# Heimdall: Artificial Intelligence System for Threat Hunting and Detection

**Final Year Project**

**2021-2025**

A project submitted in partial fulfillment of the degree of

BS in Cyber Security

Submitted to

Dr. Zunera Jalil

Department of Cyber Security

Faculty of Computing & Artificial Intelligence (FCAI)

Air University, Islamabad

| | |
|---|---|
| Type (Nature of project) | [✓] Development [ ] R&D |
| Area of specialization | [✓] WebApp [ ] Mobile App<br>[✓] AI based [ ] Embedded System |
| FYP ID | 2406 |

**Project Group Members**

| Sr.# | Reg. # | Student Name | Email ID | *Signature |
|---|---|---|---|---|
| (i) | 211037 | Mubahil Ahmad | 211037@students.au.edu.pk | |
| (ii) | 211059 | Ahsan Ahmed | 211059@students.au.edu.pk | |
| (iii) | 211113 | Syed Ali Zain Ul Abedin Naqvi | 211113@students.au.edu.pk | |

*The candidates confirm that the work submitted is their own and appropriate credit has been given where reference has been made to work of others

## Plagiarism Certificate

This is to certify that, I Mubahil Ahmad S/D of Abdul Majid, group leader of FYP under registration no 211037 at Cyber Security Department, Air University. I declare that my FYP report is checked by my supervisor.

Date:______________ Name of Group Leader: Mubahil Ahmad Signature:________

| | |
|---|---|
| Name of Supervisor: Dr. Zunera Jalil | Co-Supervisor: (Not Allocated) |
| Designation: Professor | Designation: (N/A) |
| Signature: ________________ | Signature: _________________ |

HoD: Dr. Ghalib Asadullah Shah

Signature: _______________

# Heimdall: Artificial Intelligence System for Threat Hunting and Detection

## Change Record

| Author(s) | Version | Date | Notes | Supervisor’s Signature |
|---|---|---|---|---|
| Mubahil Ahmad | 1.0 | 1-Nov-2024 | Initial draft created with system architecture and use cases defined. | |
| Ahsan Ahmed | 1.1 | 20-Nov-2024 | Integrated Kafka producer/consumer logic and ML model inference. | |
| Mubahil Ahmad | 2.0 | 15-Dec-2024 | Added Prometheus monitoring, dashboard, and adaptive retraining. | |
| Ahsan Ahmed | 2.1 | 15-Feb-2025 | Fixed retraining cooldown logic and validated model persistence. | |
| Syed Ali Zain Ul Abedin Naqvi | 2.2 | 24-Mar-2025 | Incorporated QA feedback and test case documentation. | |
| Mubahil Ahmad | 2.3 | 15-Apr-2025 | Refined Flask API endpoints for manual retraining and metrics. | |
| Ahsan Ahmed, Syed Ali Zain Ul Abedin Naqvi | 2.4 | 05-May-2025 | Final QA pass, added change log, and completed stress testing section. | |
| All Authors | 3.0 | 7-May-2025 | Final formatting, bibliography, and report submission prep completed. | |
| | | | | |
| | | | | |

# APPROVAL

**PROJECT SUPERVISOR**

Name: Dr. Zunera Jalil

Date:________________________________ Signature:__________________________

**PROJECT MANAGER**

Date:________________________________ Signature:___________________________

**CHAIR DEPARTMENT**

Date:________________________________ Signature:___________________________

# Dedication

*We dedicate this project to those who have been our unwavering sources of strength and motivation throughout this journey. To our families, who stood beside us with patience, prayers, and quiet sacrifices, your support has been the foundation of our perseverance. To our friends, who encouraged us during the most challenging moments and shared in our victories and setbacks alike, thank you for being part of the experience. And to our teachers and supervisors, whose knowledge, guidance, and encouragement helped shape our thinking and push us toward excellence—this accomplishment is just as much yours as it is ours.*

# Acknowledgements

We hereby declare that the work presented in this Final Year Project is the result of our own efforts and has not been previously submitted or published elsewhere. All external references and sources consulted during the course of this project have been properly acknowledged and cited in accordance with academic standards.

We would like to extend our heartfelt gratitude to our respected supervisor, **Dr. Zunera Jalil**, for her consistent guidance, valuable feedback, and encouragement throughout every phase of the project. Her mentorship played a crucial role in helping us navigate the technical and conceptual challenges of this project.

We are also deeply thankful to our professors and faculty members at **Air University**, whose teachings and support over the years have equipped us with the knowledge and skills necessary to carry out this work. Finally, we appreciate the university for providing access to the resources, facilities, and a collaborative learning environment that made the successful completion of this project possible.

# Executive Summary

*Heimdall is an artificial intelligence-based intrusion detection system designed to perform real-time monitoring and classification of network traffic using machine learning techniques. It captures live packets from the network, extracts relevant features, and streams this data through Apache Kafka, which handles high-throughput, asynchronous communication between system components. The Kafka consumer processes the incoming data and applies pre-trained machine learning models to classify packets as either benign or potentially malicious. Classification results and system metrics are displayed on a Flask-based web dashboard, offering users a clear view of ongoing traffic, detected threats, and system activity. A central aspect of Heimdall is its adaptive learning mechanism, which enables the system to improve over time by retraining its models based on new labeled data. Retraining can be triggered automatically when a certain volume of labeled samples is collected or manually through the dashboard interface. This approach allows Heimdall to adjust to changes in network behavior and maintain relevant detection capabilities as traffic patterns evolve, addressing the problem of concept drift commonly found in static machine learning systems. Prometheus is used for monitoring and collecting performance metrics such as packet throughput, model accuracy, and retraining events, enhancing the system's transparency and maintainability. The system is built using a modular architecture, separating key functions like data capture, classification, visualization, and retraining into independent components, which makes development and updates more manageable. Overall, Heimdall applies adaptive, data-driven learning to enhance the detection of network anomalies in environments where traffic behavior is subject to frequent change.*

# Table of Contents

## List of Figures

# List of Tables

# List of Abbreviations

| | | |
|---|---|---|
| 1 | AI | Artificial Intelligence |
| 2 | ML | Machine Learning |
| 3 | IDS | Intrusion Detection System |
| 4 | DoS | Denial of Service |
| 5 | DDoS | Distributed Denial of Service |
| 6 | SSL | Secure Sockets Layer |
| 7 | TLS | Transport Layer Security |
| 8 | UI | User Interface |
| 9 | API | Application Programming Interface |
| 11 | JSON | JavaScript Object Notion |
| 12 | TCP | Transmission Control Protocol |
| 13 | IP | Internet Protocol |
| 14 | NIC | Network Interface Card |
| 16 | SQLi | Structured Query Language Injection |
| 17 | XSS | Cross-Site Scripting |
| 18 | SaaS | Software as a Service |
| 19 | Prometheus | Monitoring and Metrics Toolkit |
| 20 | Grafana | Visualization and Analytics Platform |
| 21 | SDK | Software Development Kit |
| 22 | MSSP | Managed Security Service Provider |
| 23 | SRS | Software Requirement Solution |
| 24 | IDE | Integrated Development Environment |
| 25 | UML | Unified Modeling Language |
| 26 | SOC | Security Operation Center |
| 27 | SIEM | Security Information and Event Management |

# Chapter 1
# Introduction & Background

# Chapter 1: Introduction

This chapter introduces the core idea behind our project, Heimdall – An AI-Based System for Real-Time Threat Detection. It outlines the cybersecurity problems we set out to solve, the motivation that inspired the solution, and the key challenges encountered along the way. The chapter also explains how current systems fall short in defending against modern cyber threats, presents our proposed solution, and provides a clear breakdown of the project plan, roles, and deliverables.

## 1.1. Background

With the explosive growth of digital transformation, cloud computing, and connected devices, modern networks have become far more complex, distributed, and data-intensive than ever before. These changes have significantly expanded the attack surface, giving rise to a surge in cyberattacks ranging from targeted intrusions and data breaches to large-scale denial-of-service (DoS) attacks. Organizations now operate across hybrid infrastructures, where internal networks are linked with cloud services, remote endpoints, and third-party APIs, all of which can be potential entry points for attackers.

Traditional cybersecurity mechanisms - especially **Intrusion Detection Systems (IDS)** - have long relied on **signature-based detection**, where known patterns of malicious activity are encoded into rules. While effective against previously identified threats, these systems are inherently **reactive**. They struggle to keep pace with modern threats such as **zero-day vulnerabilities**, **polymorphic malware**, **encrypted payloads**, and **insider threats**, all of which can bypass fixed rules or evade detection by mimicking legitimate traffic.

To overcome these limitations, the cybersecurity community has increasingly turned to **machine learning (ML)** as a more proactive and intelligent approach. ML-based IDS[1] models analyze historical traffic and learn to recognize the statistical features of normal versus abnormal behavior. These models can detect previously unseen anomalies and adapt to varying patterns in traffic volume, flow structure, and protocol usage. However, deploying ML in real-time environments comes with its own set of challenges. Most notably, these models are trained on static datasets, and as **network behavior evolves**, their accuracy diminishes - this is known as **concept drift**.

Concept drift occurs when the statistical properties of the input data change over time, making previously trained models less reliable. For example, a spike in video conferencing traffic or the sudden use of new communication ports might be flagged as suspicious if the model hasn't seen such behavior during training. In practical terms, this leads to **false positives** and **false negatives**, eroding trust in the IDS and requiring frequent manual retraining by cybersecurity experts.

**Heimdall** was conceived to address this exact gap. It not only uses machine learning to detect threats in real time but also incorporates **adaptive learning** techniques to evolve with the network it monitors. By integrating live data pipelines, retraining mechanisms, and real-time classification, Heimdall ensures that detection accuracy remains high—even as traffic patterns change, new types of behavior emerge, and attackers adapt. Its modular architecture, automatic retraining, and user-friendly dashboard make it an intelligent and future-ready solution for modern network security.

## 1.2. Motivations and Challenges

The main motivation for this project came from the need to build a smarter, more resilient threat detection system—one that could evolve without constant human intervention. Cybersecurity is a high-stakes field, and the longer a threat goes undetected, the greater the potential damage. Heimdall aims to reduce this gap by combining real-time data processing with adaptive learning.

However, building such a system is far from straightforward. One of the biggest technical challenges was ensuring that the system could handle high volumes of traffic without lag. Real-time processing, especially when paired with machine learning inference and data streaming, can strain resources and introduce latency. Another challenge was designing a retraining mechanism that could intelligently decide *when* and *how* to retrain models without human oversight. Finally, providing a clean, usable interface for system monitoring and control added an additional layer of complexity, requiring a balance between backend functionality and frontend usability.

## 1.3. Goals and Objectives

The *Heimdall* project was guided by a blend of technical innovation and strategic foresight. At its core, the goal was to develop a robust and intelligent intrusion detection system that not only identifies a wide spectrum of network-based threats in real time but also evolves over time to remain effective against new and emerging attack patterns. From a strategic perspective, the system was designed with modularity, scalability, and extensibility in mind—making it suitable for integration into enterprise, academic, and cloud-based environments with minimal overhead.

The key technical and strategic objectives of the project were as follows:

- **To design and implement a real-time network intrusion detection system** powered by machine learning classifiers capable of detecting various types of cyber threats such as DoS, brute force, botnets, and web-based attacks[2].
- **To establish a Kafka-based**[3] **data streaming architecture** that decouples traffic capture from processing, ensuring asynchronous, fault-tolerant, and scalable packet handling.
- **To develop an adaptive learning mechanism** that enables automatic model retraining based on labeled data accumulation or a decline in classification performance, ensuring the system remains effective despite evolving network behavior (i.e., concept drift).
- **To create an interactive, web-based dashboard using Flask**[4]**,** allowing users to monitor threats in real time, review system logs, visualize prediction trends, and manually control model retraining.

**To integrate Prometheus for observability and metrics tracking**, enabling system administrators to monitor packet processing rates, prediction accuracy, retraining events, and system health through a standardized and extensible metrics interface.

## 1.4. Literature Review/Existing Solutions

1. **Artificial Intelligence for Real-Time Threat Detection and Monitoring – Steven W. Tolbert (Florida Atlantic University)**

   Tolbert's research [5]focuses on the increasing need for intelligent systems capable of detecting cyber threats in real time, as traditional signature-based systems fall short in rapidly evolving threat landscapes. The proposed framework leverages artificial intelligence to not only automate the detection process but also introduce adaptability through self-learning models. Unlike static intrusion detection systems (IDS), this framework incorporates data-driven decision-making algorithms that continuously monitor network traffic and learn from patterns over time. The system is designed to handle varying traffic conditions and detect zero-day threats without prior knowledge or manually crafted rules. A major emphasis is placed on maintaining detection accuracy while minimizing false positives, which is achieved by integrating real-time monitoring modules with machine learning models that adapt to traffic behavior dynamically. The study advocates for AI as a replacement for static defense mechanisms, pointing out its capacity to learn from both benign and malicious behaviors to improve threat identification over time. The paper serves as a strong foundational reference for adaptive, real-time IDS development and underscores the need for continuous model updates to handle novel attack vectors.

2. **Smart Detection: An Online Approach for DoS/DDoS Attack Detection Using Machine Learning – Lima Filho et al. (2019)**
   This paper[6] presents a machine learning-based system for the online detection of Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks—some of the most common and damaging forms of cyberattacks. The researchers developed a lightweight yet effective framework that processes real-time traffic in a streaming environment, with models trained on four modern benchmark datasets. The system achieved a detection rate exceeding 96%, maintaining both high precision and a low false alarm rate, even when analyzing only 20% of the total network traffic. This level of efficiency is attributed to the use of sampling and optimized classifiers, which ensure the system can handle large volumes of traffic without degradation in performance. Importantly, the research highlights the need for maintaining detection accuracy without overwhelming system resources—a key concern for live deployment in both corporate and institutional networks. The findings strongly support the feasibility of ML-based online detection and emphasize the potential of scalable models that operate effectively in real-world, resource-constrained environments.

3. **Real-Time Intrusion Detection via Machine Learning Approaches – CEUR Workshop Proceedings (2023)**
   The ReTiNA-IDS framework introduced in this paper demonstrates how machine learning models can be applied in a practical, simulated network environment using the GNS3 emulator. [7]The study employs Random Forest and Multi-Layer Perceptron (MLP) models to detect various forms of intrusions within live network traffic. Key contributions

include the implementation of an end-to-end pipeline that captures packets, extracts features, classifies threats, and logs outcomes—all in real time. The system is tested for latency, classification accuracy, and system throughput to ensure it can be used in production-grade environments. By validating its performance in a simulated yet realistic environment, the study makes a significant step toward bridging the gap between theoretical IDS models and deployable systems. It also highlights deployment considerations such as classifier inference time, memory usage, and scalability. This paper demonstrates the feasibility of live ML inference and reinforces the practicality of integrating such models into operational cybersecurity architectures.

.

4. **Kafka-Based Intrusion Detection System – GIJET, GRENZE Scientific Society (2024)**[8]

   This paper proposes a novel intrusion detection system that leverages Apache Kafka as a real-time messaging backbone to manage high-throughput network data. Kafka’s publish-subscribe model is employed to decouple the data ingestion, classification, and logging stages, allowing each to operate asynchronously. The study explores how this architecture enables real-time packet analysis while maintaining low latency and high system resilience. Machine learning models are embedded within Kafka consumer nodes to classify packets as they are streamed through Kafka topics. The system is evaluated in terms of scalability, fault tolerance, and modularity, showing that Kafka enables smooth performance under increasing network load. Additionally, external integrations like Slack notifications are incorporated to enhance usability and alert delivery. The modular structure makes it easy to add components such as retraining modules, dashboards, or log exporters without affecting the core functionality. This research demonstrates that Kafka-based systems offer a powerful foundation for building scalable, event-driven IDS capable of handling dynamic and high-volume environments.

