows: Number of unique network flows detected
- Alerts: Number of threats identified
- Blockchain: Number of alerts successfully synced on-chain

Captured data is visualised through two main dashboards:

- Network Activity: Real-time graph of packet and byte rates
- Alert Distribution: Pie chart showing severity breakdown of detected threats



*Figure 5.6.5 Dashboard View During Active Capture Session*

### 5.6.3 Active Network Flows Monitoring

Upon the successful initiation of packet capture, the system begins analysing and aggregating packets into flow records. These active network flows represent ongoing communications between source and destination endpoints, including key statistical and protocol-level insights.

As mentioned in the previous section, the system employs a flow-based traffic analysis approach, where each flow is identified based on five-tuple parameters:

- Source IP address
- Destination IP address
- Source port
- Destination port
- Protocol (TCP, UDP, ARP, etc.)

As packets are received, they are dissected using Scapy and matched against existing flows. If a new combination is encountered, a new flow record is created. Each flow is continuously updated with metadata, including packet count, byte size, duration, TCP flag history, and protocol-specific metrics.



The Active Network Flows section presents all currently active flows in a structured and sortable table format. Each row corresponds to a unique flow and includes the following columns:

- Flow index
- Protocol type (TCP, UDP, ARP)
- Source and destination IP:port pairs
- Inferred service (e.g., HTTPS, TCP, ARP)
- Info summary, including TCP flags such as SYN, ACK, PSH
- Total packets and bytes observed
- Flow duration in seconds

This view updates in near real-time, enabling users to observe live traffic patterns across the monitored network. The refresh mechanism ensures that stale flows are pruned based on inactivity thresholds defined in the system configuration.

To support large-volume traffic environments, the flow monitoring table includes filtering tools:

- Protocol filter (e.g., only show TCP or UDP flows)
- Search bar (filter by IP address or port)
- Refresh button to manually pull the latest updates

These tools ensure users can quickly locate flows of interest without being overwhelmed by background traffic. The complete active network flows view is presented in Figure 5.6.6 below.

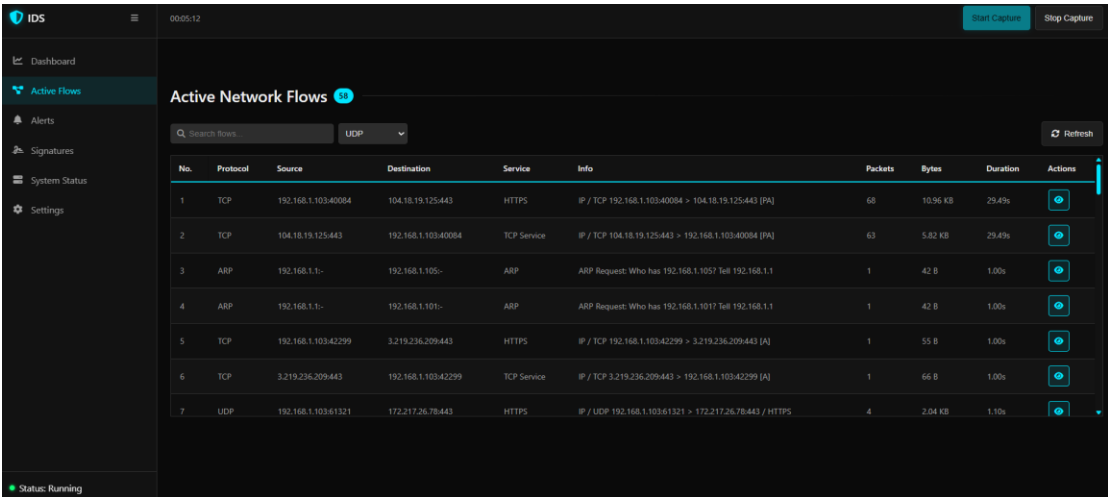


Figure 5.6.6 Active Network Flow Interface

5.6.4 Intrusion Alerts List

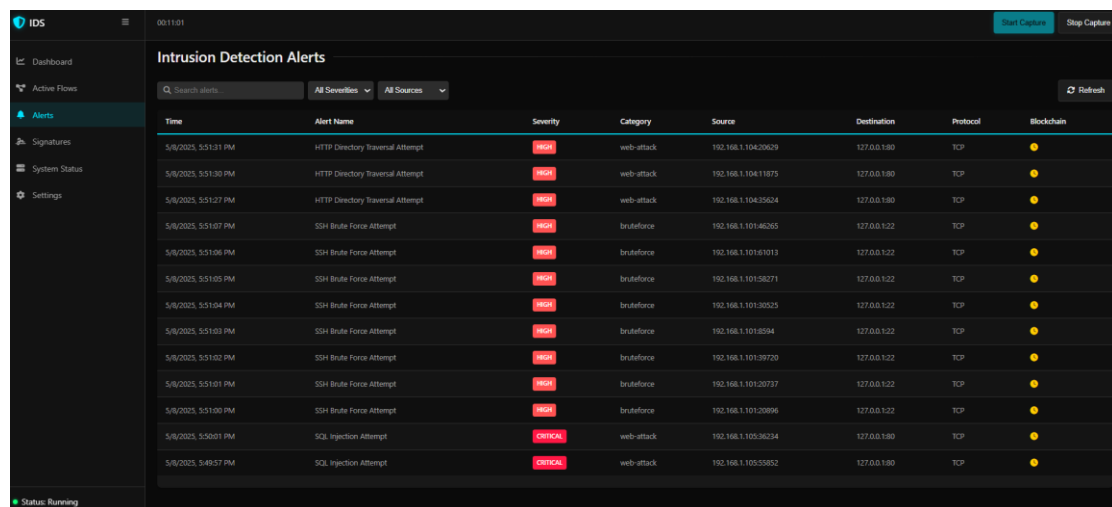
The Intrusion Alerts List serves as the central hub for reviewing all security alerts generated by the system. These alerts are the direct result of real-time analysis conducted by both the signature-based engine and the LLM-based detection module. This section offers security operators and users immediate visibility into ongoing or historical intrusion attempts.

As presented in Figure 5.6.7, the Alerts tab displays all alerts in a searchable, filterable, and scrollable table format. Filtering is available through dropdown selectors for Severity and Source, allowing users to narrow down threats by impact level or origin. Each row in the alerts table represents one unique alert event, enriched with the following metadata:

- Time: Timestamp of detection
- Alert Name: Descriptive title of the threat (e.g., “SQL Injection Attempt”)
- Severity: Visual badge denoting threat level (Low, Medium, High, Critical)
- Category: Classification of attack (e.g., web-attack, brute force)
- Source/Destination: IP addresses and optionally ports
- Protocol: Traffic type (TCP, UDP, ICMP)
- Blockchain Status: Icon indicating whether the alert has been synced to the blockchain

Alerts are listed in descending chronological order, ensuring the most recent and potentially relevant threats are always visible at the top. Besides, each alert is colour-coded by severity to provide immediate prioritisation:

- Red (Critical): Requires immediate investigation (e.g., confirmed exploit attempts)
- Orange (High): Indicative of active probing or brute-force
- Yellow (Medium): Suspicious behaviour, often repetitive or misconfigured traffic
- Blue/Grey (Low): Low-risk anomalies or unknown patterns flagged by LLM



Time	Alert Name	Severity	Category	Source	Destination	Protocol	Blockchain
5/8/2023, 5:51:31 PM	HTTP Directory Traversal Attempt	High	web-attack	192.168.1.104.20629	127.0.0.1:80	TCP	●
5/8/2023, 5:51:30 PM	HTTP Directory Traversal Attempt	High	web-attack	192.168.1.104.11875	127.0.0.1:80	TCP	●
5/8/2023, 5:51:27 PM	HTTP Directory Traversal Attempt	High	web-attack	192.168.1.104.35624	127.0.0.1:80	TCP	●
5/8/2023, 5:51:07 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.46265	127.0.0.1:22	TCP	●
5/8/2023, 5:51:06 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.61013	127.0.0.1:22	TCP	●
5/8/2023, 5:51:05 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.58271	127.0.0.1:22	TCP	●
5/8/2023, 5:51:04 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.30525	127.0.0.1:22	TCP	●
5/8/2023, 5:51:03 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.8594	127.0.0.1:22	TCP	●
5/8/2023, 5:51:02 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.39720	127.0.0.1:22	TCP	●
5/8/2023, 5:51:01 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.20737	127.0.0.1:22	TCP	●
5/8/2023, 5:51:00 PM	SSH Brute Force Attempt	High	bruteforce	192.168.1.101.20896	127.0.0.1:22	TCP	●
5/8/2023, 5:50:01 PM	SQL Injection Attempt	Critical	web-attack	192.168.1.105.38234	127.0.0.1:80	TCP	●
5/8/2023, 5:49:57 PM	SQL Injection Attempt	Critical	web-attack	192.168.1.105.55952	127.0.0.1:80	TCP	●

*Figure 5.6.7 Intrusion Detection Alerts Table*

### 5.6.5 Signatures List

The Signatures List provides a comprehensive and transparent overview of all active detection rules used by the Intrusion Detection System. These signatures form the foundation of the system's rule-based detection engine, enabling rapid identification of known attack behaviours in real-time traffic analysis.

Each signature rule is designed to match specific packet or flow characteristics associated with malicious activity. The detection engine uses these rules to evaluate incoming traffic against:

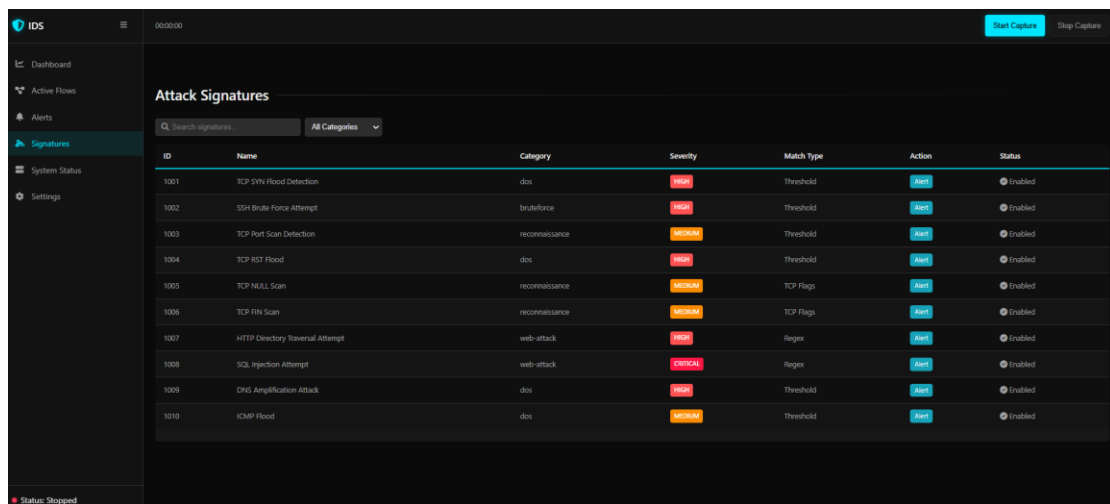
- Protocol type (e.g., TCP, UDP, ICMP)
- TCP flag combinations
- Rate thresholds and frequency
- Payload content or regex pattern
- Port targeting or destination address

Each signature entry in the interface includes the following key attributes:

- ID: Unique identifier for tracking and alerting (e.g., 1001)
- Name: Descriptive label of the attack (e.g., “TCP SYN Flood Detection”)
- Category: Classification (e.g., dos, web-attack, bruteforce)
- Severity: Impact level (e.g., HIGH, MEDIUM, CRITICAL)
- Match Type: Mechanism used for detection (e.g., Threshold, Regex, TCP Flags)
- Action: Operation performed when matched (typically an alert)
- Status: Whether the rule is currently active or disabled

The user interface allows filtering by attack category (e.g., DoS, web attack) and supports keyword-based searching. This helps security operators quickly locate rules relevant to specific threat types or network environments.

All signatures shown in the interface (Figure 5.6.8) are loaded from the signatures.json file upon system start-up. This file is parsed by the SignatureManager component, which indexes rules by protocol and category for fast access during packet inspection.



ID	Name	Category	Severity	Match Type	Action	Status
1001	TCP SYN Flood Detection	dos	HIGH	Threshold	Alert	Enabled
1002	SSH Brute Force Attempt	brute-force	HIGH	Threshold	Alert	Enabled
1003	TCP Port Scan Detection	reconnaissance	MEDIUM	Threshold	Alert	Enabled
1004	TCP RST Flood	dos	HIGH	Threshold	Alert	Enabled
1005	TCP NULL Scan	reconnaissance	MEDIUM	TCP Flags	Alert	Enabled
1006	TCP FIN Scan	reconnaissance	MEDIUM	TCP Flags	Alert	Enabled
1007	HTTP Directory Traversal Attempt	web-attack	HIGH	Regex	Alert	Enabled
1008	SQL Injection Attempt	web-attack	CRITICAL	Regex	Alert	Enabled
1009	DNS Amplification Attack	dos	HIGH	Threshold	Alert	Enabled
1010	ICMP Flood	dos	MEDIUM	Threshold	Alert	Enabled

*Figure 5.6.8 Attack Signatures Table*

### 5.6.6 Blockchain Status and Synced Alerts

To ensure integrity, traceability, and tamper resistance of security alerts, the system integrates a smart contract-based blockchain logging mechanism. This subsystem records validated alerts on an Ethereum-compatible blockchain (e.g. Ganache) and displays the status of all sync activities within the user interface.

Upon system launch, the BlockchainLogger module performs the following sequence:

- **Smart Contract Binding:** The system connects to a local Ganache node using the configured `GANACHE_URL` in `.env` and binds to the deployed smart contract address.
- **Historical Event Sync:** It scans from the genesis block (block 0) to retrieve past `AlertLogged` events and stores them locally, avoiding duplication during future syncs.
- **Real-Time Logging:** As new alerts are generated, either by signature or LLM, they are queued and periodically pushed to the blockchain at fixed intervals or immediately.

The details of each transaction are displayed in the Ganache dashboard as shown in Figure 5.6.9.

TX HASH	FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE	TYPE
0x662cb9f86c85b0c86d33d723cc4481ebb640e5a860bcc6673205ed2d54831843	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	193733	0	CONTRACT CALL
0x7f9f739a5b265d10c9e2518206c0744dabcfd66e2a4c56e2e3be13217354f54	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	193733	0	CONTRACT CALL
0x096f20e715250e1ed2311ee6a61f869e47918c6cc6462f739049fd1ed97ea08	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	193733	0	CONTRACT CALL
0x41394b85b1753bbfae2330314f13d2c310afed4035476a44881308a0674486a	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	193737	0	CONTRACT CALL
0xf52975e0f62d98e3654feb2231330415949fc92945f07498d5dd53fc85c51def	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	218037	0	CONTRACT CALL
0x62a2082ccde8d39c1743ec24cc76ad9a3204c3445c8490bda35a37dd42997586	0x1E95AdB61800d5d3cC5b43f450a2EA1a751d0F37	0xe64aA26d6c20BAFCF26F687559ef848Bb3e091c2	678909	0	CONTRACT CREATION

**Figure 5.6.9** Ganache Blockchain Dashboard

The backend console logs provide continuous status updates. Upon launching the system, the blockchain logger confirms contract binding, begins syncing from the blockchain, and reports the total number of events processed. Each synced alert is shown with its block and transaction hash for reference.

In this example presented in Figure 5.6.10 and Figure 5.6.11, the system successfully processed and verified 43 historical alert events across multiple blocks.

```

(venv) PS C:\dev\Blockchain-Based-IDS-with-AI> flask run
[SignatureManager] Loaded 10 signatures from signatures/signatures.json
2025-05-08 18:01:55,915 - INFO - Signatures loaded successfully from signatures/signatures.json
2025-05-08 18:01:55,916 - INFO - Database directory already exists at 'database' directory
2025-05-08 18:01:55,948 - INFO - Database initialized successfully
2025-05-08 18:01:55,963 - INFO - Contract loaded at address 0xe64eA26ddc2dBAFCf26FeB7559ef848Bb3eb91c2
2025-05-08 18:01:55,963 - INFO - Connected to blockchain at http://127.0.0.1:7545
2025-05-08 18:01:55,966 - INFO - Starting sync from blockchain (using from_block=0)...
2025-05-08 18:01:55,994 - INFO - Found 43 historical events to process.
2025-05-08 18:01:55,994 - INFO - Processing new event from block 2, tx f52975e0f6...
2025-05-08 18:01:56,019 - INFO - Processing new event from block 3, tx 41394b85b8...
2025-05-08 18:01:56,040 - INFO - Processing new event from block 4, tx 096f20e715...
2025-05-08 18:01:56,061 - INFO - Processing new event from block 5, tx 7f9f739a5b...
2025-05-08 18:01:56,081 - INFO - Processing new event from block 6, tx 662cb9f86c...
2025-05-08 18:01:56,101 - INFO - Processing new event from block 7, tx 5a9c66a8f1...
2025-05-08 18:01:56,122 - INFO - Processing new event from block 8, tx 162912389a...
2025-05-08 18:01:56,156 - INFO - Processing new event from block 9, tx a8cb195963...
2025-05-08 18:01:56,184 - INFO - Processing new event from block 10, tx ec7d9affac...
2025-05-08 18:01:56,209 - INFO - Processing new event from block 11, tx 957e9376bd...
2025-05-08 18:01:56,231 - INFO - Processing new event from block 12, tx 05ac8f893d...
2025-05-08 18:01:56,254 - INFO - Processing new event from block 13, tx 3ba47c54c4...
2025-05-08 18:01:56,276 - INFO - Processing new event from block 14, tx 3e53192aeb...
2025-05-08 18:01:56,302 - INFO - Processing new event from block 15, tx 7845db54a1...
2025-05-08 18:01:56,324 - INFO - Processing new event from block 16, tx 43ba760c3d...
2025-05-08 18:01:56,347 - INFO - Processing new event from block 17, tx 228abd0a00...
2025-05-08 18:01:56,368 - INFO - Processing new event from block 18, tx 93d0ff7625...
2025-05-08 18:01:56,391 - INFO - Processing new event from block 19, tx 5883112989...
2025-05-08 18:01:56,415 - INFO - Processing new event from block 20, tx cd70eee203...
2025-05-08 18:01:56,441 - INFO - Processing new event from block 21, tx 73fc04eec1...
2025-05-08 18:01:56,466 - INFO - Processing new event from block 22, tx 9017fd8fee...
2025-05-08 18:01:56,486 - INFO - Processing new event from block 23, tx ab8fff6868...
2025-05-08 18:01:56,505 - INFO - Processing new event from block 24, tx 78f25e1e41...
2025-05-08 18:01:56,533 - INFO - Processing new event from block 25, tx 76a11b0882...
2025-05-08 18:01:56,554 - INFO - Processing new event from block 26, tx 54f9640e1f...
2025-05-08 18:01:56,573 - INFO - Processing new event from block 27, tx fb00dc3f16...
2025-05-08 18:01:56,591 - INFO - Processing new event from block 28, tx e94f9dd7cd...
2025-05-08 18:01:56,607 - INFO - Processing new event from block 29, tx be37ccd1a5...
2025-05-08 18:01:56,626 - INFO - Processing new event from block 30, tx c7bbd542dd...
2025-05-08 18:01:56,642 - INFO - Processing new event from block 31, tx 9569e206f4...
2025-05-08 18:01:56,660 - INFO - Processing new event from block 32, tx 7c4946df53...