5. **A Comprehensive Review of AI-Based Intrusion Detection Systems – Journal of Information Security and Applications (2023)**

This comprehensive survey reviews the spectrum of AI techniques used in modern IDS, including supervised, unsupervised, semi-supervised, and deep learning models. The review provides insights into the trade-offs between different methods, such as accuracy, scalability, training complexity, and interpretability. It discusses commonly used algorithms like SVM, Decision Trees, k-NN, and deep neural networks, as well as their applications in anomaly detection, signature matching, and hybrid models. One of the key challenges addressed in the review is **concept drift**, which occurs when network behavior changes over time, leading to a decline in model performance. The authors propose adaptive learning and ensemble approaches as effective solutions to maintain long-term accuracy. Additionally, the paper explores the difficulties in deploying AI models in real-time systems, where data volume, processing speed, and system responsiveness are critical. This review serves as a theoretical backbone for IDS research and provides essential guidance on model selection, deployment strategies, and future trends in AI-based cybersecurity.[9]

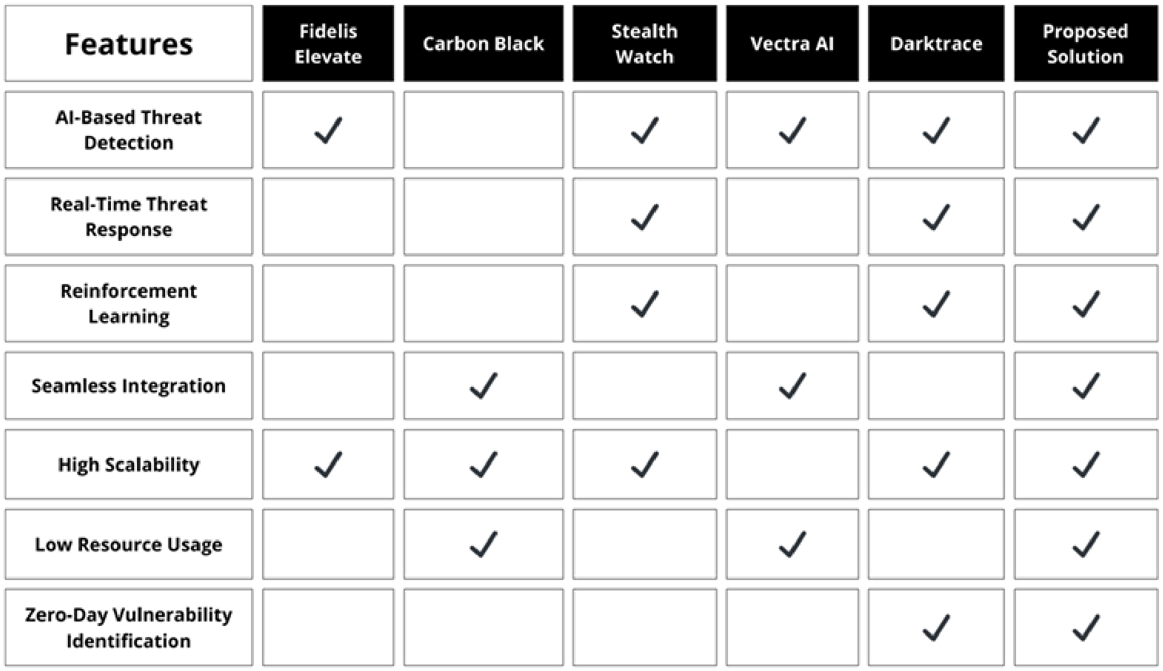
6. **Towards a Scalable and Adaptive Learning Approach for Network Intrusion Detection – Alebachew Chiche & Million Meshesha (2021)**

   This paper [10] tackles two critical problems in IDS design: scalability and adaptability. The authors present a system that uses **incremental learning techniques** to update the model continuously without requiring full retraining. This is especially important in dynamic network environments where threat patterns and data distributions can change frequently. The paper outlines a model that adjusts in real time to evolving traffic, improving its detection accuracy and reducing performance decay due to concept drift. The framework is built with scalability in mind, capable of handling large volumes of data across distributed systems. Through simulations, the researchers demonstrate the system's effectiveness in detecting unknown attacks and adapting to new behaviors without sacrificing performance or requiring manual intervention. The work makes a strong case for the use of **adaptive learning** in IDS, emphasizing how self-updating models can sustain long-term system reliability in fluctuating environments like enterprise or cloud networks.

## 1.5. Gap Analysis

The gap analysis reveals that while existing intrusion detection systems like Fidelis Elevate, Carbon Black, Stealth Watch, Vectra AI, and Darktrace offer several strong features, none provide a complete, balanced solution that addresses the evolving needs of modern cybersecurity environments. Most tools focus on either high scalability, AI-based detection, or seamless integration, but few combine these capabilities with low resource usage, real-time response, and adaptive learning. For instance, reinforcement learning—a key feature for adapting to new and unseen attack patterns—is missing in most commercial tools. Additionally, only a limited number support real-time threat handling while maintaining lightweight performance suitable for smaller or distributed systems.

The proposed solution aims to close these gaps by offering a more unified and efficient approach. It integrates real-time packet classification with the ability to retrain machine learning models automatically based on collected data. Unlike many existing tools, it is designed to operate with minimal overhead, making it suitable even for environments with limited computational resources. Furthermore, features like adaptive learning, manual retraining control, and integration with Prometheus for metrics help improve both visibility and long-term accuracy. This design not only addresses the limitations found in current solutions but also aligns with the growing need for flexible, intelligent, and self-updating intrusion detection systems.



| Features | Fidelis Elevate | Carbon Black | Stealth Watch | Vectra AI | Darktrace | Proposed Solution |
|---|---|---|---|---|---|---|
| AI-Based Threat Detection | ✓ |  | ✓ | ✓ | ✓ | ✓ |
| Real-Time Threat Response |  |  | ✓ |  | ✓ | ✓ |
| Reinforcement Learning |  |  | ✓ |  | ✓ | ✓ |
| Seamless Integration |  | ✓ |  | ✓ |  | ✓ |
| High Scalability | ✓ | ✓ | ✓ |  | ✓ | ✓ |
| Low Resource Usage |  | ✓ |  | ✓ |  | ✓ |
| Zero-Day Vulnerability Identification |  |  |  |  | ✓ | ✓ |

,

*Figure 1: Gap Analysis*

## 1.6. Proposed Solution

Heimdall is a real-time, intelligent intrusion detection system developed to address the shortcomings of traditional signature-based methods and static machine learning approaches. Unlike conventional IDS solutions that rely on fixed rules or models, Heimdall introduces adaptability and modularity at its core, allowing it to remain effective in dynamic and evolving network environments. Its architecture is designed to ensure scalability, flexibility, and ease of integration, making it suitable for both enterprise and academic use cases.

The system begins by capturing live network traffic through **Scapy**, a Python-based packet sniffer, which operates on the host's network interface. From each captured packet, a carefully selected set of features—based on widely studied intrusion detection datasets—is extracted. These features represent key attributes of the traffic such as source and destination IPs, packet size, protocol types, and flow-based behavior. Once extracted, the features are serialized into JSON format and published to a **Kafka topic**, creating a decoupled and scalable streaming pipeline that allows the system to ingest and process large volumes of data asynchronously.

On the processing side, a **Kafka consumer** reads incoming feature data and applies **pre-trained machine learning models** to classify each packet as either benign or malicious. These models are trained on benchmark datasets like **CICIDS2017** and **NSL-KDD**, using algorithms such as **Random Forest** and **Decision Tree**, selected for their high interpretability and performance on structured network data. Each prediction is timestamped and logged, and results are simultaneously streamed to a **Flask-based web dashboard**, where they are displayed in visual formats such as bar charts, logs, and performance summaries.

A defining feature of Heimdall is its **adaptive learning capability**. The system is built to monitor its own predictive performance over time. When a drop in accuracy is detected or when a configurable threshold of newly labeled samples is reached, Heimdall triggers its **retraining module**. This process involves reloading collected labeled data, updating the model, and replacing the current version without interrupting ongoing classification. This retraining mechanism ensures the system remains effective in the face of **concept drift**—the gradual change in network behavior over time.

Additionally, Heimdall integrates with **Prometheus** for system observability. Key metrics such as total packets processed, threats detected, retraining events, and system uptime are exposed via a /metrics endpoint and can be visualized using **Grafana**[11] for extended monitoring. This integration provides administrators with real-time insights into system health and threat trends, allowing for proactive response and resource optimization.

## 1.7. Project Plan

This section outlines the detailed planning and execution strategy followed throughout the development lifecycle of the Heimdall project. The project was managed in phases across the FYP-I, FYP-II, and FYP-III stages, ensuring a structured and goal-oriented approach. It includes a comprehensive Work Breakdown Structure (WBS) that divides the entire scope into manageable components, and a Roles & Responsibility Matrix that defines each team member's contributions. This plan was instrumental in tracking progress, managing dependencies, and achieving the milestones required for a functional, scalable, and adaptive real-time intrusion detection system.

### 1.7.1 Work Breakdown Structure

The Work Breakdown Structure (WBS) divides the project into hierarchical tasks, which were tackled across phases aligned with the academic calendar. Each major deliverable was broken down into actionable subtasks to ensure clarity and efficient progress tracking.

**Phase I: Research and Requirement Gathering (FYP-I)**

1. **Topic Finalization**
   Selection of project scope, domain, and problem statement.

2. **Literature Review**
   Review of AI-based intrusion detection systems, machine learning for cybersecurity, adaptive learning, and real-time data pipelines.

3. **Dataset Identification**
   Study and selection of publicly available datasets (e.g., NSL-KDD, CICIDS2017).

4. **Initial Tool Research**
   Evaluation of Kafka, Scapy, Flask, Prometheus, and Python ML libraries.

**Phase II: System Design and Dataset Preparation (FYP-II)**

1. **Architecture Design**
   Development of system architecture including Kafka-based pipeline and ML integration.

2. **UML Diagrams**
   Creation of use case, sequence, activity, component, and deployment diagrams.

3. **Feature Engineering**
   Extracting and processing relevant features from raw packet data.

4. **Dataset Preprocessing**
   Data cleaning, normalization, splitting, and labeling for training/testing.

**Phase III: Development and Integration (FYP-II to FYP-III)**

1. **Backend Development**
   Kafka Producer for feature streaming. Kafka Consumer for real-time classification using ML models.

2. **ML Model Development**
   Training and testing of models (Random Forest, Decision Tree).

### 1.7.2 Roles & Responsibility Matrix

The project team was composed of three members, with clearly assigned roles and shared collaborative duties. Each member contributed according to their strengths while maintaining active participation in group decisions and integration.

| Team Member | Role | Responsibilities |
|---|---|---|
| **Ahsan Ahmed** | Machine Learning Engineer | Selected and evaluated ML models (RandomForest, Decision Tree). Implemented training and testing pipelines- Developed adaptive retraining logic |
| **Mubahil Ahmad** | Backend Developer and Integration Lead | Developed Kafka Producer and Consumer modules. Integrated ML with live packet data stream. Handled Prometheus-based monitoring and logging |
| **Syed Ali Zain Ul Abedin Naqvi** | Frontend Developer &Test Coordinator | Designed and built Flask dashboard for live predictions, logs, and retraining. Managed system-wide testing and validation. Prepared UI for metrics |

All three members were equally involved in:

- Conducting the literature review
- Preparing diagrams and technical documentation
- Report writing and formatting
- Presentation and defense preparation

This division ensured smooth progress, accountability, and efficient handling of all system modules.

### 1.7.3 Gantt Chart

The Gantt chart outlines the project timeline for the development and implementation of the Heimdall threat detection system. It is divided into distinct phases, each mapped to specific weeks, ensuring structured progress and effective milestone tracking. The project spans over several months and includes the following key phases:

1. **Requirement Gathering and Analysis (Week 1–2):**
   The team begins by identifying system requirements, including functional and non-functional needs. Key components such as Kafka integration, Scapy for packet capture, and machine learning model needs are defined here.
2. **Design and Architecture Planning (Week 3–4):**
   Focuses on outlining the architecture of Heimdall. This includes defining the Kafka-based streaming pipeline, Prometheus integration, and dashboard architecture.
3. **Dataset Collection and Preprocessing (Week 5–6):**
   CICIDS and NSL-KDD datasets are obtained and preprocessed. Feature selection and scaling techniques (e.g., MinMaxScaler, StandardScaler) are applied to prepare for model training.
4. **Model Training and Evaluation (Week 7–8):**
   Initial models such as Decision Tree and Random Forest are trained and evaluated for threat classification performance using metrics like accuracy and F1-score.
5. **Kafka Producer & Consumer Development (Week 9–10):**
   Development of producer scripts to capture live network traffic and send it to Kafka, along with consumer scripts that process and classify this data.
6. **Dashboard Development (Week 11–12):**
   Creation of a Flask-based dashboard for threat visualization, retraining controls, and log viewing. Metrics endpoints are added for Prometheus scraping.
7. **Prometheus and Grafana Integration (Week 13):**
   Prometheus metrics (CPU usage, threat counts, retraining stats) are integrated and visualized via Grafana dashboards.
8. **Testing and Optimization (Week 14):**
   System-wide testing is conducted including load tests, edge-case validation, and performance tuning.
9. **Final Deployment and Documentation (Week 15):**
   Heimdall is deployed on an Ubuntu-based VM. Documentation is finalized, covering architecture, usage instructions, and system behavior.

10. **Presentation and Final Report Submission (Week 16):**

    Project is presented to stakeholders. Final report, including evaluation results and implementation details, is submitted.

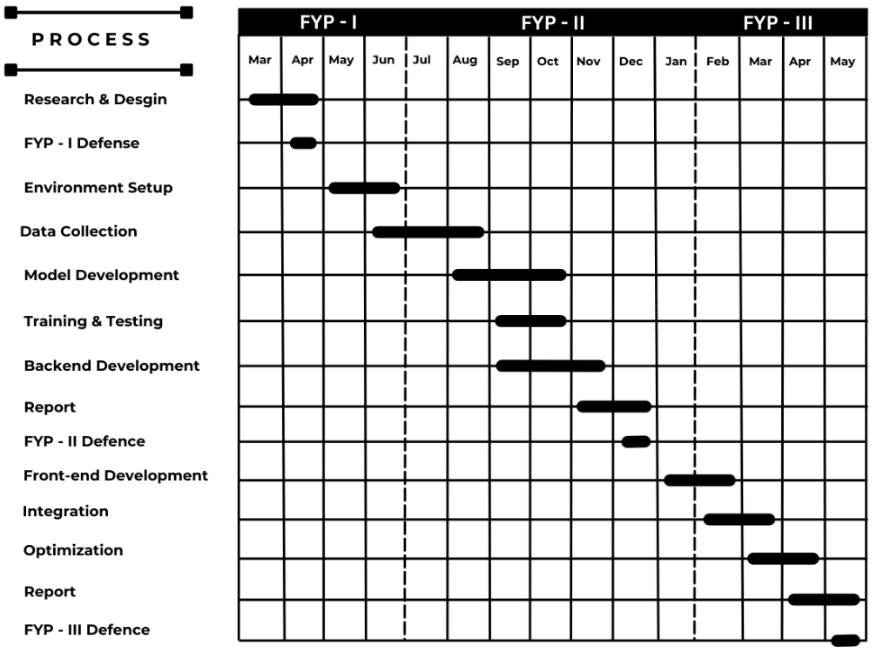


*Figure 2: Gantt Chart*

## 1.8. Report Outline

This report is organized into eight chapters, each covering a critical aspect of the Heimdall project, from inception to deployment. The structure provides a logical progression of ideas, technical insights, and strategic considerations to guide the reader through the complete development lifecycle of the system.