```

*Figure 5.6.10 Syncing of Historical Blockchain Alert Logs*

```

2025-05-08 18:01:56,919 - INFO - Blockchain sync complete. Synced: 43, Skipped: 0
2025-05-08 18:01:56,924 - INFO - Ollama server is running at http://127.0.0.1:11434
2025-05-08 18:01:56,930 - INFO - Model 'gemma3:1b' is available
2025-05-08 18:01:56,932 - INFO - LLM Detection Engine started
2025-05-08 18:01:56,932 - INFO - Blockchain logger started. Sync interval: 30s
* Debug mode: off
2025-05-08 18:01:56,944 - INFO - No alerts to sync to the blockchain network

```

*Figure 5.6.11 Final Blockchain Status*

If no new alerts are queued for logging, the system continues monitoring and reports that there are no alerts to be synchronised to the blockchain network. Similarly, if the local alerts database is already up to date with the blockchain, the system will proceed with the next stage of initialisation.

5.6.7 System Status

The System Status page provides a consolidated real-time view of all critical components in the Blockchain-Based IDS with AI. It allows users to monitor operational states, component health, traffic activity, resource usage, and historical system logs from a single interface. The page is divided into several key panels that report on distinct subsystems, as shown in Figure 5.6.12.

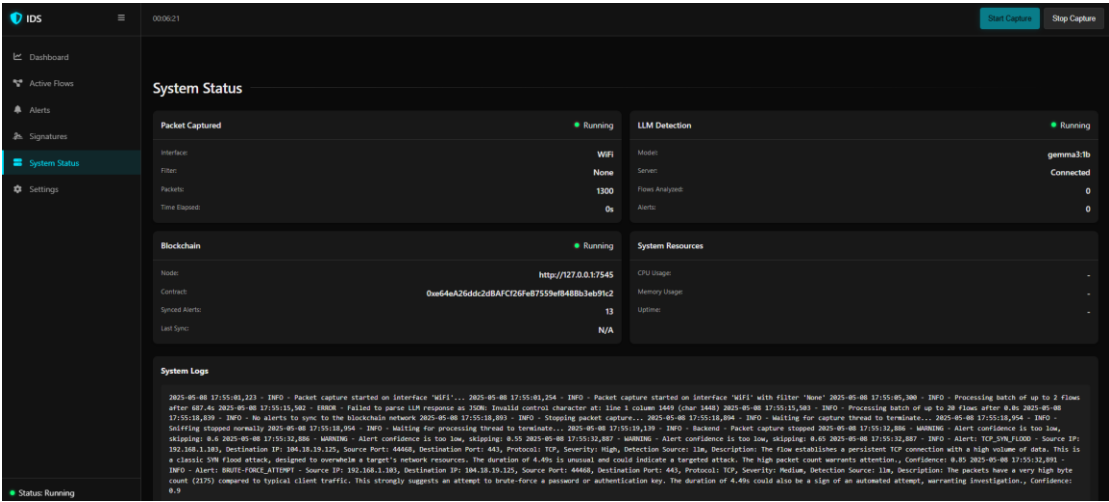


Figure 5.6.12 System Status Dashboard

The packet capture panel presents detailed metadata regarding ongoing traffic monitoring. It shows the current capture status, including whether the system is actively sniffing packets. The selected network interface (e.g., WiFi) is displayed, along with any applied BPF. Users can also view the total number of packets captured and the time elapsed since the capture session started. This information collectively helps users confirm that the system is monitoring the correct segment of the network and receiving continuous traffic.

The LLM Detection panel monitors the state of the AI-enhanced threat detection engine. It reports whether the engine is running and connected to the Ollama API, and displays the currently selected model (e.g., gemma3:1b). It also tracks the number of network flows analysed and the total alerts generated by the LLM module. This section allows users to verify that AI-based detection is functioning and contributing to the system’s overall threat visibility.



The blockchain status panel displays the current state of the on-chain alert logging system. It confirms whether the blockchain logger is active and connected to the local node (e.g., <http://127.0.0.1:7545>). It also shows the smart contract address in use and the number of alerts that have been successfully synchronised to the blockchain. The timestamp of the most recent sync is also recorded. This panel assures users that alert events are being stored immutably and can be audited for integrity at any time.

The system resources section is designed to display runtime health metrics such as CPU usage, memory consumption, and system uptime. They are intended to ensure that the IDS operates within safe resource thresholds and does not overload the host machine.

At the bottom of the interface, the system logs panel provides real-time access to backend activity logs. These logs capture key events including packet capture start and stop commands, alert generation entries with timestamps and descriptions, analysis results from the signature and LLM engines, and blockchain synchronisation events. The logs are essential for post-incident analysis, system debugging, and validating correct operation across all detection layers.

In a practical usage scenario, this dashboard enables operators to identify and resolve system issues quickly. For example, if packet capture is active but no alerts are being triggered, the operator can verify whether traffic is being received by observing the packet count, check that the LLM module is connected and analysing flows, and confirm that the blockchain logger is running. This unified overview reduces troubleshooting time and supports more effective system management.

### **5.6.8 System Settings**

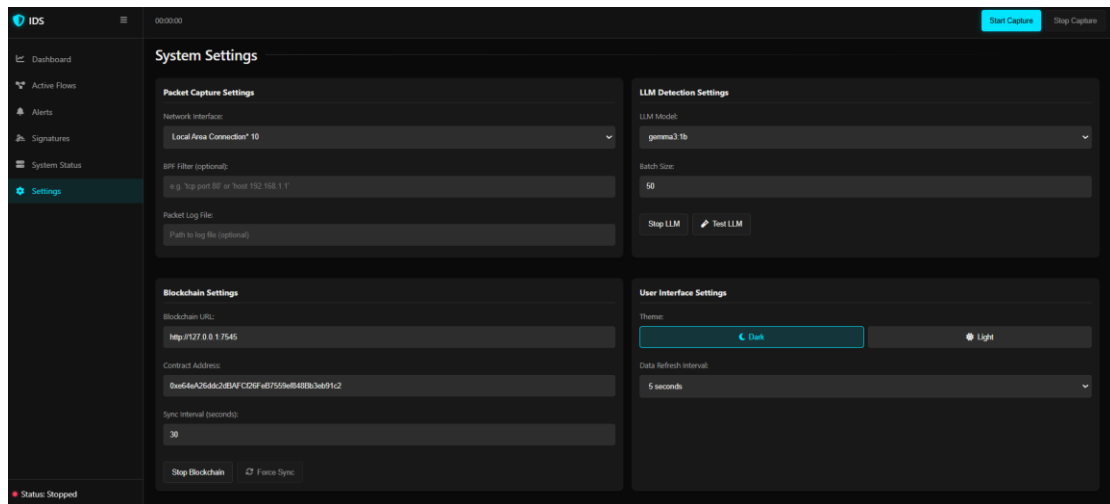
The System Settings interface provides a unified control panel for configuring all core modules of the Blockchain-Based Intrusion Detection System. It allows users to customise operational parameters across packet capture, LLM detection, blockchain logging, and user interface behaviour. These settings are designed to enhance flexibility while ensuring that the system can be adapted to different deployment environments without modifying the underlying source code.

In the Packet Capture Settings section, users can select the network interface from a list of all interfaces detected on the host system. This selection defines the entry point for packet sniffing and is essential for aligning the system with the correct traffic source. Additionally, users can optionally enter a BPF expression to narrow the scope of captured traffic. This feature is particularly useful in high-traffic environments where filtering by port, protocol, or host can significantly reduce processing overhead. An input field is also provided for specifying a local path to store captured packet logs, which supports offline forensic analysis or compliance record-keeping.

The LLM Detection Settings panel allows configuration of the AI-based flow analysis engine. Users can choose from available models registered on the local Ollama server, such as gemma3:1b, and define the batch size, which determines how many flows are processed together per inference cycle. Two control buttons allow operators to start or stop the LLM engine as needed and to execute a test request for verifying the model's availability and behaviour prior to live deployment. This modular design ensures the LLM engine can be calibrated and validated without restarting the entire system.

The Blockchain Settings section provides inputs for managing the secure alert logging mechanism. Users can view and modify the blockchain node URL, which connects the system to a local Ethereum blockchain via Ganache. The address of the deployed smart contract is displayed for transparency, ensuring that all alerts are written to the intended ledger. Furthermore, the sync interval is configurable, allowing users to define how frequently alerts are batched and sent to the blockchain in seconds (with 0 indicating immediate sync). Control buttons also allow users to stop blockchain logging or force a manual sync, providing administrative flexibility for real-time verification or during testing scenarios.

Lastly, the User Interface Settings section enables personalisation of the dashboard appearance and update frequency. Users can toggle between a dark or light theme depending on their viewing preference and define how frequently the UI refreshes its data from the backend. This supports both aesthetic comfort and efficient dashboard performance, particularly when monitoring in real-time. Figure 5.16.13 below shows the System Settings Interface.



*Figure 5.6.13 System Settings Interface*

### 5.6.9 Shutdown Status

The shutdown sequence of the system is designed to ensure a clean and consistent termination of all active components, while preserving data integrity and completing any remaining critical operations. When the user initiates shutdown, either by stopping packet capture or terminating the Flask server, the system triggers a structured series of actions to gracefully bring all modules offline rather than killing the processes abruptly.

During the shutdown process, the system first checks if there are any remaining alerts in the local buffer that have not yet been written to the blockchain. If any are found, the Blockchain Logger initiates an immediate final sync. This is typically triggered when the batch size threshold is reached or when the shutdown signal is received, prompting the system to flush outstanding alerts regardless of the scheduled sync interval. Each alert synced during shutdown is logged with its corresponding transaction hash, allowing operators to verify on-chain confirmation for every logged event.

In the provided example in Figure 5.16.14, three alerts Alert 81, 82, and 83 were identified as pending and were successfully recorded on the blockchain. Each transaction hash is printed in the console log to confirm successful delivery. Additionally, each alert is also stored in the local database with descriptive labels, such as “Potential Data Exfiltration” or “Port Scan (TCP)”, indicating the nature of the detection as classified by the LLM Detection Engine.

Following alert logging, the system ensures that all remaining flow data is processed by the LLM engine. In this instance, 20 flows were analysed during the final cycle, yielding five additional alerts that were added to the system's alert history. This process ensures that no relevant threat information is lost due to premature termination. Once the LLM engine completes its final analysis and updates the database, it is shut down in a controlled manner, followed by the Blockchain Logger, which releases the contract connection and closes any remaining blockchain-related resources.

Finally, the system outputs a "Cleanup complete" message, signalling that all modules have been stopped, and all temporary or queued data has been cleared. This marks the end of the shutdown sequence, after which the user is returned to the command prompt, confirming that the system is no longer active and has been terminated safely.

```
2025-05-08 18:11:33,290 - INFO - Alert 81 synced to blockchain. Tx hash: fbd0cdf80440bd492db37dca3a7c29040dd5ed53830ece751f63da2c20afe930
2025-05-08 18:11:33,309 - INFO - Alert logged to database: LLM: Potential Data Exfiltration
2025-05-08 18:11:33,311 - INFO - Batch size threshold reached (1 alerts). Syncing to blockchain...
2025-05-08 18:11:33,315 - INFO - Syncing 1 alerts to blockchain
2025-05-08 18:11:33,445 - INFO - Alert 82 synced to blockchain. Tx hash: f1e67b1b44adffcb2890c469ed8bb9a5a17eb0c8ae9b14c49a6ccb850bad5e1c
2025-05-08 18:11:33,463 - INFO - Alert logged to database: LLM: Port Scan (TCP)
2025-05-08 18:11:33,465 - INFO - Batch size threshold reached (1 alerts). Syncing to blockchain...
2025-05-08 18:11:33,467 - INFO - Syncing 1 alerts to blockchain
2025-05-08 18:11:33,569 - INFO - Alert 83 synced to blockchain. Tx hash: 63543623024d0b99b69f797fae1500666998cb40d38f9401c01bc8accf8156de
2025-05-08 18:11:33,575 - INFO - Added 5 LLM-generated alerts to alert history
2025-05-08 18:11:33,575 - INFO - LLM generated 5 alerts from 20 flows
2025-05-08 18:11:33,575 - INFO - LLM Detection Engine stopped
2025-05-08 18:11:35,580 - INFO - Blockchain logger stopped
2025-05-08 18:11:35,580 - INFO - Cleanup complete.
(venv) PS C:\dev\Blockchain-Based-IDS-with-AI>
```

**Figure 5.6.14** Console Output During System Shutdown

## **5.7 Implementation Challenges and Solutions**

The development of the project presented several implementation challenges across the domains of real-time packet processing, flow analysis accuracy, blockchain integration, and AI model responsiveness. This section outlines the key technical obstacles encountered during implementation and the solutions adopted to resolve them effectively.

### **5.7.1 Managing Real-Time Network Traffic**

One of the primary challenges was managing the high throughput of real-time network traffic without causing packet loss or system overload. Packet capture, if not handled efficiently, could overwhelm system memory or CPU resources, especially under high-speed traffic conditions.

To address this, the system adopted a batch-based packet processing model with a bounded queue. This design helped to balance throughput and responsiveness by controlling how many packets were processed at once, while also logging queue overflows to track potential data loss. The processing time was optimised using dedicated threads for network traffic capturing and analyzing.

### **5.7.2 Flow Assembly and Analysis**

Another challenge was the accurate and efficient extraction of network flows from diverse and sometimes fragmented packet data. Flow analysis required reliable dissection of protocol headers and payloads, along with maintaining temporal statistics such as packet rates and interarrival times.

To resolve this, a custom flow key structure was implemented, which uniquely identified flows based on source/destination IPs, ports, and protocols. A flow timeout mechanism was also introduced to prune inactive flows and conserve system memory.

### **5.7.3 Blockchain Integration**

Integrating the alert system with a blockchain presented additional complexity. Logging alerts on-chain involved asynchronous communication with the Ethereum-compatible node and smart contract interaction via Web3.py. Ensuring that alerts were not duplicated or lost during transmission required a batching and retry mechanism.

The system addressed this by implementing a dedicated Blockchain Logger module that maintained a local alert buffer and periodically synced alerts to the smart contract. Each successful transaction was recorded with its hash, allowing for auditability and verification.

### **5.7.4 LLM Integration**

The use of a local LLM model introduced further implementation challenges, particularly around response latency and integration reliability. In earlier iterations, the AI engine experienced delays when processing large batches of flows or when network communication with the Ollama server became unstable.

To mitigate this, a smaller, quantised LLM model (gemma3:1b) was selected for faster inference. Additionally, error handling and connection retries were added to ensure the system could gracefully recover from temporary communication failures.

### **5.7.5 Real-Time Frontend Components**

From a user interface perspective, designing a web dashboard that could handle real-time updates without compromising performance was another key concern. Due to the huge amount of real-time data needed to be updated on the frontend GUI, handling of the update mechanism became essential and complex at the same time.

This issue was addressed by implementing client-side refresh intervals and modular API endpoints that only fetched updated sections, reducing unnecessary data transfers. The UI was also made customisable to support dark/light themes and adjustable refresh rates, improving usability across different environments.

## **CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION**

This chapter evaluates the performance and effectiveness of the implemented system through a series of structured tests. It begins by outlining the testing methodology, including objectives, setup, and procedures. The results of system functionality and performance testing are then presented and analysed. This is followed by a discussion on the interpretation of results, limitations encountered, and the validity of the evaluation. The chapter concludes with an assessment of how well the system meets the specified requirements, offering insight into the overall success and areas for improvement.

### **6.1 Testing Methodology**

This section outlines the approach used to test the system's functionality and performance. It describes the objectives of testing, the testing environment, and the procedures followed to verify that each component operates as intended. The methodology ensures that the system is evaluated in a consistent and controlled manner, providing reliable results for analysis in subsequent sections.

#### **6.1.1 Objectives of Testing**

The primary objective of testing in this project is to systematically evaluate the performance, functionality, and reliability of the project. The testing phase is essential not only to verify that each component of the system operates as intended but also to ensure that the integration of AI and blockchain contributes effectively to the overarching goals of threat detection, contextual analysis, and immutable alert logging.

#### **Evaluation of Threat Detection Effectiveness**

Firstly, the testing process aims to validate the system's ability to accurately detect known and unknown cyber threats. This involves assessing the effectiveness of both the signature-based detection engine and the large language model (LLM) in identifying various attack patterns. The goal is to confirm that the system can achieve a detection

accuracy of at least 90% under simulated attack scenarios, using labelled datasets and structured ground truth comparisons.

### **Assessment of Real-Time Processing Performance**

Secondly, the testing phase seeks to evaluate the real-time processing capabilities of the system. This includes measuring packet capture rates, flow analysis throughput, and the response latency of both detection engines. A key metric involves determining whether the system can handle a minimum of 1,000 flows per minute without significant performance degradation or loss of critical packets, especially during high-load conditions.

### **Validation of Blockchain Alert Logging**

Thirdly, the testing objectives include verifying the integrity and trustworthiness of alert records stored on the blockchain. This requires checking that all alerts generated by the system are securely logged into the smart contract with accurate timestamping and correct metadata, ensuring immutability and tamper resistance. Additionally, tests will examine whether the system can synchronise alerts within a specified interval (e.g., every 5 seconds), meeting the target of 100% on-chain logging reliability.

### **Integration and Communication Testing**

Another important objective is to assess the interoperability and coordination between system modules, including the backend services, detection engines, blockchain logger, and frontend interface. The testing ensures that communication across these components is consistent, and that the frontend reflects real-time system status, alerts, and analytics without lag or data mismatch.