- **Chapter 1** introduces the background of the project, highlighting the motivations, challenges, and the need for an adaptive intrusion detection system. It also outlines the proposed solution and project scope.
- **Chapter 2** defines the functional and non-functional software requirements, along with system constraints, assumptions, and external interfaces.

- **Chapter 3** presents the use case model, identifying the core functionalities of the system from the perspective of various user roles.
- **Chapter 4** explains the system architecture and design decisions, supported by relevant architectural and UML diagrams.
- **Chapter 5** details the implementation process, including key control flows, technology stack, coding standards, and development tools used.
- **Chapter 6** outlines the business plan, target market segments, competitive analysis, product offerings, and a comprehensive SWOT analysis.
- **Chapter 7** discusses the testing strategies, evaluation metrics, and performance outcomes of the system during validation.
- **Chapter 8** provides a conclusion, summarizing achievements, lessons learned, and offering recommendations for future improvements and extensions.

# Chapter 2
# Software Requirement Specifications

# **Chapter 2:** Software Requirement Specifications

## 2.1 Introduction

### 2.1.1 Purpose

This Software Requirements Specification (SRS) document defines the complete set of functional and non-functional requirements for the Heimdall system — an AI-driven, real-time intrusion detection platform. Heimdall is designed to monitor network traffic, detect threats using machine learning classifiers, and adapt over time through automated retraining mechanisms. This document outlines the scope of version 1.0 of the software and includes specifications for all core components: live data capture, feature extraction, Kafka-based message streaming, real-time classification, dynamic model retraining, system observability, and the web-based user interface. The SRS serves as a foundational reference for the development, testing, deployment, and future maintenance of the system, ensuring all stakeholders have a shared understanding of the project's technical and operational expectations.

### 2.1.1 Document Conventions

This document uses 12 pt Times New Roman font with 1.5 line spacing. Section headings follow hierarchical numbering. Requirements are identified with tags such as REQ-SF#-# and are categorized under functional and non-functional types. Important system features are described using standard IEEE SRS formatting.

### 2.1.2 Intended Audience and Reading Suggestions

This document is intended for individuals involved in the development, testing, and evaluation of the Heimdall intrusion detection system. The primary audience includes software developers, security engineers, quality assurance (QA) testers, project supervisors, and anyone responsible for reviewing or maintaining the system in the future.

Developers are encouraged to focus on the system features, functional requirements, and interface specifications, especially in Sections 2.3 and 2.4. These parts explain how different modules interact and what is expected during implementation. QA testers should refer to the performance, security, and reliability aspects described in Section 2.5 to better understand what needs to be validated during system testing. For readers unfamiliar with the project or those reviewing it for the first time, it is recommended to start with Sections 2.1 and 2.2. These provide a general overview of the system's purpose, scope, and design context before moving on to the more technical sections that follow.

### 2.1.3 Product Scope

Heimdall is a lightweight, Python-based intrusion detection system designed to detect cyber threats in real time using machine learning. It is built to handle dynamic network environments where traffic patterns change frequently, and traditional rule-based systems fall short. One of the key features of Heimdall is its ability to adapt over time through both automated and manual model retraining. This reduces the need for constant manual updates and ensures the system stays effective as new types of network behavior or attacks emerge.

The system offers a complete end-to-end pipeline—from packet capture and feature extraction to classification, retraining, and monitoring. A web-based dashboard allows users to view logs, monitor performance, and manually control retraining when needed. Heimdall is aimed at improving threat detection accuracy while remaining easy to deploy and maintain, making it suitable for educational institutions, small enterprises, and research environments where flexibility and adaptability are important.

## 2.2 Overall Description

### 2.2.1 Product Perspective

Heimdall is a newly developed, standalone software system aimed at overcoming the limitations of traditional intrusion detection tools that rely on static signature-based methods. Unlike conventional systems that need manual updates to detect known threats, Heimdall uses machine learning to identify unusual or malicious network behavior in real time. It is built using a modular architecture where each component handles a specific task—from packet capturing and feature extraction to data streaming, classification, and visualization.

The system captures live network packets using **Scapy**[12], processes the extracted features through **Apache Kafka**, [11] and classifies the traffic using trained machine learning models such as Decision Tree and Random Forest. All results are logged and displayed through a web-based **Flask dashboard**, which provides visibility into current network activity and system performance. The design also includes a retraining module that updates the ML model based on new labeled data, helping the system adapt over time.

Thanks to its modular structure and lightweight design, Heimdall can be deployed in various environments such as university labs, research settings, or small-to-medium organizations that need an adaptable and cost-effective security tool.

### 2.2.2 User Classes and Characteristics

The Heimdall system is designed to support several types of users, each playing a distinct role in its deployment, operation, and maintenance:

- **Admin / Operator**: Responsible for installing and deploying the system, monitoring service uptime, and ensuring all components (Kafka, Flask, Prometheus) are running properly. This role requires moderate technical experience, particularly with Linux systems and networking.
- **Security Analyst**: Focuses on reviewing detected threats, monitoring logs, and initiating retraining when necessary. A background in cybersecurity or experience with network traffic analysis would help this user make effective use of the dashboard and prediction insights.
- **Developer**: In charge of implementing new features, updating the ML models, and maintaining the backend logic of the system. This role involves experience in Python, Kafka, and machine learning tools like scikit-learn.
- **Tester**: Evaluates the functionality and performance of the system under different network scenarios. Familiarity with traffic simulation tools, packet generators, or network testing utilities is beneficial for this role.

### 2.2.3 Operating Environment

Heimdall runs in Linux environments, preferably on Ubuntu 20.04 or later, with Python 3.8 or above. The system depends on several open-source components such as Apache Kafka for data streaming and Prometheus for metrics collection. A modern desktop browser (e.g., Chrome or Firefox) is required to access the Flask-based dashboard. The setup is designed to be container-friendly and can be deployed using Docker or Docker Compose for convenience and portability. To enable live packet capture, the host machine must have root privileges and meet the following minimum hardware requirements: at least 8 GB of RAM and a quad-core processor

### 2.2.4 Design and Implementation Constraints

Heimdall operates under a few technical and design-related constraints that must be considered during development and deployment:

- Packet sniffing operations require administrative (root) privileges.
- Kafka and Prometheus must be properly installed and running before Heimdall starts.
- Only Python-based machine learning libraries (such as scikit-learn and joblib) are permitted for model training and inference.
- Retraining operations should not block or interfere with ongoing real-time classification.
- The Flask dashboard must remain responsive and stable under concurrent access from multiple users.

### 2.2.5 Assumptions and Dependencies

The following assumptions and dependencies are acknowledged for the current version of Heimdall:

- Kafka and Prometheus services are expected to be running and accessible when the system starts.
- Users are responsible for correctly labeling data that will be used for retraining purposes.
- The network traffic captured will include a representative mix of both benign and malicious activity.
- Docker is used as the preferred method for deployment in production environments.

No external or third-party machine learning APIs will be used; all model training and inference take place locally.

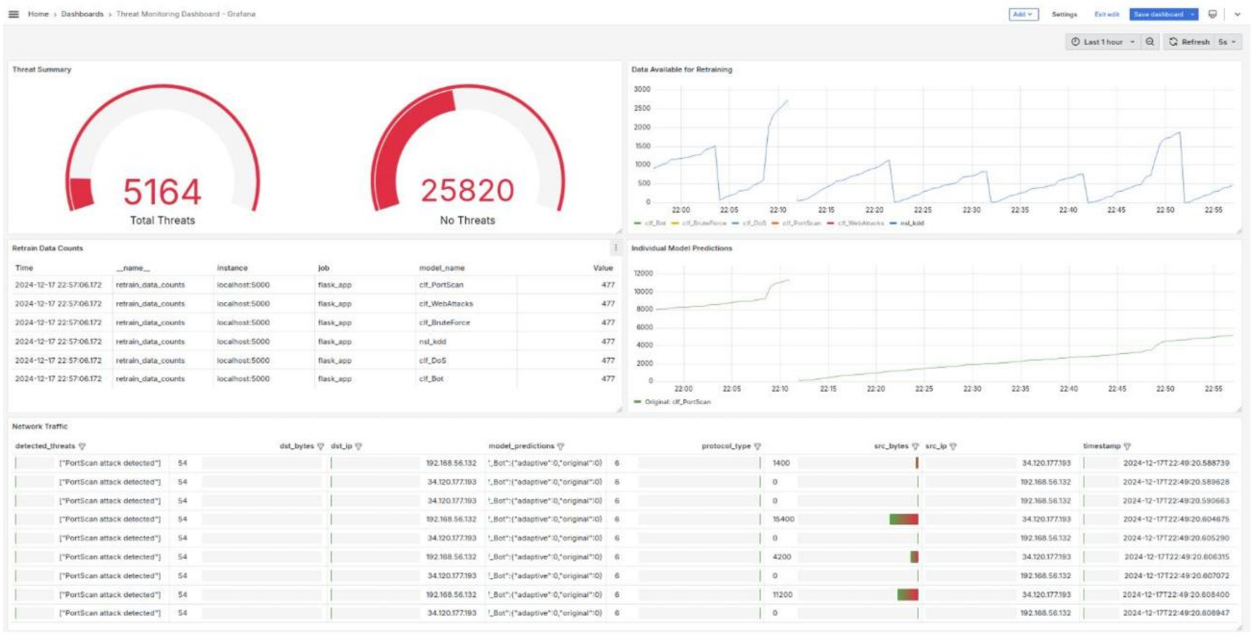
## 2.3 External Interface Requirements

### 2.3.1 User Interfaces

Users interact with Heimdall through a lightweight web dashboard built with Flask and standard front-end technologies. The dashboard includes:

- A visual threat summary showing detected attack types and their frequency.
- A live log table displaying real-time classification results with timestamps and IP addresses.
- A retraining status section showing the number of labeled samples and retrain thresholds.
- A manual retraining button that allows users to trigger model updates.
- Simple navigation between core sections such as "Home" and "Logs."

The dashboard is fully responsive and intended for desktop browsers.



Retrain Data Counts

| Time | __name__ | instance | job | model_name | Value |
|---|---|---|---|---|---|
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | clf_PortScan | 477 |
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | clf_WebAttacks | 477 |
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | clf_BruteForce | 477 |
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | nsl_kdd | 477 |
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | clf_DoS | 477 |
| 2024-12-17 22:57:06.172 | retrain_data_counts | localhost:5000 | flask_app | clf_Bot | 477 |

Network Traffic

| detected_threats | dst_bytes | dst_ip | model_predictions | protocol_type | src_bytes | src_ip | timestamp |
|---|---|---|---|---|---|---|---|
| ["PortScan attack detected"] | 54 | 192.168.56.132 | '_Bot":{"adaptive":0,"original":0} | 6 | 1400 | 34.120.177.193 | 2024-12-17T22:49:20.588739 |
| ["PortScan attack detected"] | 54 | 34.120.177.193 | '_Bot":{"adaptive":0,"original":0} | 6 | 0 | 192.168.56.132 | 2024-12-17T22:49:20.589628 |
| ["PortScan attack detected"] | 54 | 34.120.177.193 | '_Bot":{"adaptive":0,"original":0} | 6 | 0 | 192.168.56.132 | 2024-12-17T22:49:20.590663 |
| ["PortScan attack detected"] | 54 | 192.168.56.132 | '_Bot":{"adaptive":0,"original":0} | 6 | 15400 | 34.120.177.193 | 2024-12-17T22:49:20.604675 |
| ["PortScan attack detected"] | 54 | 34.120.177.193 | '_Bot":{"adaptive":0,"original":0} | 6 | 0 | 192.168.56.132 | 2024-12-17T22:49:20.605290 |
| ["PortScan attack detected"] | 54 | 192.168.56.132 | '_Bot":{"adaptive":0,"original":0} | 6 | 4200 | 34.120.177.193 | 2024-12-17T22:49:20.606315 |
| ["PortScan attack detected"] | 54 | 34.120.177.193 | '_Bot":{"adaptive":0,"original":0} | 6 | 0 | 192.168.56.132 | 2024-12-17T22:49:20.607072 |
| ["PortScan attack detected"] | 54 | 192.168.56.132 | '_Bot":{"adaptive":0,"original":0} | 6 | 11200 | 34.120.177.193 | 2024-12-17T22:49:20.608400 |
| ["PortScan attack detected"] | 54 | 34.120.177.193 | '_Bot":{"adaptive":0,"original":0} | 6 | 0 | 192.168.56.132 | 2024-12-17T22:49:20.608947 |

*Figure 3: Frontend*

### 2.3.2 Hardware Interfaces

Heimdall interfaces directly with the host machine's network interface card (NIC), using Scapy to capture packets. Promiscuous mode and raw socket access are required for capturing all incoming and outgoing packets. The system supports standard Ethernet-based interfaces.

### 2.3.3 Software Interfaces

Heimdall communicates with and depends on the following software components:

- **Kafka**: Used for publishing and consuming serialized packet data.
- **Prometheus**: Collects custom metrics from Heimdall via HTTP.[13]
- **Flask**: Provides the backend API and web interface for the dashboard.
- **joblib**: Saves and loads trained ML models.
- **scikit-learn**: Handles model training, prediction, and preprocessing.

All inter-component communication occurs locally, and no data is transmitted to external servers.

### 2.3.4 Communications Interfaces

Heimdall supports the following communication methods:

- HTTP/HTTPS for dashboard access and Prometheus metric scraping.
- Kafka's native protocol for communication between producers and consumers.
- RESTful API endpoints served via Flask, typically available on port 5000.

Optional communication with **Grafana** is enabled via Prometheus' data exporter for advanced dashboard visualization.

## 2.4 System Features

Heimdall offers several key system features. It performs real-time classification of network packets using machine learning models, enabling immediate threat detection and response. Adaptive retraining ensures the models remain accurate over time by learning from newly labeled data, either automatically or manually. A Flask-based web dashboard provides a user-friendly interface for monitoring threats, reviewing logs, and managing retraining actions. Prometheus integration exposes system metrics like detection counts, resource usage, and retraining events for monitoring and visualization via Grafana. The system captures live network traffic using Scapy and extracts relevant features for analysis. Kafka enables scalable and decoupled data streaming between the packet capture module and the classification engine, supporting high-throughput and fault-tolerant processing.

## 2.4.1 Real-Time Classification

### 2.4.1.1 Description and Priority

This is one of the most critical features of Heimdall, allowing it to detect threats as they occur. When a network packet is captured, Heimdall immediately extracts relevant features and uses a trained machine learning model to classify the packet as either benign or malicious. Real-time classification minimizes response time and allows administrators to act quickly against emerging threats. The accuracy and speed of this feature are essential for system reliability and effectiveness. Therefore, it is marked as high priority.

### 2.4.1.2 Stimulus/Response Sequences

- Stimulus: A network packet is captured through the Scapy module.
- Response: The system extracts features, sends them to Kafka, retrieves them via a Kafka consumer, applies the ML model, logs the result, and visualizes it on the dashboard.

### 2.4.1.3 Functional Requirements

- REQ-SF1-1: Each packet must be classified within 100 milliseconds of capture to meet real-time constraints.
- REQ-SF1-2: Classification output must include metadata such as timestamp, source IP, destination IP, and classification label.
- REQ-SF1-3: Prediction results must be logged persistently and visualized on the web dashboard with minimal delay.

### 2.4.2 Adaptive Retraining

#### 2.4.2.1 Description and Priority

To maintain long-term accuracy and combat concept drift, Heimdall incorporates an adaptive retraining feature. This component enables the system to learn from new, labeled traffic data. The retraining can occur either automatically—once a defined data threshold is reached—or manually, via user interaction through the dashboard. This ensures the model evolves alongside changes in network behavior. Given its role in maintaining system performance, it is considered high priority.