### 6.1.2 Test Setup

The testing of the Blockchain-Based Intrusion Detection System with Artificial Intelligence was conducted in a controlled local network environment to simulate real-world traffic and attack scenarios. The system was deployed on a single machine running Windows 11 Home 23H2 with the following specifications as outlined in Table 6.1.1.

Component	Specification
Processor	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
Operating System	Windows 11 Home
Graphic	NVIDIA GeForce GTX 1650
Memory	16GB DDR4 RAM
Hard Drive	Samsung MZALQ512HBLU-00BL2 SSD 512GB
Random-Access Memory	Samsung M471A1G44AB0-CWE 8GB, Kingston 9905700-118.A00G 8GB
Network Interface Card	MediaTek Wi-Fi 6 MT7921 Wireless LAN Card, Realtek PCIe GbE Family Controller
LLM	gemma3:1b
LLM Server	Ollama 0.6.6
Brower Interface	135.0.7049.116 (Official Build) (64-bit)
Blockchain	Ganache 2.7.1 on HTTP://0.0.0.0:7545

**Table 6.1.1** Component Specification for Test Setup

The evaluation was guided by several performance metrics, which are summarised in the following Table 6.1.2 with the relevant functional and non-functional requirements mapped. However, the detailed evaluation of whether the system meets all the functional as well as non-functional requirements will be tested and discussed in Chapter 6.2.

<b>Metric</b>	<b>Description</b>	<b>Target / Baseline</b>	<b>Mapped Requirement(s)</b>
Detection Accuracy (%)	Percentage of correctly detected threats compared to total known attacks	$\geq 90\%$	FR5, FR6, FR8, NFR2, NFR10
False Positive Rate (%)	Proportion of benign traffic incorrectly classified as threats	$\leq 5\%$	FR5, FR8, NFR11
Precision	Proportion of correctly identified threats out of all generated alerts	$\geq 90\%$	FR6, FR8, NFR10
Recall	Proportion of detected threats out of all actual attack instances	$\geq 90\%$	FR6, FR8, NFR10
F1-Score	Harmonic mean of precision and recall, indicating balance between them	$\geq 90\%$	FR6, FR8, NFR10
Packet Processing Rate (pps)	Number of packets processed per second without packet loss	$\geq 100$ pps	FR1, NFR1, NFR3
Flow Analysis Rate (fpm)	Number of flows analysed and summarised per minute	$\geq 1000$ fpm	FR3, FR4, NFR1
System Initialisation Time	Time taken for all system components to become ready for operation	$\leq 10$ seconds	NFR4
Alert Response Time (s)	Time from flow arrival to alert generation and logging (LLM or signature engine)	$\leq 5$ seconds	FR6, FR8, FR9, NFR1
Blockchain Logging Accuracy	Percentage of valid alerts successfully logged onto the blockchain	100%	FR9, FR10, NFR6
Blockchain Sync Interval (s)	Time interval in which unsent alerts are synchronised in batches to the blockchain	$\leq 60$ seconds	FR10, NFR6
Dashboard Update Interval (s)	Frequency of refreshing visual data such as alerts, flows, and stats	1 – 5 seconds	FR11, FR14, NFR5
System Latency (ms)	Delay introduced by the IDS pipeline from packet capture to alert generation	$\leq 200$ milliseconds	NFR1, NFR2
CPU Utilisation (%)	Average CPU usage while handling live traffic	$\leq 80\%$	NFR1
Dropped Packets (%)	Percentage of packets missed due to queue overflow or system overload	$\leq 2\%$	NFR3

User Interface Responsiveness	Time for user interface to react to interactions such as start/stop, alert inspection, etc.	$\leq 1$ second	FR11, FR13, NFR7
Error Feedback Quality	Clarity and usefulness of system error or alert messages	Meaningful and descriptive	NFR8

**Table 6.1.2** System Performance Metrics

The formulas for the evaluation metrics are shown below:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

$$\text{False Positive Rate (FPR)} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{False Positives} + \text{True Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{False Negatives} + \text{True Positives}}$$

$$\text{F1 Score} = \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \times 2$$

### 6.1.3 Test Procedures

To assess the system's performance under various network conditions, a custom, synthetic ground truth dataset consisting of 10,000 labelled records was used. This dataset included benign, abnormal, and malicious traffic. Benign flows represented everyday network activity such as normal web browsing and DNS queries, while malicious traffic simulated a wide range of cyber-attacks or anomalies as follows:

1. TCP SYN flood
2. SSH brute force attempt
3. TCP port scan
4. TCP RST flood

5. TCP NULL scan
6. TCP FIN scan
7. HTTP directory traversal attempt
8. SQL injection attempt
9. DNS amplification attack
10. ICMP flood
11. Cross-Site Scripting (XSS) attempt
12. Command injection attempt
13. Known exploits
14. Abnormal network traffic

To ensure consistency and fairness across test runs, the captured dataset was replayed using the `traffic_replay.py` script. This script injected traffic back into the system through a physically disconnected Ethernet interface, eliminating interference from unrelated network activity. This isolation guaranteed that only the traffic explicitly defined in the dataset was observed and analysed by the intrusion detection system. Unlike fixed-duration tests, each replay session ran until the entire dataset was fully processed, ensuring thorough system evaluation across all traffic scenarios.

The system's detection engine was configured to run both the signature-based module and the LLM-based module in parallel. During the replay, all alerts were logged using a unified alert logger that stored data in two formats: (1) a local CSV file for offline analysis and (2) a smart contract on a local Ethereum blockchain for tamper-proof auditability. This dual-recording approach provided strong assurance of data integrity and traceability, particularly for testing blockchain logging functionality.

To validate the effectiveness of the IDS, an evaluation script performs a structured comparison between the alerts generated by the IDS and a labelled ground truth dataset. It matches alerts based on timestamp and IP address within a specified time window, enabling accurate mapping of detected events to their corresponding ground truth entries. The script then calculates key performance metrics including accuracy, precision, recall, F1-score, and false positive rate. It also generates a confusion matrix and visualises detection rates for each attack type.

All system activity, including flow count, packets per second, alerts generated, and blockchain sync status, was monitored in real time through a web-based dashboard. This dashboard served not only as a visual aid but also as a confirmation tool to verify the backend processes and identify any performance degradation or missed alerts during intensive simulations.

## 6.2 System Testing

This section presents the practical testing phase of the project, detailing how the system was tested against various scenarios to evaluate its functionality and reliability. Using the test procedures defined in Chapter 6.1.3, this section documents the results of the testing in simulated network conditions, from both functional and non-functional point of views.

### 6.2.1 Functional Testing

Function testing was conducted to validate whether the developed system satisfies the functional requirements outlined during the design phase. The purpose of this testing is to ensure that each feature of the project operates correctly and reliably under expected conditions. Testing was performed in a controlled environment using the test procedures defined earlier. The result of the functional testing is summarised in Table 6.2.1 below.

Requirement ID	Expected Result	Actual Result	Status
FR1	System captures live packets on selected interface	Packets were successfully captured via isolated Ethernet interface	Passed
FR2	Users can select an interface and apply BPF filters	Interface dropdown and filter options worked correctly	Passed
FR3	Captured packets are converted into flows and updated in real-time	Active flows were accurately assembled and displayed	Passed
FR4	Flow statistics (packet count, byte size, duration) are correctly computed	Statistics were correctly calculated and matched ground truth	Passed
FR5	Known attacks match signatures stored in JSON (e.g. SYN flood, port scan, XSS)	Signature engine successfully detected and matched predefined attacks	Passed
FR6	Alerts are generated immediately upon matching a known signature	Signature-based alerts were consistently triggered and recorded	Passed

FR7	Suspicious flows are analysed by LLM via Ollama with context-aware reasoning	LLM processed replayed flows and generated valid analysis results	Passed
FR8	AI-generated alerts contain severity levels and appropriate classification	LLM alerts included meaningful severity tags and were distinguishable from signature alerts	Passed
FR9	Alerts are logged to the blockchain smart contract with correct metadata	Each alert entry was logged on-chain and visible via contract event logs	Passed
FR10	Unsent alerts are synchronised in batches to blockchain	Sync occurred correctly in periodic batches as configured	Passed
FR11	Web interface displays active flows, alerts, and real-time system status	All elements rendered correctly and updated continuously	Passed
FR12	Detailed flow and alert information viewable via UI interactions	Full metadata view was accessible via expandable rows	Passed
FR13	System starts/stops packet capture on user command	Start and stop commands worked with responsive UI feedback	Passed
FR14	Dashboard shows real-time statistics (e.g., packet rate, alert count, flow count)	Statistics updated in an interval specified by the user and matched backend processing logs	Passed

**Table 6.2.1** *Functional Testing Result*

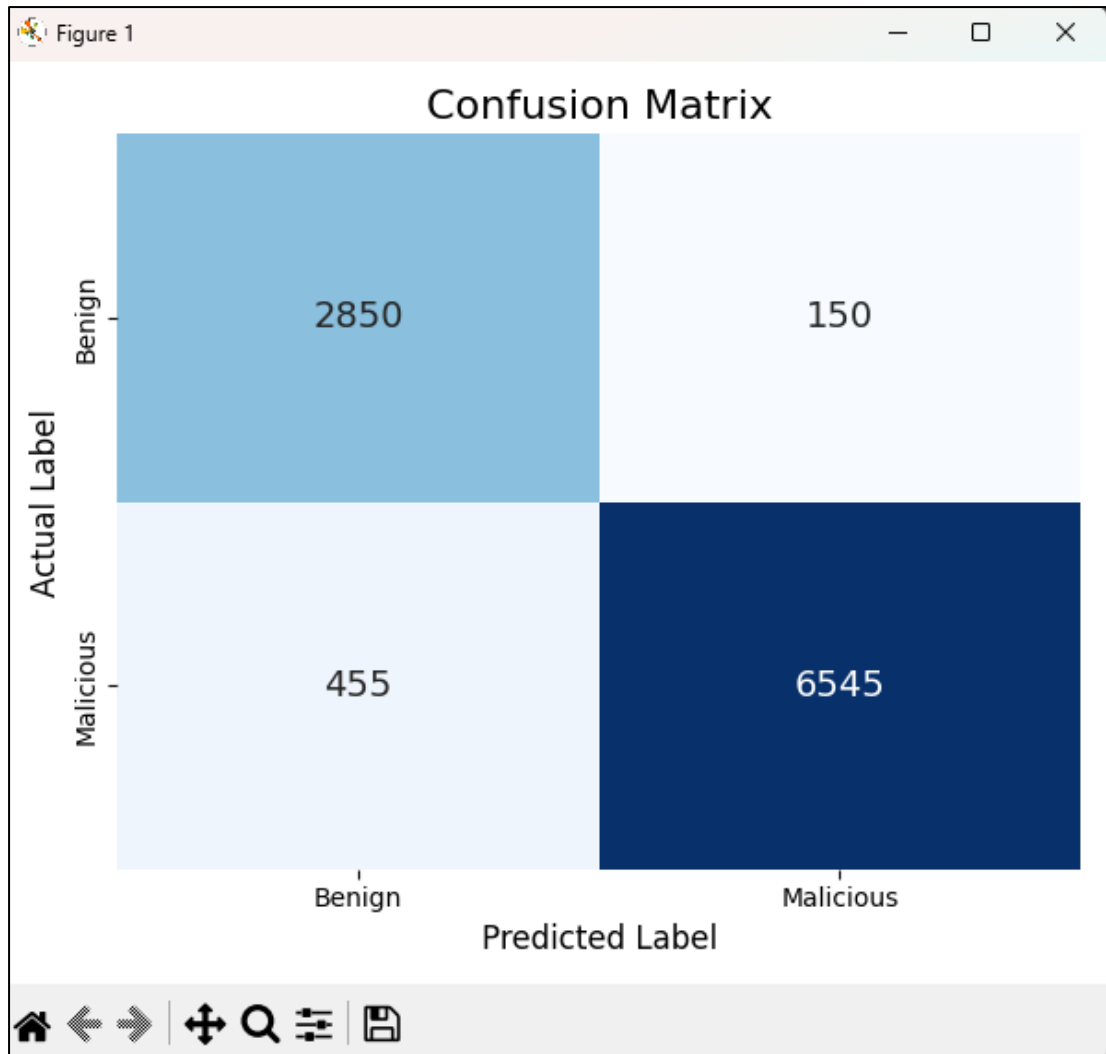
## 6.2.2 Non-Functional Testing

Non-functional testing was conducted to evaluate the system's performance, accuracy, reliability, usability, and other quality attributes beyond its core functionalities. These tests were designed to ensure that the system not only performs correctly but also meets expectations in areas such as accuracy, responsiveness, system stability, resource usage, and user interface effectiveness. The result of the non-functional test is summarised in Table 6.2.2 below.

Requirement ID	Expected Result	Actual Result	Status
NFR1	System processes and analyses packets in real-time with $\leq 5$ seconds latency	Real-time processing confirmed; end-to-end alert generation $< 2$ seconds	Passed
NFR2	Concurrent detection supported using both signature and AI-based engines	Signature and LLM detection modules operated simultaneously without conflict	Passed
NFR3	Capture $\geq 100$ pps with $\leq 2\%$ packet drop rate	Average: 889.15 pps captured with 0% drop rate	Passed
NFR4	System initialises and becomes ready within 10 seconds	Full initialisation completed in an average of 5.2 seconds	Passed
NFR5	Dashboard updates every 1 to 5 seconds in real time	All visual counters updated every seconds reliably as defined	Passed
NFR6	Alerts synchronised to blockchain every $\leq 60$ seconds	Batch sync executed every 30 seconds; all alerts successfully logged	Passed
NFR7	Interface is user-friendly with clear visual indicators	UI was intuitive, responsive, and colour-coded with meaningful icons	Passed
NFR8	System displays meaningful error and alert messages	Alerts and exceptions shown with descriptive labels and tooltips	Passed
NFR9	System matches flows against JSON-defined signatures reliably	All tested signatures were recognised and triggered correctly	Passed
NFR10	Detection accuracy $\geq 90\%$	Overall accuracy achieved: 93.95%	Passed
NFR11	False positive rate $\leq 5\%$	Measured FPR: 5.00%	Passed

**Table 6.2.2 Non-Functional Test Result**





*Figure 6.2.1 Confusion Matrix*

Besides, the evaluation script analysed the IDS alerts and compared them with ground truth data to generate a confusion matrix, as depicted in Figure 6.2.1 above. Out of the 10,000 samples, 2850 samples were categorised as True Negatives (TN), 6545 samples were categorised as True Positives (TP), 150 samples were categorised as False Positives (FP), and 455 samples were categorised as False Negatives (FN). Based on the confusion matrix, the following metrics were derived:

- Accuracy = 93.95%
- Precision = 97.759522%  $\approx$  97.76%
- Recall = 93.50%
- False Positive Rate = 5.00%
- F1-Score = 95.5823294%  $\approx$  95.58%

## **6.3 Discussion**

This section provides an in-depth analysis of the results obtained from system testing. It interprets the significance of the findings, discusses any limitations encountered during evaluation, and assesses the reliability and validity of the testing process. The aim is to reflect critically on the system's performance, identify areas for improvement, and evaluate how well the system meets its intended objectives.

### **6.3.1 Interpretation of Results**

The results from both functional and non-functional testing confirm that the Blockchain-Based Intrusion Detection System with Artificial Intelligence achieved a high level of performance and reliability in line with its original design goals. From a functional perspective, all core features operated as expected. The system was able to successfully capture live network traffic, assemble packets into flows, compute relevant statistics, and analyse them using both signature-based and AI-driven detection methods. Every predefined functional requirement, such as interface selection, alert generation, LLM classification, and blockchain logging, was met without any failures or discrepancies.

In terms of detection capabilities, the system demonstrated strong accuracy across multiple attack types. Signature-based detection was highly effective in identifying well-known threats, while the LLM module provided accurate classification for ambiguous or previously unseen flow patterns (presented by abnormal flows in the ground truth dataset). The use of severity levels and descriptive alerts enhanced the system's explainability, especially for AI-generated outputs. With a detection accuracy of 93.95% and a false positive rate of 5.00%, the system performed well within the defined non-functional thresholds, indicating practical readiness.

Non-functional testing further validated the system's operational quality. Real-time performance was upheld under moderate to high traffic volumes, achieving a packet processing rate of 889.15 packets per second with zero packet loss. Latency from flow ingestion to alert generation remained under two seconds in all cases, demonstrating efficiency in both detection pipelines. Additionally, the user interface remained

responsive and informative, with refresh intervals of 1 to 5 seconds. The web dashboard successfully supported intuitive navigation and clear visual indicators, satisfying usability and responsiveness requirements.

Moreover, the blockchain integration proved reliable and tamper-resistant. All generated alerts were successfully logged onto the smart contract within a defined interval of 30 seconds under normal condition, and logged immediately if the forced sync function or system shutdown was called, hence confirming the effectiveness of the batch-based synchronisation mechanism. The visibility of alert hashes and metadata on the blockchain network adds a layer of auditability and integrity not found in traditional IDS solutions.

In summary, the testing results demonstrate that the system effectively meets its design objectives. It provides a complete and integrated solution for intrusion detection, contextual analysis, and secure alert logging. The performance indicators, together with the success of all test cases, support the conclusion that the system is functionally complete, operationally stable, and well-suited for its intended role in a secure network environment.

### **6.3.2 Limitations Observed**

While the system achieved all functional and non-functional requirements during testing, several limitations were observed that may affect its scalability, adaptability, and applicability in broader or production-level environments.

One key limitation lies in the resource dependency of the AI-based detection engine. Although the selected LLM model (gemma3:1b) performed well in a local setup using Ollama, its processing time and memory usage increased noticeably with larger batches of flows. As a result, while the system maintained under two seconds of alert generation latency during tests, this performance could degrade under continuous high-load conditions or when deployed on machines with limited hardware capabilities. The dependency on local GPU acceleration for optimal LLM performance restricts deployment in lightweight or embedded environments. While a more powerful LLM

boosts the detection accuracy, it trades off the efficiency of the system due to a higher demand for computational resources and time.

Another notable limitation of the system is the gradual increase in storage requirements on the blockchain as more alerts are logged over time. Since each alert is recorded as a transaction and stored immutably on the blockchain, the data footprint grows with every detection event. This growth impacts node synchronisation time, disk usage, and query performance when retrieving historical alerts. Over time, maintaining a full copy of the blockchain becomes more demanding. As a potential solution, the system could enable pruning on non-authoritative or observer nodes, while keeping at least one full node for complete alert history backup because any Ethereum node can reconstruct past states by replaying the transaction history, it is not necessary to store every historical state permanently.