#### 2.4.2.2 Stimulus/Response Sequences

- Stimulus: A threshold number of labeled data points are collected, or the user clicks the manual retrain button.
- Response: The system initiates a new training process, validates the model, and seamlessly replaces the old model without stopping real-time classification.

#### 2.4.2.3 Functional Requirements

- REQ-SF2-1: The system shall trigger retraining automatically upon reaching 100 new labeled samples.
- REQ-SF2-2: Manual retraining must be accessible via a clear control on the dashboard.
- REQ-SF2-3: Retraining must occur in the background, allowing real-time classification to continue without interruption.

### 2.4.3 System Configuration and Control

#### 2.4.3.1 Description and Priority

System administrators and developers need the ability to fine-tune operational parameters such as classification thresholds, model hyperparameters, and logging preferences. This feature allows for easy customization and tuning without code modifications. Configuration is managed through external files or environment variables. It is of medium priority due to its role in system flexibility.

#### 2.4.3.2 Stimulus/Response Sequences

- Stimulus: A user changes a value in a configuration file or sets a new environment variable.
- Response: The system reloads the configuration on the next cycle or request, applying the updated settings without a restart.

#### 2.4.3.3 Functional Requirements

- REQ-SF3-1: The system must detect and apply changes in configuration files without a full restart.
- REQ-SF3-2: System behavior must be modifiable through environment variables or config files (e.g., logging level, retraining threshold).

### 2.4.4 Metric Monitoring

#### 2.4.4.1 Description and Priority

Observability is essential for maintaining system performance and diagnosing problems. Heimdall integrates with Prometheus to expose internal metrics such as packet classification rate, system uptime, retraining frequency, and memory usage. These can be visualized using external tools like Grafana. This is a medium-priority feature that enhances system maintainability and transparency.

#### 2.4.4.2 Stimulus/Response Sequences

- Stimulus: A Prometheus server queries the Heimdall metrics endpoint.
- Response: Heimdall responds with the latest metrics in the expected format.

#### 2.4.4.3 Functional Requirements

- REQ-SF4-1: Heimdall must expose key metrics through a Prometheus-compatible endpoint.
- REQ-SF4-2: Exposed metrics must include classification rate, error counts, retrain events, and system resource usage.

## 2.5 Nonfunctional Requirements

### 2.5.1 Performance Requirements

The nonfunctional requirements define how the Heimdall system should behave under various conditions. These requirements ensure the system’s performance, reliability, maintainability, and usability in real-world environments. While functional requirements define what the system does, nonfunctional requirements describe how well it performs and how easily it can be used, maintained, or deployed.

Heimdall is a real-time system, and performance is a critical consideration. The system must be capable of classifying network packets within 100 milliseconds from the time they are captured. It should sustain a throughput of at least 1000 packets per second under standard conditions, without introducing bottlenecks. During retraining operations, the classification functionality must remain unaffected. The dashboard must refresh real-time charts and logs within 1 second of receiving new data. These performance targets ensure the system can operate in high-traffic environments such as corporate networks or academic institutions.

### 2.5.2 Safety Requirements

As Heimdall deals with live network traffic, the integrity and reliability of its logs are crucial. The system must ensure that log data is persistently stored and not lost due to system crashes or unexpected shutdowns. In the event of a failure, Heimdall must automatically resume from the last known state without data corruption. The packet sniffer must operate in passive mode to avoid interfering with live network operations. Additionally, administrators must have clear visibility into system status to prevent misconfigurations or accidental data loss. No part of the system should allow unauthorized modification of training data or prediction logs.

### 2.5.3 Security Requirements

Security is a top priority in any network monitoring system. Heimdall must enforce authentication for access to its dashboard and retraining functionality to prevent unauthorized usage. All sensitive endpoints, such as those controlling model retraining or viewing logs, must be protected using credentials. Data collected from network packets must be stored locally and should not be transmitted to third-party servers. The system must not expose raw traffic or user IPs outside of the host environment. Additionally, all communications between components (e.g., Flask server and Prometheus) should occur over secure internal channels. If deployed in a cloud environment, further encryption or access control measures must be applied.

### 2.5.4 Usability Requirements

Heimdall must be designed for ease of use by cybersecurity analysts and administrators, even those with limited technical expertise. The dashboard should present threat summaries, logs, and retraining controls in a visually intuitive format, using charts, tables, and buttons. Navigation must be minimal, with critical actions such as reviewing logs or triggering retraining accessible within two clicks. The interface should be responsive and compatible with major desktop browsers, ensuring that users can monitor the system without needing to install any additional software.

### 2.5.5 Reliability Requirements

The system must remain operational during long monitoring sessions without memory leaks, crashes, or degraded performance. If any module, such as the Kafka consumer or classifier, fails, the system should detect the failure and attempt to restart the service automatically. Heimdall must maintain consistent behavior during high traffic volumes and retrain models without affecting ongoing predictions. A log of system events and errors should be maintained for debugging and auditing purposes.

### 2.5.6 Maintainability/Supportability Requirements

The codebase must be modular and adhere to best practices for readability and reusability. Each module (e.g., capture, classify, retrain, dashboard) should be independently upgradable without requiring changes to the others. The system must be documented with clear setup instructions, configuration files, and code comments. Support tools such as Docker Compose should be used to simplify deployment and configuration. Additionally, log files should be easy to access and interpret for debugging and performance tuning.

### 2.5.7 Portability Requirements

Heimdall should run on all modern Linux distributions, including Ubuntu and Debian. It must support deployment via Docker, allowing the entire system (Kafka, Prometheus, Flask app, and ML models) to be bundled and launched in a containerized environment. This ensures easy deployment across cloud platforms, local servers, or virtual machines. All dependencies should be listed in a requirements.txt file or Docker file to ensure compatibility.

### 2.5.8 Efficiency Requirements

The system should make optimal use of CPU and memory resources. Asynchronous processing or multithreading should be implemented for components handling live data, such as the Kafka consumer and model predictor. Retraining operations must run in parallel without blocking classification tasks. Data storage must be minimal yet sufficient, avoiding unnecessary duplication of logs or models. The system should be able to operate continuously for extended periods with minimal manual intervention or resource leaks.

## 2.6 Domain Requirements

In addition to the functional and non-functional requirements outlined earlier, Heimdall must meet several domain-specific needs to ensure extensibility, compliance, and reusability. The system must be able to process standard cybersecurity datasets such as NSL-KDD and CICIDS2017 for training and testing purposes, using consistent feature extraction methods.

While Heimdall does not require a persistent database for its core operations, it must support log storage in formats like CSV or JSON for external analysis or auditing. Future versions may require database integration for long-term log retention or user role management.

Internationalization support should be considered for later releases, with a flexible front-end architecture that allows UI labels and content to be localized for different languages. The system must also follow legal and ethical guidelines for network monitoring, ensuring that captured data remains on-premises and is not shared without authorization.

Additionally, the software should be designed with reusability in mind—individual components such as the classifier module, retraining logic, or dashboard interface should be easily replaceable or portable to other projects in the same domain.

# Chapter 3
# Use Case Analysis

# **Chapter 3:** Use Case Analysis

This chapter presents the functional behavior of the Heimdall threat detection system through detailed use case analysis. The system is designed to detect, classify, and log real-time network threats using machine learning, Kafka-based data ingestion, and adaptive retraining mechanisms. This chapter includes a comprehensive use case model that illustrates how users and system components interact. Each individual use case is described technically, reflecting how the system responds to specific threat scenarios such as DoS, Brute Force, Malware, and Zero-Day attacks. The structured approach ensures that every actor's role, system functionality, exception handling, and technical dependencies are clearly outlined to validate the system design and implementation.

## 3.1. Use Case Model

The Heimdall system consists of three main actor roles: the **Network Administrator**, the **Security Analyst**, and the **System (Heimdall)** itself. The use case model reflects interactions where the system captures real-time network traffic, analyzes it using ML classifiers, generates alerts, retrains models if necessary, and visualizes threat metrics through Prometheus and Grafana.

## 3.2. Use Cases Description

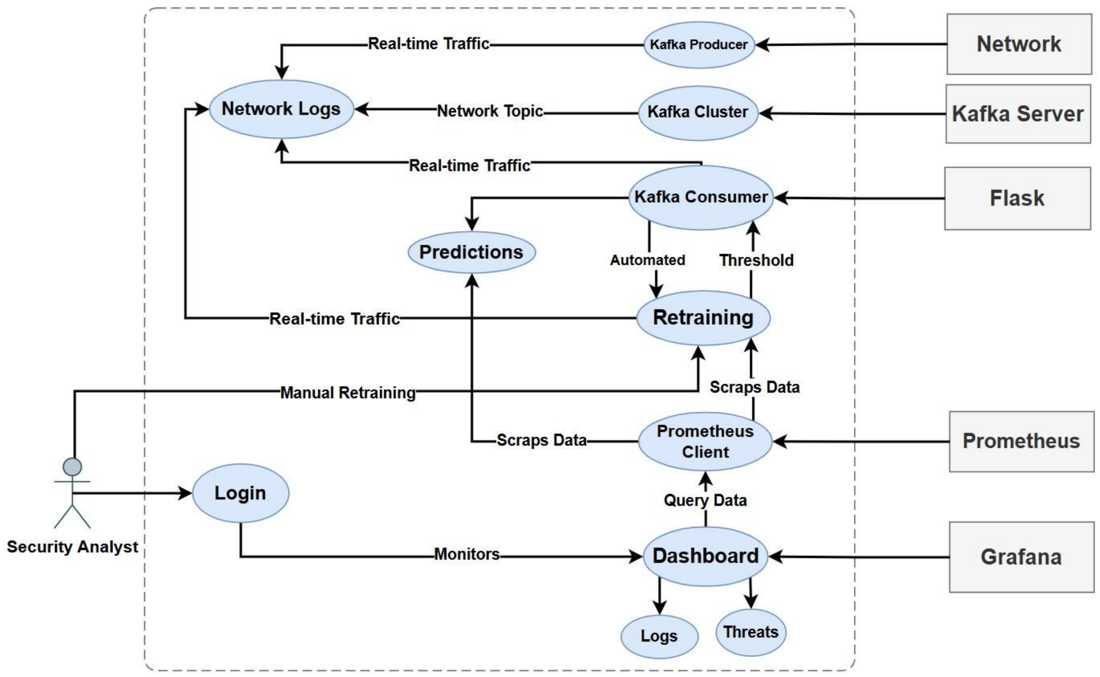


*Figure 4: Use Case 1: Network Traffic Logging and Real-Time Predictions*

### 3.1.1. User Login Function

| | |
|---|---|
| **Title** | Network Traffic Logging and Real-Time Predictions |
| **Requirement** | Heimdall must log network traffic and generate threat predictions in real time |
| **Rational** | Ensure visibility, threat identification, and adaptive learning from live traffic |
| **Restriction or Risk** | Delays during Kafka failures or resource shortages |
| **Dependency** | Kafka setup, Prometheus monitoring, and Flask dashboard |
| **Priority** | High |

*Table 1: Use Case 1*

## Use case 1

| UC_01 |
|---|
| **Actor** |
| • System<br>• Security Analyst<br>• Network Admin |
| **Preconditions** |
| • Kafka infrastructure<br>• Prometheus<br>• Flask must be active |
| **Basic flow** |
| • System captures real-time traffic<br>• Kafka Producer sends features to Kafka Cluster<br>• Kafka Consumer processes and classifies data<br>• Predictions are logged<br>• Retraining is triggered if threshold is met<br>• Metrics displayed via Prometheus and Dashboard |
| **Alternate flows** |
| • Kafka fails — retry logic is applied after delay |
| **Post Condition** |
| • Logs<br>• Predictions<br>• Metrics are generated; retraining is triggered if required |

*Table 2: Use Case 1*

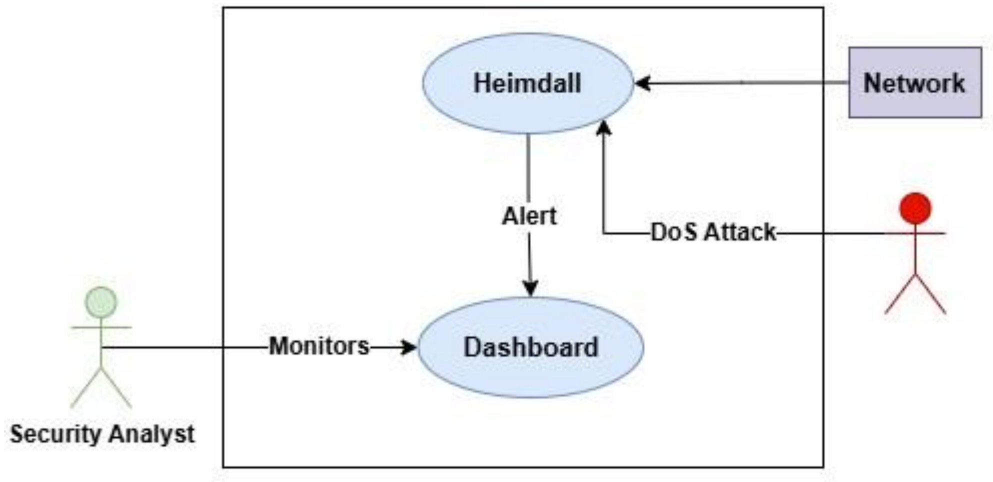


*Figure 5: Use Case 2: DoS Attack Detection*

| | |
|---|---|
| **Title** | Detect Denial of Service (DoS) Attack |
| **Requirement** | Heimdall must detect high-frequency traffic spikes |
| **Rational** | Avoid server overload or network downtime |
| **Restriction or Risk** | Legitimate spikes may resemble attack behavior |
| **Dependency** | Accurate timing and packet rate thresholds |
| **Priority** | High |

*Table 3: Use Case 2*

## Use case 2

| |
|---|
| UC_02 |
| **Actor** |
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • Monitoring running with traffic limits defined |
| **Basic flow** |
| • DoS patterns matched<br>• Attack logged |

| **Alternate flows** |
|---|
| ● Legitimate traffic spike passed |
| **Post Condition** |
| ● Alert generated and visualized |

*Table 4: Use Case 2*

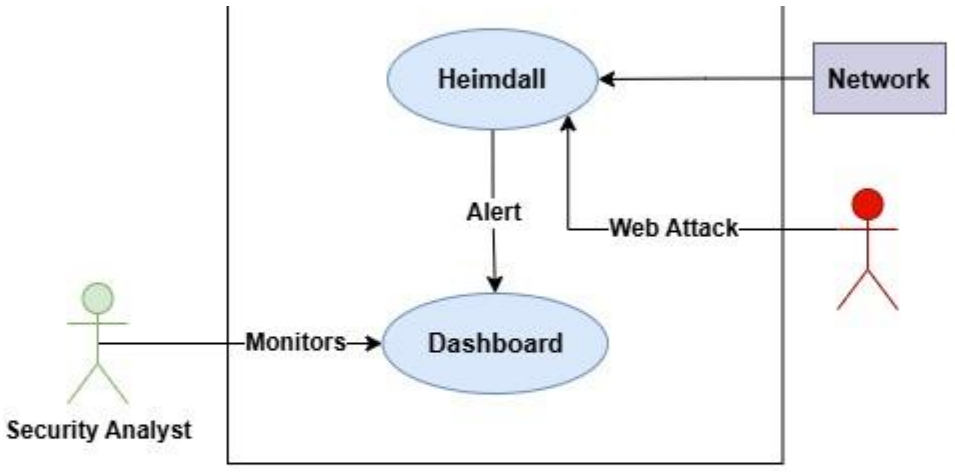


*Figure 6: Use Case 3: Web Attack Detection*

| | |
|---|---|
| **Title** | Detect Web-based Attacks (e.g., XSS, SQLi) |
| **Requirement** | Heimdall must identify suspicious web requests |
| **Rational** | Prevent data theft or service disruption |
| **Restriction or Risk** | Payload inspection not yet implemented |
| **Dependency** | Feature extraction quality |
| **Priority** | Medium |