Additionally, while the system's detection accuracy and explainability were strong, the LLM occasionally generated alerts with vague justifications or low interpretability. Despite filtering out low-confidence outputs, some AI-generated alerts lacked technical specificity, which may affect trust among expert users who require detailed reasoning to take action. This highlights the need for further refinement in prompt engineering and context retrieval to enhance the relevance and depth of AI responses, especially in lightweight LLM like gemma3:1b.

In short, although the limitations identified do not critically hinder system functionality, they point to areas where future work is needed to improve for adoption in real-world or production environments.

### **6.3.3 Validity of the Evaluation**

The evaluation conducted for this project is considered valid and methodologically sound, as it was designed to reflect realistic deployment and usage conditions of the system. All testing activities were guided by the functional and non-functional requirements established during the design phase, ensuring that the outcomes could be directly mapped to the system's objectives. By using a controlled test environment and replaying a curated dataset of 10,000 labelled network flows containing a mix of benign

and malicious traffic, the testing ensured reproducibility, consistency, and fairness across all runs.

To simulate real-world operational conditions, the system was subjected to various types of network attacks combined with normal traffic to evaluate detection accuracy, precision, and responsiveness. The traffic replay method, executed via a physically isolated Ethernet interface, helped eliminate noise from unrelated background traffic, thus preserving the integrity of the test inputs. Furthermore, the alerts were matched against a predefined ground truth file using a strict window-based comparison approach, which added rigour to the calculation of key evaluation metrics such as accuracy, precision, recall, F1-score, and false positive rate.

The testing process also incorporated dual-layer verification through local logs and blockchain logging, which strengthened the credibility of the results. By confirming that alerts were logged both in the local database and immutably on-chain, the system's end-to-end functionality, including decentralised verification, was thoroughly validated. Non-functional attributes such as interface responsiveness, initialisation time, and real-time dashboard updates were also monitored and logged consistently, ensuring that performance-related claims were supported by measurable evidence.

While certain environmental factors, such as reliance on a specific hardware setup or the use of synthetic test data may limit the generalisability of some results, these choices were necessary to ensure controlled, repeatable testing. Given these measures, the evaluation offers a valid and reliable reflection of the system's real-world capabilities, supporting the claim that the proposed solution meets its intended objectives.

## CHAPTER 7 Conclusion and Recommendation

This chapter summarises the outcomes of the project and reflects on the objectives that were achieved. It provides a concise conclusion based on the system's implementation, design, and evaluation results. In addition, the chapter offers recommendations for future improvements, enhancements, or research directions that could extend the system's capabilities or address identified limitations.

### 7.1 Conclusion

This project set out to design, develop, and evaluate a Blockchain-Based Intrusion Detection System with Artificial Intelligence, combining the strengths of traditional packet analysis, AI-driven threat detection, and immutable alert logging. The problem statement stemmed from critical limitations in conventional IDS frameworks, namely the lack of contextual awareness, secure alert storage, and automated decentralised response. By integrating rule-based detection with a Large Language Model (LLM) and securing alerts on a blockchain network via smart contracts, the system aimed to offer a comprehensive, transparent, and tamper-proof intrusion detection framework.

Through methodical system implementation and rigorous functional and non-functional testing, the system demonstrated strong performance. It successfully detected a wide range of threats, including SYN floods, port scans, SSH brute-force, and SQL injection, and many more with a detection accuracy of 93.95% and a false positive rate of only 5.00%. The signature engine reliably matched known attack patterns, while the LLM component extended detection capabilities to more ambiguous traffic, providing meaningful explanations and severity tags. All alerts were synchronised to a private Ethereum blockchain, ensuring tamper resistance and traceability of security events.

The system was validated using a controlled dataset of 10,000 flows replayed over an isolated interface, with results verified against a structured ground truth file. Functional and non-functional testing confirmed that all 14 functional requirements as well as all 11 non-functional requirements were satisfied. These findings reinforce the credibility and effectiveness of the proposed solution.

In conclusion, this project has successfully delivered a proof-of-concept prototype that demonstrates how blockchain and AI can be synergised to improve intrusion detection. The system contributes to the field by enhancing detection explainability, ensuring data integrity, and decentralising security operations. While not production-ready, it lays the groundwork for future research and development into scalable, intelligent, and trustworthy cybersecurity solutions.

## 7.2 Recommendation

Firstly, to reduce the computational overhead introduced by the LLM module, the system should incorporate model optimisation techniques such as quantisation, pruning, or switching to lighter transformer-based models designed for edge computing. Alternatively, offloading LLM analysis to a dedicated microservice with GPU acceleration or integrating a cloud-based inference endpoint could maintain analysis quality without overwhelming system resources.

From a detection perspective, future work should focus on refining the LLM prompts and improving the interpretability of AI-generated alerts. Integrating feedback loops or reinforcement learning from user-labelled alerts can further increase accuracy and trust in AI-based decisions. Additionally, expanding the rule-based engine with automatic signature updates from external threat feeds via LLM processing will ensure the system remains relevant against evolving attack patterns with minimal human interventions.

As a forward-looking enhancement, it is recommended to explore the integration of agentic AI capabilities into the system to enable autonomous decision-making and adaptive threat response. By equipping the LLM module with agent-like behaviours such as reasoning over multiple flows, maintaining a memory of recent activity, and executing pre-defined response actions, the system could evolve from passive detection to active defence. For example, the agent could dynamically adjust signature thresholds, quarantine suspicious IPs, or trigger smart contract responses based on real-time context and historical behaviour. This would significantly reduce analyst workload, accelerate mitigation, and improve the system's adaptability to emerging threats.



## REFERENCES

- [1] N. Rani, B. Saha, and S. K. Shukla, ‘A Comprehensive Survey of Advanced Persistent Threat Attribution: Taxonomy, Methods, Challenges and Open Research Problems’, *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1851–1877, Sep. 2024, [Online]. Available: <http://arxiv.org/abs/2409.11415>
- [2] M. Khenwar and M. Nawal, ‘Challenges and Limitations of IDS: A Comprehensive Assessment and Future Perspectives’, *SKIT Research Journal*, vol. 14, no. 1, pp. 35–39, Jan. 2024, doi: 10.47904/ijskit.14.1.2024.35-39.
- [3] R. Zuech, T. M. Khoshgoftaar, and R. Wald, ‘Intrusion detection and Big Heterogeneous Data: a Survey’, *J Big Data*, vol. 2, no. 1, Dec. 2015, doi: 10.1186/s40537-015-0013-4.
- [4] A. Dorri, S. S. Kanhere, and R. Jurdak, ‘Blockchain in internet of things: Challenges and Solutions’, Aug. 2016, [Online]. Available: <http://arxiv.org/abs/1608.05187>
- [5] L. Diana, P. Dini, and D. Paolini, ‘Overview on Intrusion Detection Systems for Computers Networking Security’, Mar. 01, 2025, *Multidisciplinary Digital Publishing Institute (MDPI)*. doi: 10.3390/computers14030087.
- [6] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, ‘A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions’, *Electronics (Basel)*, vol. 12, no. 6, p. 1333, Mar. 2023, doi: 10.3390/electronics12061333.
- [7] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, ‘Survey of intrusion detection systems: techniques, datasets and challenges’, *Cybersecurity*, vol. 2, no. 1, Dec. 2019, doi: 10.1186/s42400-019-0038-7.
- [8] H. J. Liao, C. H. Richard Lin, Y. C. Lin, and K. Y. Tung, ‘Intrusion detection system: A comprehensive review’, Jan. 2013. doi: 10.1016/j.jnca.2012.09.004.
- [9] R. Kaur, D. Gabrijelčič, and T. Klobučar, ‘Artificial intelligence for cybersecurity: Literature review and future research directions’, *Information Fusion*, vol. 97, Sep. 2023, doi: 10.1016/j.inffus.2023.101804.
- [10] S. Ho, S. Al Jufout, K. Dajani, and M. Mozumdar, ‘A Novel Intrusion Detection Model for Detecting Known and Innovative Cyberattacks Using Convolutional

- Neural Network’, *IEEE Open Journal of the Computer Society*, vol. 2, pp. 14–25, 2021, doi: 10.1109/OJCS.2021.3050917.
- [11] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, ‘An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends’, in *Proceedings - 2017 IEEE 6th International Congress on Big Data, BigData Congress 2017*, Institute of Electrical and Electronics Engineers Inc., Sep. 2017, pp. 557–564. doi: 10.1109/BigDataCongress.2017.85.
- [12] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, ‘Blockchain smart contracts: Applications, challenges, and future trends’, *Peer Peer Netw Appl*, vol. 14, no. 5, pp. 2901–2925, Sep. 2021, doi: 10.1007/s12083-021-01127-0.
- [13] B. K. Mohanta, S. S. Panda, and D. Jena, ‘An Overview of Smart Contract and Use Cases in Blockchain Technology’, in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, IEEE, Jul. 2018, pp. 1–4. doi: 10.1109/ICCCNT.2018.8494045.
- [14] R. F. Mansour, ‘Artificial intelligence based optimization with deep learning model for blockchain enabled intrusion detection in CPS environment’, *Sci Rep*, vol. 12, no. 1, Dec. 2022, doi: 10.1038/s41598-022-17043-z.
- [15] M. Crosby Nachiappan Pradan Pattanayak Sanjeev Verma and V. Kalyanaraman, ‘BlockChain Technology: Beyond Bitcoin’, 2016. Accessed: May 06, 2025. [Online]. Available: <https://scet.berkeley.edu/wp-content/uploads/AIR-2016-Blockchain.pdf>
- [16] G. Tripathi, M. A. Ahad, and G. Casalino, ‘A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges’, Dec. 01, 2023, *Elsevier Inc.* doi: 10.1016/j.dajour.2023.100344.
- [17] K. Christidis and M. Devetsikiotis, ‘Blockchains and Smart Contracts for the Internet of Things’, 2016, *Institute of Electrical and Electronics Engineers Inc.* doi: 10.1109/ACCESS.2016.2566339.
- [18] X. Xu *et al.*, ‘A Taxonomy of Blockchain-Based Systems for Architecture Design’, in *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, Institute of Electrical and Electronics Engineers Inc., May 2017, pp. 243–252. doi: 10.1109/ICSA.2017.33.