*Table 5: Use Case 3*

## Use case 3

| UC_03 |
|---|
| **Actor** |

| |
|---|
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • System is monitoring HTTP traffic |
| **Basic flow** |
| • Anomaly detected in request pattern |
| **Alternate flows** |
| • Normal traffic passed through |
| **Post Condition** |
| • Dashboard reflects threat status |

*Table 6: Use Case 3*

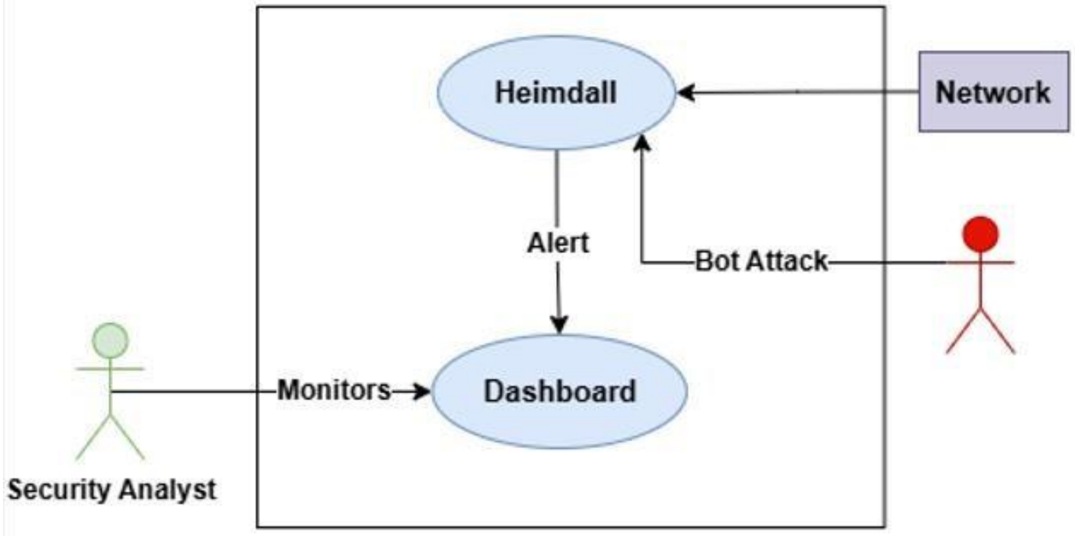


*Figure 7: Use Case 4: Bot Attack Detection*

| | |
|---|---|
| **Title** | Detect Botnet-based Attacks |
| **Requirement** | Detect repetitive or scripted attack behaviors |
| **Rational** | Prevent unauthorized automated access |
| **Restriction or Risk** | False positives with API or system monitoring agents |

| | |
|---|---|
| **Dependency** | Feature vectors reflecting automation patterns |
| **Priority** | Medium |

*Table 7: Use Case 4*

**Use case 4**

| |
|---|
| UC_04 |
| **Actor** |
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • Model trained on botnet datasets |
| **Basic flow** |
| • Pattern detected<br>• Logged in real time |
| **Alternate flows** |
| • Low-activity bots remain unnoticed |
| **Post Condition** |
| • Threat dashboard updated |

*Table 8: Use Case 4*

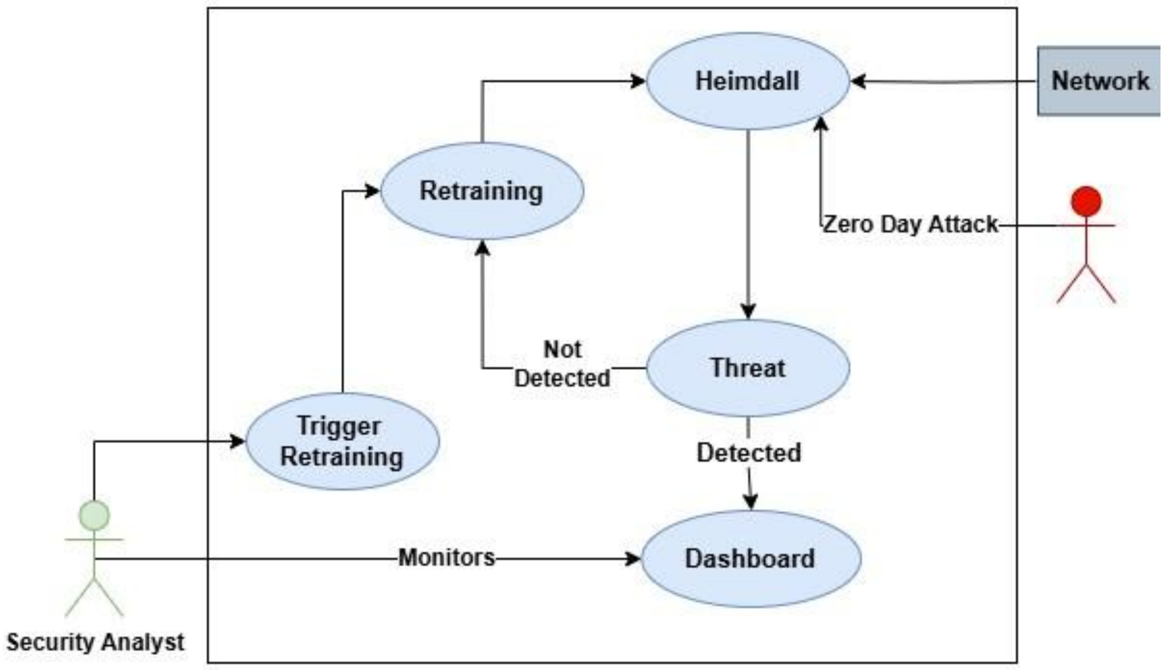


*Figure 8: Use Case 5: Zero-Day Attack Detection*

| | |
|---|---|
| **Title** | Detect Zero-Day Attacks and Trigger Adaptive Retraining |
| **Requirement** | Identify novel traffic patterns not matching known threats |
| **Rational** | Maintain high detection accuracy against emerging threats |
| **Restriction or Risk** | Anomaly-based models may raise false positives |
| **Dependency** | Anomaly detection models, retraining threshold logic |
| **Priority** | High |

*Table 9: Use Case 5*

**Use case 5**

| |
|---|
| UC_05 |
| **Actor** |

| |
|---|
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • Anomaly detection and retraining modules are active |
| **Basic flow** |
| • Anomalous pattern detected<br>• Triggered retraining |
| **Alternate flows** |
| • False positives flagged as non-threats |
| **Post Condition** |
| • Models retrained<br>• Detection accuracy improved |

*Table 10: Use Case 5*

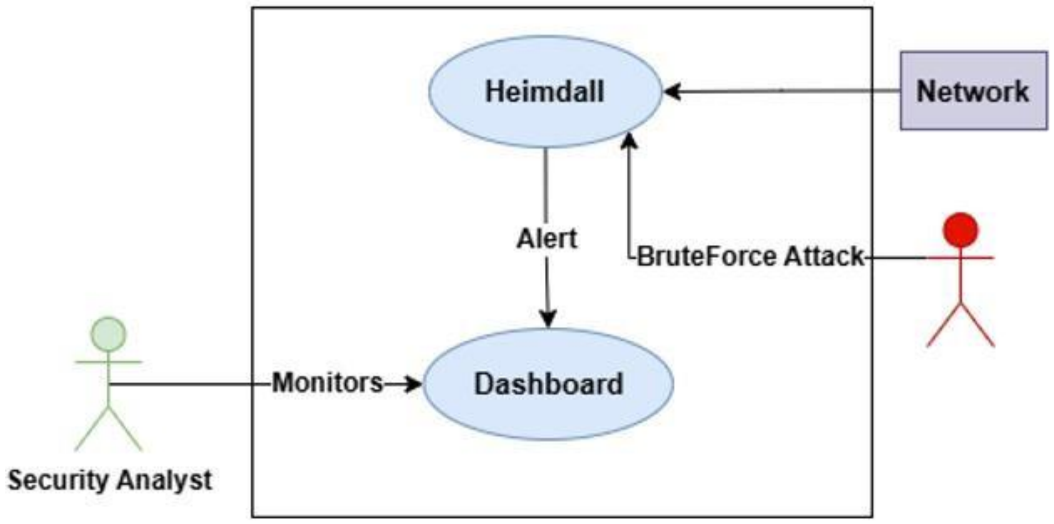


*Figure 9: Use Case 6: Brute Force Attack*

| | |
|---|---|
| **Title** | Detect Brute Force Login Attempts |
| **Requirement** | Count failed attempts over time window |
| **Rational** | Prevent account compromise |

| | |
|---|---|
| **Restriction or Risk** | Legitimate login failures may trigger false alerts |
| **Dependency** | Stateful packet tracking or login sequence logs |
| **Priority** | High |

*Table 11: Use Case 6*

### Use case 6

| |
|---|
| UC_06 |
| **Actor** |
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • Repeated login attempts in captured data |
| **Basic flow** |
| • Repeated failures flagged<br>• Alert generated |
| **Alternate flows** |
| • Legitimate retries ignored |
| **Post Condition** |
| • Security event logged for review |

*Table 12: Use Case 6*

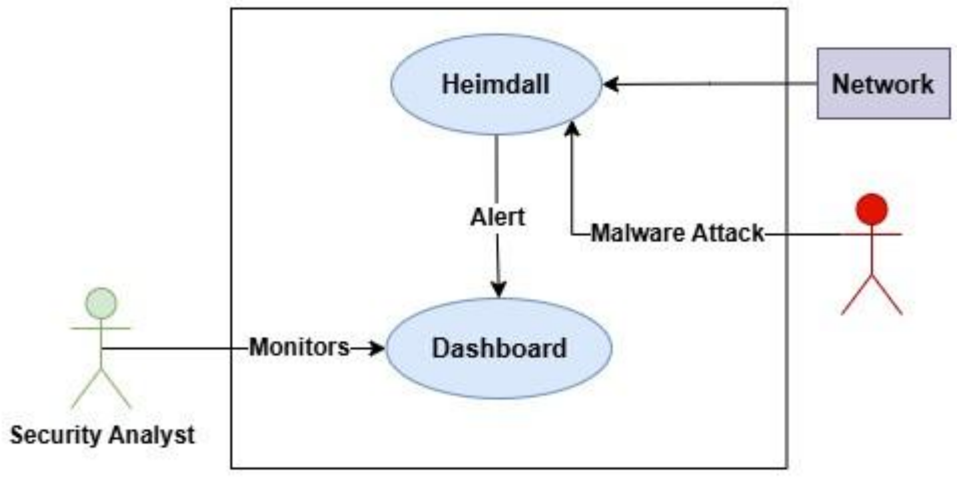


*Figure 10: Use Case 07: Malware Attack Detection*

| | |
|---|---|
| **Title** | Detect Malware Activity |
| **Requirement** | Heimdall must match abnormal traffic patterns |
| **Rational** | Prevent system compromise via malware |
| **Restriction or Risk** | Encrypted traffic may reduce visibility |
| **Dependency** | Labeled dataset and feature accuracy |
| **Priority** | High (Security) |

*Table 13: Use Case 7*

## Use case 7

| |
|---|
| UC_07 |
| **Actor** |
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • ML model trained on malware signatures |
| **Basic flow** |
| • Traffic analyzed<br>• Malware classified |

| |
|---|
| • Logged |
| **Alternate flows** |
| • Misclassification due to unseen patterns |
| **Post Condition** |
| • Analyst alerted<br>• Threat logged |

*Table 14: Use Case 7*

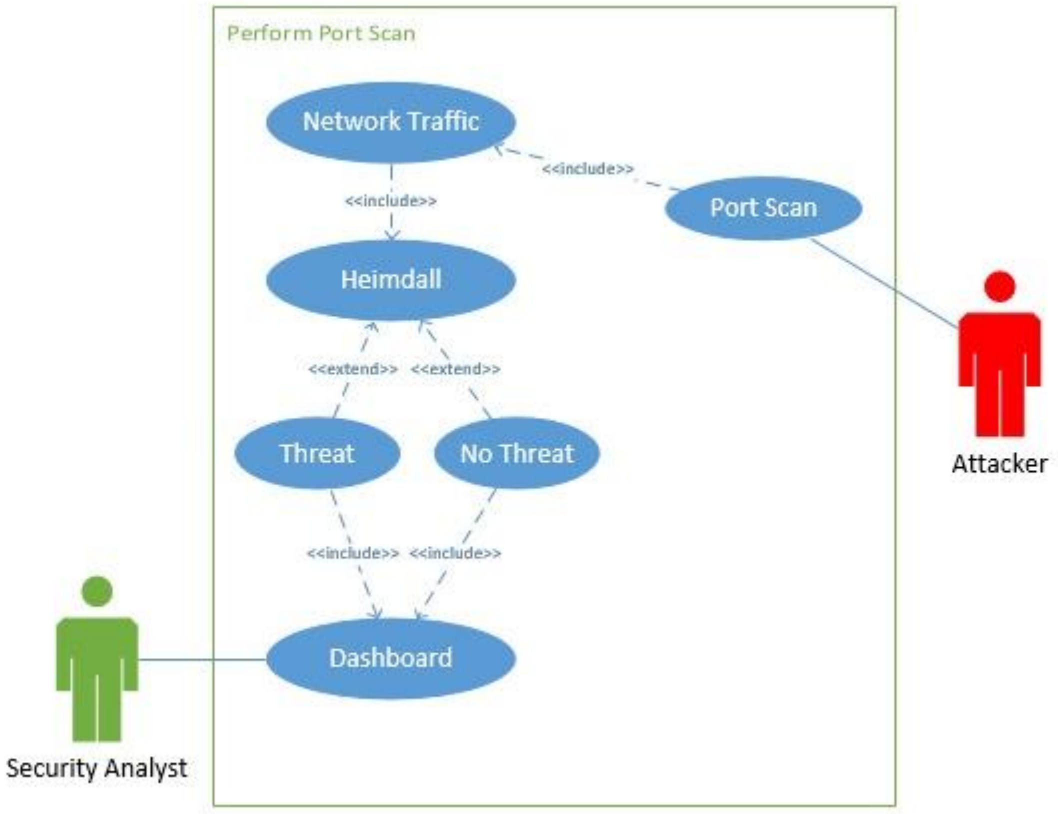


*Figure 11: Use Case 08: Port Scan Detection*

| | |
|---|---|
| **Title** | Detect Port Scan Attacks[14] |
| **Requirement** | Identify sequential or parallel probing of ports on target hosts |
| **Rational** | Prevent reconnaissance and pre-attack enumeration |

| | |
|---|---|
| **Restriction or Risk** | Legitimate port checks (e.g., vulnerability scanners) may be flagged |
| **Dependency** | Accurate flow-level traffic features and timing-based patterns |
| **Priority** | High |

*Table 15: Use Case 8*

## Use case 8

| |
|---|
| UC_08 |
| **Actor** |
| • Attacker<br>• Security Analyst |
| **Preconditions** |
| • Heimdall system and port scan detection model is active |
| **Basic flow** |
| • Port scan behavior is detected in traffic<br>• Alert generated |
| **Alternate flows** |
| • Low-rate scans may go undetected |
| **Post Condition** |
| • Detection result shown on Dashboard; threat log updated |

*Table 16: Use Case 8*

# Chapter 4
# System Design

# **Chapter 4:** System Design

This chapter explains the internal structure and design of the Heimdall system. It includes the layered architecture used to organize different system responsibilities and the domain model that outlines key entities, their attributes, and how they relate to each other. Together, these diagrams provide a complete understanding of how the system components interact to deliver real-time threat detection, retraining [15], and monitoring capabilities. This chapter serves as a blueprint for both implementation and future extension of the system.