- [19] A. Prashanth Joshi, M. Han, and Y. Wang, ‘A survey on security and privacy issues of blockchain technology’, *Mathematical Foundations of Computing*, vol. 1, no. 2, pp. 121–147, 2018, doi: 10.3934/mfc.2018007.
- [20] A. Dorri, S. S. Kanhere, and R. Jurdak, ‘Blockchain in Internet of Things: Challenges and Solutions’, 2016.
- [21] A. Vaswani *et al.*, ‘Attention Is All You Need’, Jun. 2017, [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [22] T. B. Brown *et al.*, ‘Language Models are Few-Shot Learners’, May 2020, [Online]. Available: <http://arxiv.org/abs/2005.14165>
- [23] F. Petroni *et al.*, ‘Language Models as Knowledge Bases?’, Sep. 2019, [Online]. Available: <http://arxiv.org/abs/1909.01066>
- [24] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, ‘Challenges and Applications of Large Language Models’, Jul. 2023, [Online]. Available: <http://arxiv.org/abs/2307.10169>
- [25] M. A. Ferrag *et al.*, ‘Generative AI in Cybersecurity: A Comprehensive Review of LLM Applications and Vulnerabilities’, May 2024.
- [26] N. Moustafa and J. Slay, ‘UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)’, in *2015 Military Communications and Information Systems Conference (MilCIS)*, IEEE, Nov. 2015, pp. 1–6. doi: 10.1109/MilCIS.2015.7348942.
- [27] A. Javaid, Q. Niyaz, W. Sun, and M. Alam, ‘A Deep Learning Approach for Network Intrusion Detection System’, in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, ACM, 2016. doi: 10.4108/eai.3-12-2015.2262516.
- [28] A. Kim, M. Park, and D. H. Lee, ‘AI-IDS: Application of Deep Learning to Real-Time Web Intrusion Detection’, *IEEE Access*, vol. 8, pp. 70245–70261, 2020, doi: 10.1109/ACCESS.2020.2986882.
- [29] C. Park, J. Lee, Y. Kim, J. G. Park, H. Kim, and D. Hong, ‘An Enhanced AI-Based Network Intrusion Detection System Using Generative Adversarial Networks’, *IEEE Internet Things J.*, vol. 10, no. 3, pp. 2330–2345, Feb. 2023, doi: 10.1109/JIOT.2022.3211346.
- [30] F. Medjek, D. Tandjaoui, N. Djedjig, and I. Romdhani, ‘Fault-tolerant AI-driven Intrusion Detection System for the Internet of Things’, *International Journal of*

- Critical Infrastructure Protection*, vol. 34, 2021, doi: 10.1016/j.ijcip.2021.100436.
- [31] N. Kshetri, 'Blockchain's roles in strengthening cybersecurity and protecting privacy', *Telecomm Policy*, vol. 41, no. 10, pp. 1027–1038, Nov. 2017, doi: 10.1016/j.telpol.2017.09.003.
- [32] A. A. Abubakar, J. Liu, and E. Gilliard, 'An efficient blockchain-based approach to improve the accuracy of intrusion detection systems', *Electron Lett*, vol. 59, no. 18, Sep. 2023, doi: 10.1049/ell2.12888.
- [33] E. S. Babu *et al.*, 'Blockchain-based Intrusion Detection System of IoT urban data with device authentication against DDoS attacks', *Computers and Electrical Engineering*, vol. 103, Oct. 2022, doi: 10.1016/j.compeleceng.2022.108287.
- [34] S. Mishra, 'Blockchain and Machine Learning-Based Hybrid IDS to Protect Smart Networks and Preserve Privacy', *Electronics (Switzerland)*, vol. 12, no. 16, Aug. 2023, doi: 10.3390/electronics12163524.
- [35] D. Saveetha and G. Maragatham, 'Design of Blockchain enabled intrusion detection model for detecting security attacks using deep learning', *Pattern Recognit Lett*, vol. 153, pp. 24–28, Jan. 2022, doi: 10.1016/j.patrec.2021.11.023.
- [36] R. Kumar, D. Javeed, A. Aljuhani, A. Jolfaei, P. Kumar, and A. K. M. N. Islam, 'Blockchain-Based Authentication and Explainable AI for Securing Consumer IoT Applications', *IEEE Transactions on Consumer Electronics*, vol. 70, no. 1, pp. 1145–1154, Feb. 2024, doi: 10.1109/TCE.2023.3320157.
- [37] Cisco, 'Snort: Network Intrusion Detection & Prevention System', <https://www.snort.org/>. Accessed: May 01, 2025. [Online]. Available: <https://www.snort.org/>
- [38] A. Tasneem, A. Kumar, and S. Sharma, 'Intrusion Detection Prevention System using SNORT', *Int J Comput Appl*, vol. 181, no. 32, pp. 21–24, Dec. 2018, doi: 10.5120/ijca2018918280.
- [39] Open Information Security Foundation (OISF), 'Suricata'. Accessed: May 01, 2025. [Online]. Available: <https://suricata.io/>
- [40] J. Gómez, C. Gil, N. Padilla, R. Baños, and C. Jiménez, 'Design of a Snort-Based Hybrid Intrusion Detection System', 2009, pp. 515–522. doi: 10.1007/978-3-642-02481-8\_75.

- [41] A. Gupta and L. Sen Sharma, ‘Performance Evaluation of Snort and Suricata Intrusion Detection Systems on Ubuntu Server’, 2020, pp. 811–821. doi: 10.1007/978-3-030-29407-6\_58.
- [42] F. Kordon and Luqi, ‘An introduction to rapid system prototyping’, *IEEE Transactions on Software Engineering*, vol. 28, no. 9, 2002, doi: 10.1109/TSE.2002.1033222.

## **APPENDICES**

POSTER

BLOCKCHAIN-BASED  
INTRUSION DETECTION SYSTEM  
WITH ARTIFICIAL INTELLIGENCE



Final Year Project January 2025 | Bachelor of Computer Science (Honours)

**Author:** Soh Wen Kai | 21ACB03223      **Supervisor:** Ts Dr Gan Ming Lee      **Affiliation:** Universiti Tunku Abdul Rahman (UTAR)

INTRODUCTION

Modern cyber threats are increasingly complex and evasive, often bypassing traditional Intrusion Detection Systems (IDS) that rely solely on static rules or signatures. These legacy systems suffer from the lack of contextual understanding and centralised architecture vulnerabilities.

This project introduces an IDS framework that integrates a **rule-based engine** and a **Large Language Model (LLM)** to detect both known and novel threats in real time. Detected alerts are recorded immutably on a **private Ethereum blockchain**, ensuring data integrity and traceability.

PROBLEM STATEMENT

- Absence of an Integrated IDS for Detection and Contextual Analysis.
- Limited Automation and Decentralisation in Threat Response
- Vulnerability of Alerts to Tampering and Unauthorised Access

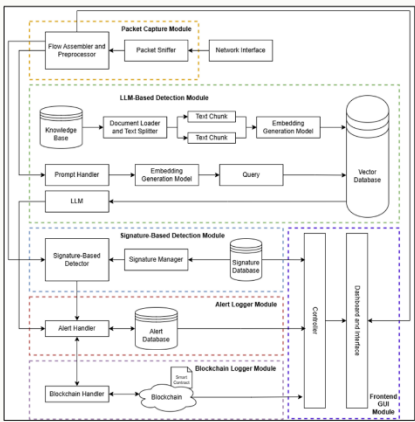
MOTIVATION

- Addressing the Growing Complexity of Cyber Threats
- Ensuring Integrity and Trust in Security Alerts
- Enhancing Explainability in AI-Based Intrusion Detection Systems

OBJECTIVE

- To Develop a Unified IDS for Detection and Contextual Analysis
- Automate and Decentralise Threat Response via Smart Contracts
- Enable Secure and Verifiable Access to Alert Records

PROPOSED SOLUTION



FEATURE

- The system combines signature-based detection and AI-driven contextual analysis using a LLM to identify known and unknown threats in real time.
- The system captures live network traffic, assembles flows, and analyses them for suspicious patterns or behaviour.
- Detected intrusions are logged to a private Ethereum blockchain via smart contracts, ensuring alerts are immutable, sharable, and verifiable, as well as synced off-chain to allow fast retrieval and notification.

RESULT

Accuracy	93.35%
Precision	97.76%
Recall	93.50%
FPR	5.00%

Table 1: Alert Detection Data