## 4.1. Architecture Diagram

Heimdall follows a multi-tiered architecture consisting of four main layers: Presentation, Business Logic, Data Access, and Data Store. Each layer is responsible for specific tasks, enabling modular development and easier maintenance.

- **Tier 1 – Presentation Layer**: This includes the UI Framework built with Flask and front-end technologies. It provides users with real-time dashboards to view logs, threat summaries, and retraining status.
- **Tier 2 – Business Logic Layer**: This tier handles core processing. It includes the Kafka Consumer Service (for feature ingestion), the Model Manager (for prediction logic), the Metrics Collector (for Prometheus integration), and the Re-trainer Module (for model updates).
- **Tier 2 – Data Access Layer**: Interfaces between services and the data store. It includes the Kafka Producer (for feature publishing), Prometheus Exporter (for metrics exposure), and Logs Handler (for log persistence).
- **Tier 3 – Data Store Layer**: This tier holds persistent data such as the Kafka Topic stream, serialized ML model files (.pkl), and stored logs and system metrics.

This layered approach ensures that components can be developed and tested independently, improving system scalability and flexibility.

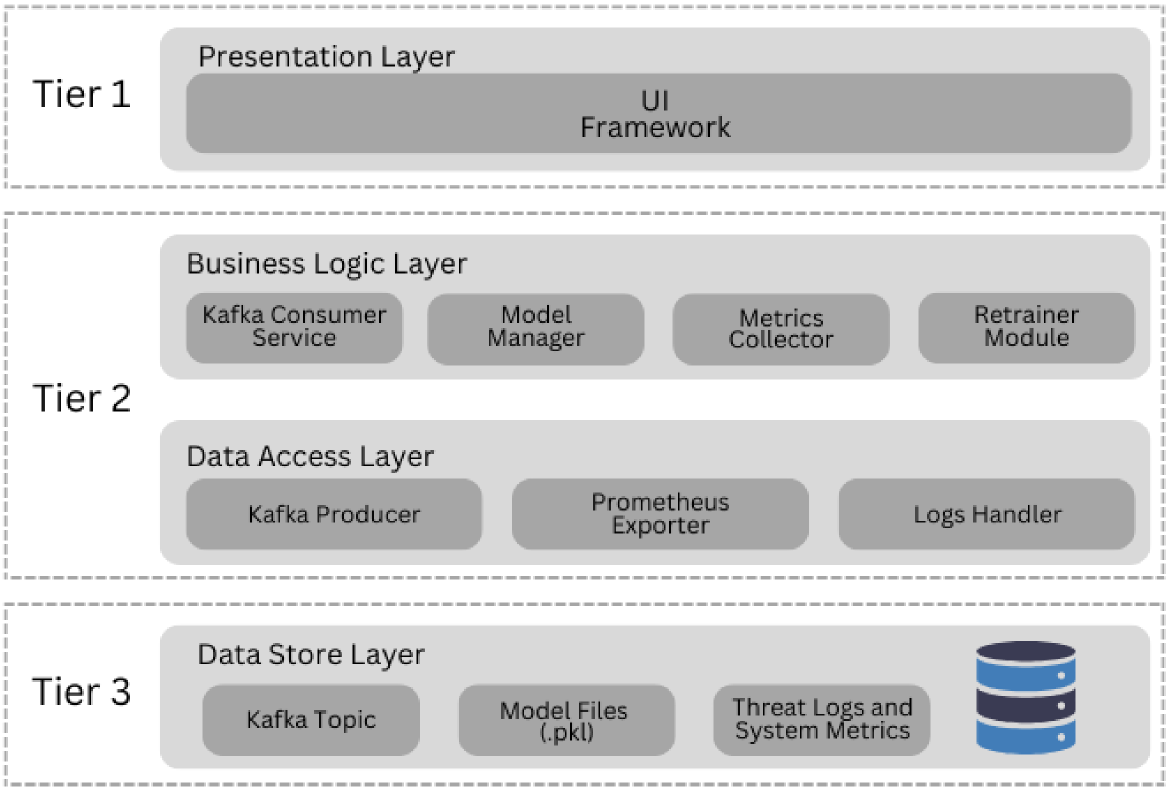


*Figure 12: Architecture Diagram*

## 4.2. Domain Model

The domain model of Heimdall captures all major entities involved in the system and their relationships. It reflects how data flows from raw packets to final predictions and metrics, and how different modules interact.

- **NetworkPacket** represents raw input traffic data.
- **FeatureVector** is the processed form of a packet, which gets published to a KafkaTopic for classification.
- **MLModel** is the trained machine learning model used for classification, which is periodically updated by **RetrainingEvent**.
- **ThreatPrediction** stores the classification result for each packet, including model details and timestamps.

- **Dashboard** displays threat statistics and allows retraining controls.
- **PrometheusMetrics** logs system performance metrics tied to specific system components.

The relationships between these entities show how Heimdall transforms data, applies machine learning, and makes information available for operators and monitoring tools.

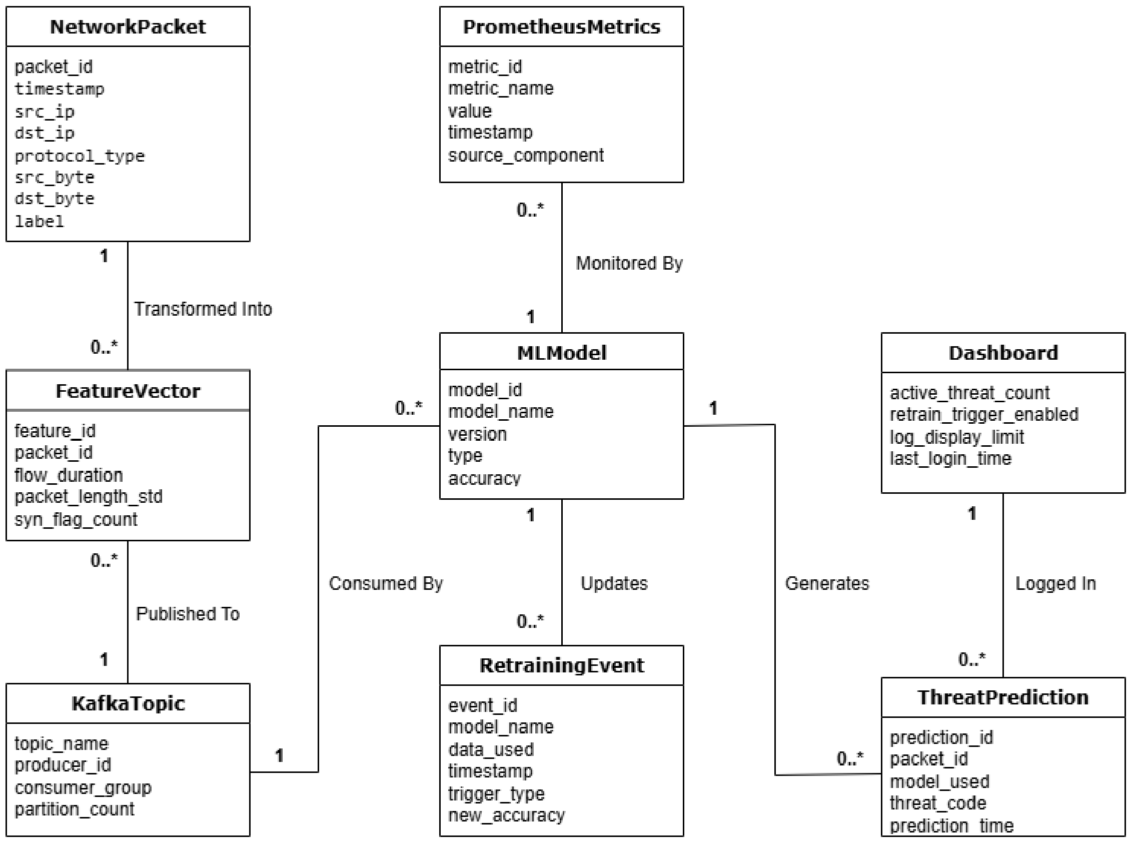


*Figure 13: Domain Model Diagram*

## 4.3. Entity Relationship Diagram with data dictionary

The Entity Relationship (ER) diagram illustrates the core entities and relationships that define Heimdall's internal data model. It visually represents how data flows from user interactions and network packets through the machine learning pipeline and into prediction and retraining processes.

**Entity Descriptions:**

- **User**: Represents the administrator or analyst interacting with the system. A user may manually trigger model retraining based on observed behavior or log analysis.
- **Packet**: The raw network data captured from the interface. Each packet is analyzed to generate relevant features and is ultimately classified by the system.
- **Feature**: These are the processed attributes derived from packets, such as byte size, protocol type, or flow duration. A single packet can produce multiple features.
- **Prediction**: The outcome of applying a machine learning model to packet features. Each prediction includes labels such as 'benign' or 'threat' and is linked to both the model and packet.
- **Model**: Represents a trained machine learning classifier used for inference. One model may produce many predictions and can be retrained over time.
- **Retraining Log**: Records every retraining event, including trigger source, timestamp, and new model metrics. Users may initiate retraining manually or it may occur automatically.

**Relationship Highlights:**

- A **Packet** generates one or more **Predictions** via a **Model**.
- Each **Model** can produce predictions and be retrained multiple times, with every event logged in the **Retraining Log**.
- A **User** can trigger multiple retraining sessions, linking them to decision-making activity.

Each **Packet** has a corresponding set of **Features**, which serve as the input for classification.

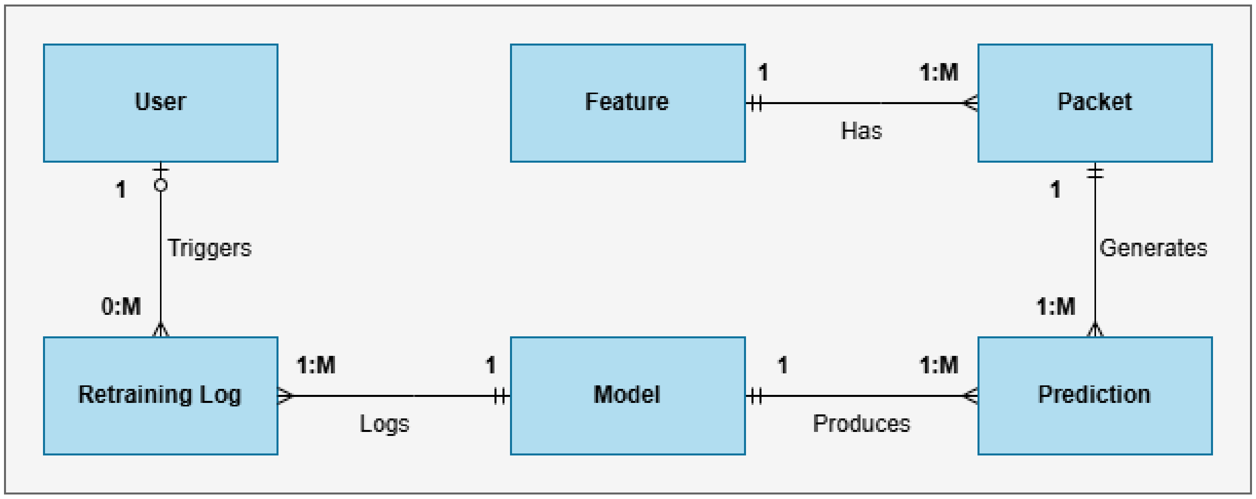


*Figure 14: ER Diagram*

## 4.4. Class Diagram

The class diagram illustrates the main classes and their relationships within the Heimdall system. It captures the internal structure of the software, showing how core components interact in real-time packet processing, threat classification, retraining, and monitoring.

**Overview of Key Classes:**

- **KafkaProducer**: Captures live network traffic and extracts features using the FeatureExtractor. Sends the structured data to Kafka for further processing.
- **FeatureExtractor**: Parses raw packet data and generates feature vectors suitable for input to machine learning models.
- **KafkaConsumerService**: Subscribes to Kafka topics, receives incoming packet features, and collaborates with one or more MLModel instances to classify traffic. It also manages retraining triggers.
- **MLModel**: Represents the machine learning logic for prediction and retraining. Includes metadata such as model name, type, and accuracy.
- **PrometheusMetrics**: Collects and exposes system metrics like threat counts and latency using Prometheus-compatible formats. Updated by the ML model and read by the dashboard.
- **ThreatLog**: Stores information about classified threats, including metadata like source and destination IPs, timestamp, and model used.

- **Dashboard**: Provides a user interface for reviewing logs, triggering retraining, and visualizing system metrics. Interacts with both PrometheusMetrics and ThreatLog.

**Class Relationships:**

- **KafkaProducer** sends data to **KafkaConsumerService**, which uses **MLModel** for classification.
- **FeatureExtractor** is used internally by the producer.
- **MLModel** updates **PrometheusMetrics**, which are accessed by the **Dashboard**.
- **KafkaConsumerService** also stores logs in **ThreatLog**, which the dashboard accesses for display.
- The system is structured for modularity, with clear boundaries between streaming, ML, logging, and visualization layers.

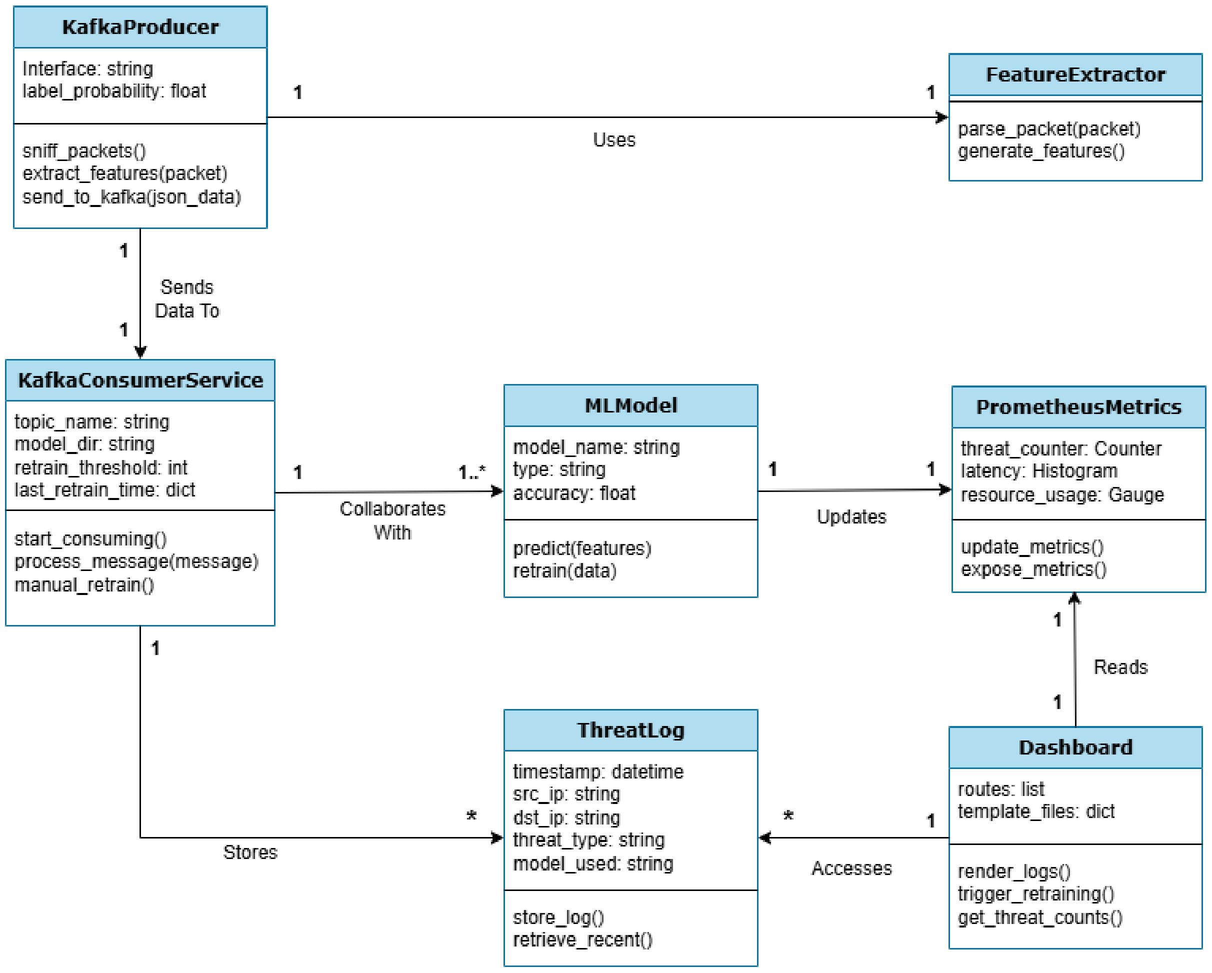


*Figure 15: Class Diagram*

## 4.5. Sequence / Collaboration Diagram

The sequence diagram in Figure 5 illustrates the flow of data and control across major components of the Heimdall system during a typical packet classification cycle.

The process begins when the **NetworkInterface** captures packets, which are forwarded to the **KafkaProducer** for feature extraction and transmission to the **KafkaBroker**. The **KafkaConsumerService** then polls for new messages, passes the extracted features to the **MLModel** for prediction, and logs the result via the **ThreatLogger**. Finally, the latest threat logs are fetched and displayed on the **Dashboard** for user visibility.

This flow highlights Heimdall's asynchronous, modular architecture that separates data capture, processing, logging, and visualization into loosely coupled components for scalability and maintainability.

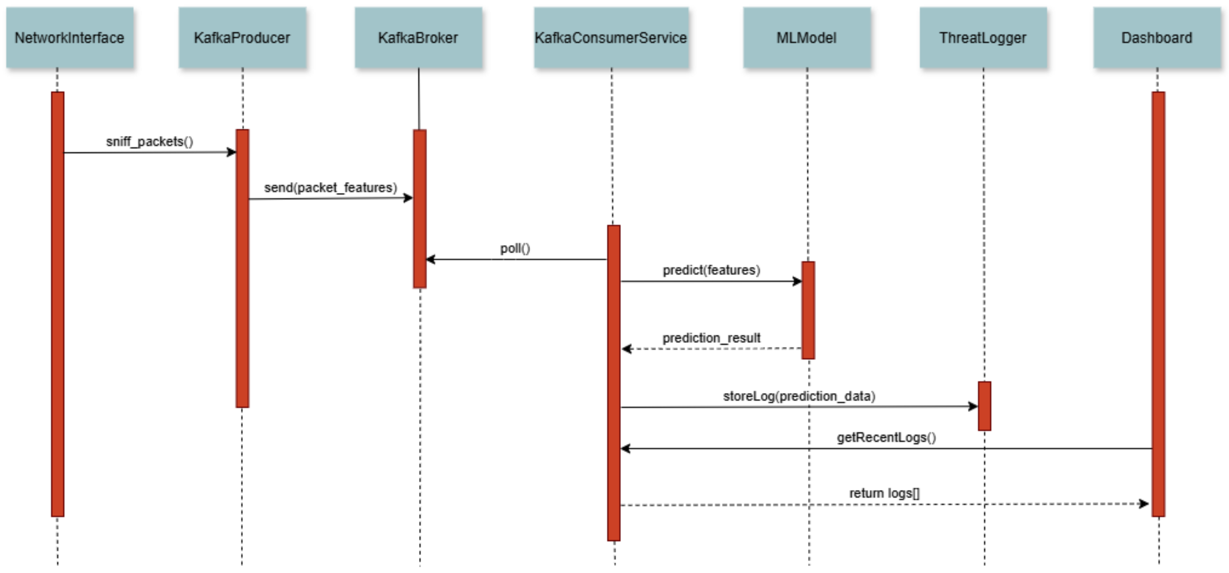


*Figure 16: Sequence Diagram*

## 4.6. Activity Diagram

The process begins with the activation of the Kafka producer, which captures network packets via Scapy and extracts relevant features. These are serialized into JSON and sent to a Kafka topic. The Kafka consumer retrieves these messages, preprocesses them, and uses preloaded ML models to predict whether the traffic is a threat.

Based on the prediction, packets are logged as either benign or threats. Metrics are updated accordingly. If enough labeled data accumulates or manual retraining is triggered, the system initiates model retraining, updates the model's accuracy, and saves the new version.

The dashboard component fetches the logs using Flask and displays the threat records to users, completing the cycle.

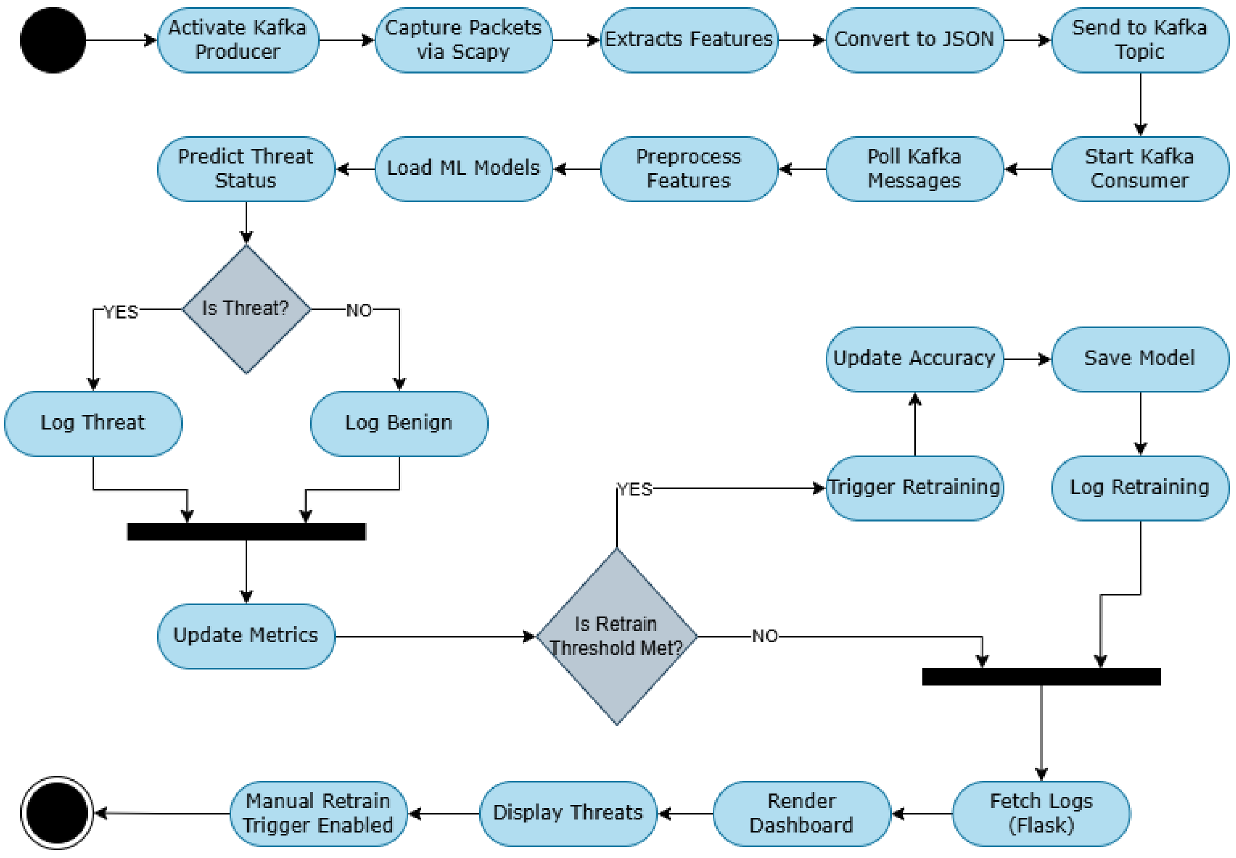


*Figure 17: Activity Diagram*

## 4.7. State Transition Diagram

The system begins in the **Initialized** state once the start command is issued. It then transitions to **Capturing**, where packet capture begins, and Kafka producers start streaming data. The system moves into the **Processing** state to classify packets in real time using machine learning models. From the processing state, Heimdall can either:

- Enter the **Paused** state when a pause command is issued (e.g., for temporary maintenance),
- Or transition into **Retraining** if triggered by a data threshold or manual action.

Once retraining is complete, the system returns to processing seamlessly. At any point, a **Stopped** state can be reached by issuing a stop command, which shuts down all services.

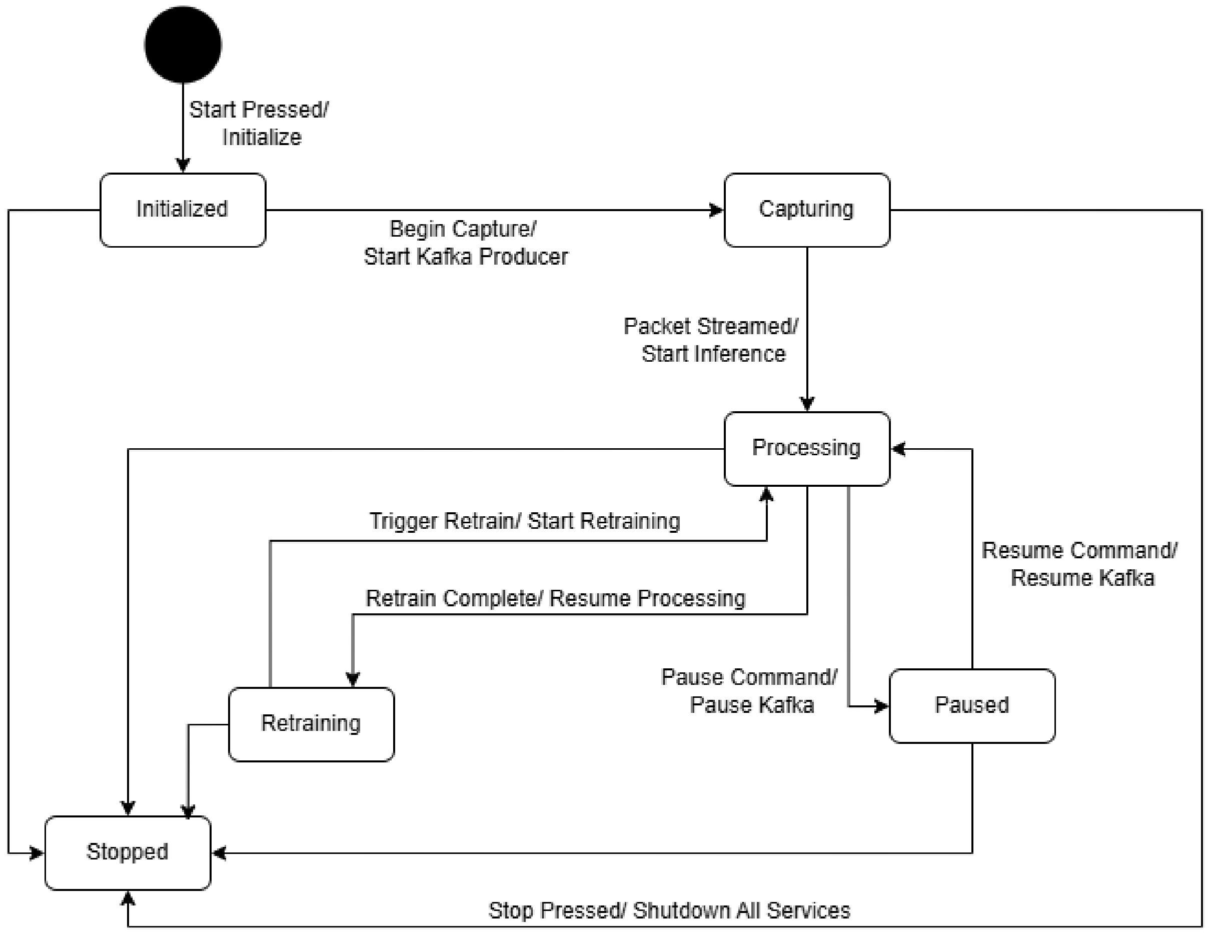


*Figure 18: State Transition Diagram*

## 4.8. Component Diagram

The component diagram illustrates the modular architecture of Heimdall, outlining how different subsystems interact to deliver real-time intrusion detection and adaptive learning. Each component is encapsulated within the Heimdall Terminal environment and communicates through defined interfaces.

- The **KafkaProducer** captures features from the **NetworkSniffer** and forwards them to the **KafkaBrokerInterface**.
- The **KafkaConsumer** processes incoming feature vectors and passes them to the **MLModel** component, which performs threat classification.
- The **PrometheusExporter** collects system and model metrics, which are later visualized using **GrafanaVisualizer**.
- **ThreatLogger** logs all prediction outputs, which are accessed by both the **GrafanaVisualizer** and **HeimdallTerminal** via the **ThreatLog** interface.

Components such as **RetrainingService** and **MetricsAPI** manage model updates and system performance monitoring, ensuring that Heimdall adapts continuously and operates reliably.

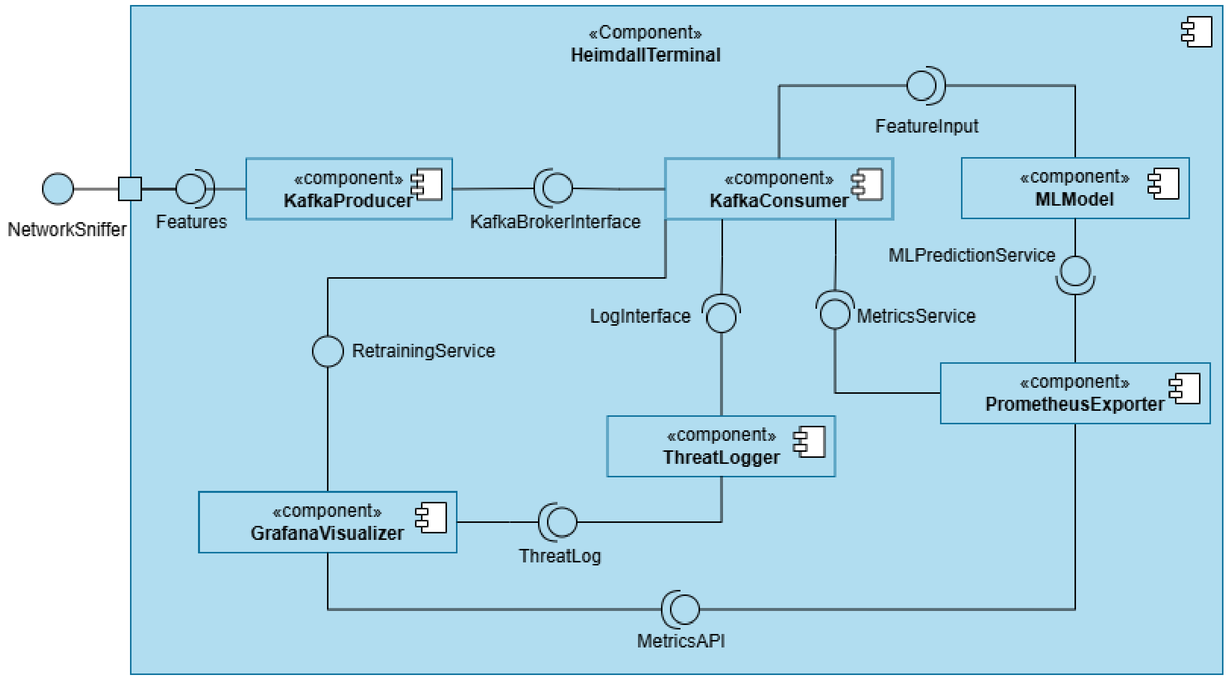


*Figure 19: Component Diagram*

## 4.9. Deployment Diagram

At the core is the **Primary Server**, which hosts all essential components including the Kafka Broker, Producer and Consumer modules, ML models, Prometheus, and Grafana. This server is connected to the **Local Network** via Ethernet and acts as the central processing and monitoring hub.

**Analyst Workstations** are connected to the same local network and communicate with the server over HTTP to access the dashboard and logs. These workstations are typically used by security analysts for monitoring and management.

All network traffic passes through a **Firewall** before reaching the local network, ensuring secure and controlled access. This setup supports internal monitoring while maintaining isolation from external networks.

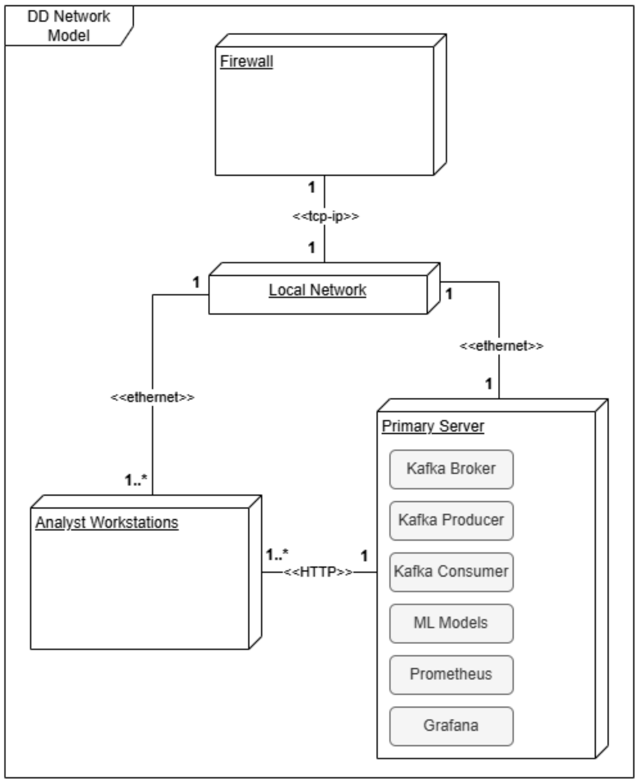


*Figure 20: Deployment Diagram*

## 4.10. Data Flow diagram

The Data Flow Diagram (DFD) illustrates the overall flow of data within the Heimdall system. It captures how network traffic is ingested, processed, and visualized by different components. The process begins with the capture of raw packets from the network, which are then streamed to a Kafka topic. These packets undergo feature extraction and are converted into structured feature vectors in JSON format.

The consumer module retrieves and normalizes these vectors to perform threat classification. The results, including detected threats, are stored in the threat log and made available for visualization on the dashboard. Security analysts can manually initiate retraining if needed, and the system metrics are continuously monitored and visualized using Prometheus and Grafana. The diagram also differentiates between three primary data types: raw traffic (blue), feature vectors (green), and logs/metrics (red), ensuring clear traceability of data movement across the system.

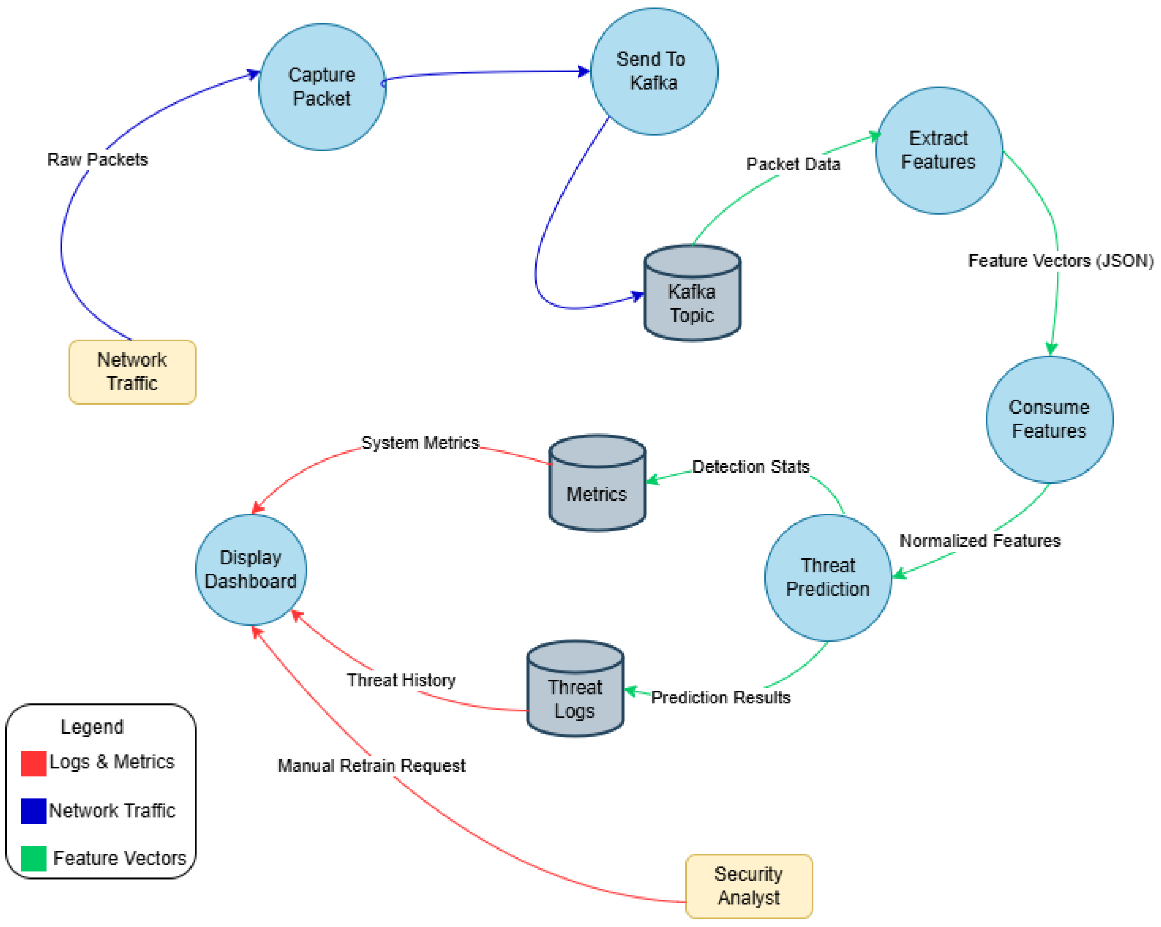


*Figure 21: Data Flow Diagram*

# Chapter 5
# **Implementation**

# **Chapter 5:** Implementation

This chapter outlines the detailed implementation of Heimdall, a real-time AI-driven intrusion detection system. It discusses how the system was structured using a modular and layered approach, the sequence of data flow and control between components, and how various open-source tools and technologies were integrated to achieve an efficient and scalable solution. The chapter also describes the development environment, libraries used, system deployment, best practices followed during development, and the version control approach maintained throughout the project lifecycle.

## 5.1. Important Flow Control/Pseudo codes

The Heimdall system is designed using an event-driven architecture. Each component of the system operates independently but communicates efficiently through Kafka messaging. The main operational flow is as follows:

1. **Packet Capture**: The system uses Scapy to capture raw network packets at the NIC level.
2. **Feature Extraction**: Each packet is parsed and transformed into a numerical feature vector.
3. **Streaming via Kafka**: The feature vector is serialized and published to a Kafka topic by the producer.
4. **Model Prediction**: The Kafka consumer reads each vector and applies the trained ML model to classify the traffic.
5. **Result Logging & Display**: Classification outcomes are logged and visualized in real time on the dashboard.
6. **System Monitoring**: Performance metrics are collected and exposed to Prometheus.
7. **Retraining Logic**: A separate module monitors prediction confidence and new labeled data. When configured thresholds are met, the system retrains the model in the background.

Sample pseudocode for the classification pipeline:

```
# Producer Flow
while True:
    packet = sniff_packet()
    features = extract_features(packet)
    kafka_producer.send(features)

# Consumer Flow
for message in kafka_consumer:
    prediction = ml_model.predict(message)
    log_prediction(prediction)
    update_dashboard(prediction)
```

This clear flow ensures data is continuously processed and decisions are made in near real-time, with retraining functioning as a background task.

## 5.2. Components, Libraries, Web Services and stubs

Heimdall system includes a variety of components, each written in Python and built to perform specialized tasks. The key modules and libraries are:

- **Packet Capture Module**: Uses Scapy to sniff packets on the default interface, filtering and preprocessing raw traffic.
- **Feature Extraction & Kafka Producer**: A custom module that converts packets into structured feature vectors and uses the confluent-kafka library to publish them to a topic.
- **Kafka Consumer & ML Prediction**: Implements the message consumer using confluent-kafka. Applies a scikit-learn model (Random Forest or Decision Tree) loaded via joblib to predict threats.
- **Web Dashboard**: Developed using Flask as the web server. Front-end elements include HTML5, Bootstrap, and Chart.js to display threat statistics and retraining logs.
- **Retraining Module**: Monitors prediction accuracy and data volume. When thresholds are met, it retrains the model without halting active traffic analysis.
- **Prometheus Monitoring**: The prometheus_client library is used to expose system metrics like packet throughput, retraining count, and error rates.

The system does not rely on any external APIs or cloud services. All components run locally, ensuring control, privacy, and low-latency performance.

## 5.3. Deployment Environment

The development and deployment of Heimdall were conducted in a controlled Linux environment to ensure compatibility, security, and repeatability. Below are the specific configuration details:

- **Operating System**: Ubuntu 20.04 LTS (64-bit)
- **Programming Language**: Python 3.8+
- **Message Broker**: Apache Kafka 2.13+
- **Web Server**: Flask
- **Monitoring Stack**: Prometheus (scraping metrics via HTTP)
- **Browser Compatibility**: Verified using Google Chrome
- **Containerization Tools**: Docker and Docker Compose for bundled deployment

The system was evaluated first on a laptop with 8GB RAM and a quad-core Intel CPU. It was later deployed to a virtual machine to ensure portability across environments and operating systems.

## 5.4. Tools and Techniques

A range of tools and software engineering practices were applied to develop Heimdall efficiently:

- **Python**: The primary programming language used for backend services and ML logic.
- **Visual Studio Code**: The integrated development environment (IDE) used throughout the project.
- **Apache Kafka**: Managed all message streaming between producers and consumers.
- **Flask**: Lightweight web framework used for dashboard backend and API endpoints.
- **Scikit-learn**: Employed for training ML models and exporting them as .pkl files.
- **Prometheus + Grafana**: Monitored system performance and visualized real-time metrics.
- **Docker**: Containerized all services to create an easily reproducible deployment pipeline.

The team followed an **iterative and test-driven development approach**, prioritizing early validation, modular integration, and continuous improvement.

## 5.5. Best Practices / Coding Standards

To ensure code clarity, scalability, and maintainability, several best practices were followed:

- Adherence to **PEP8** Python style guidelines
- Modular file structure: separate scripts for producer, consumer, re-trainer, and UI
- Use of **meaningful variable names**, **docstrings**, and **inline comments**
- Error handling for Kafka disconnections, file access issues, and JSON parsing
- Avoidance of hardcoded paths by using configuration files
- Testing of each module before integration

This disciplined structure ensured that the system remained stable, readable, and easy to debug during development and testing phases

## 5.6. Version Control

Version control played a central role in maintaining the stability, traceability, and collaborative workflow of the Heimdall project. Git was used as the primary version control tool, with changes pushed to a private GitHub repository.

Key practices included:

- **Feature Branching**: Separate branches were created for each feature or bug fix.
- **Commit History**: Each commit included descriptive messages to document progress.
- **Tagging**: Important versions were tagged to mark milestone builds.

Key tagged versions:

- **v0.1**: Initial Kafka producer-consumer communication established
- **v0.5**: Integration of ML inference pipeline with real packet data
- **v0.9**: Functional dashboard with live prediction logs and manual retraining
- **v1.0**: Final stable version including Prometheus support and modular deployment via Docker

The use of Git enabled smooth collaboration, easy rollback when bugs were encountered, and clear documentation of system evolution.

# Chapter 6
# **Business Plan**

# **Chapter 6:** Business Plan

This chapter presents a comprehensive business roadmap for Heimdall, transitioning it from a final-year project into a commercially viable cybersecurity product. The following sections detail Heimdall’s market positioning, target users, core and extended offerings, revenue strategies, and potential business challenges. The business plan also includes a Business Model Canvas (Figure 4), outlining Heimdall's ecosystem of partners, resources, customer segments, and cost structure. This strategic plan supports the long-term vision of Heimdall becoming a scalable and sustainable solution in the evolving cybersecurity landscape.

## 6.1 Business Description

Heimdall is an AI-powered intrusion detection system designed to identify real-time threats using machine learning. Unlike traditional signature-based systems that rely on known attack patterns, Heimdall uses adaptive learning to monitor network traffic and detect anomalies. Through automated retraining, the system evolves with the network it monitors, making it suitable for dynamic environments.

From a business standpoint, Heimdall is positioned as a modular cybersecurity product that can be offered in three deployment models:

- **On-Premises Deployment**: For organizations with internal IT teams seeking control and customization.
- **Cloud-Based SaaS (Planned)**: For service providers needing scalable, multi-user monitoring and alerting.
- **Modular SDK for Integration**: For MSSPs and researchers looking to embed threat detection into broader platforms.

Heimdall's initial commercial version will offer local deployment capabilities including real-time monitoring, dashboard visualization, retraining management, and threat alerts. This would later evolve into a fully hosted platform with role-based access control, remote logs, and threat feed integration.

## 6.2 Market Analysis & Strategy

The global demand for intelligent and cost-effective cybersecurity tools is rising. While large enterprises may invest in proprietary solutions like QRadar or Splunk, SMEs, educational institutions, and managed service providers often cannot afford such platforms. Heimdall directly addresses this gap.

**Target Market Segments:**

- **Small and Medium Enterprises (SMEs)** needing affordable automated security tools
- **Universities and Colleges** seeking educational or internal network monitoring systems
- **Cybersecurity Researchers** exploring adaptive threat models
- **Managed Security Service Providers (MSSPs)** requiring rebrand able, modular solutions

**Go-To-Market Strategy:**

1. **Pilot Programs:** Launch a minimal viable product (MVP) version through partnerships with universities and labs.
2. **Community Engagement:** Leverage open-source-style adoption to gather feedback and increase visibility.
3. **SME Licensing:** Introduce on-premises deployment packages for small businesses with basic support tiers.
4. **SaaS Platform Rollout:** Offer cloud-hosted services with customizable dashboards and alert systems.
5. **Channel Partnerships:** Collaborate with IT vendors and regional resellers for bundled security packages.

**Promotional Channels:**

- Web presence and online demos
- Industry-specific social media campaigns
- Participation in cybersecurity workshops and conferences
- Sponsored research or hackathons

## 6.3 Competitive Analysis

The intrusion detection space includes a mix of legacy and modern solutions. Signature-based tools like **Snort** and **Suricata** are widely used but require manual rule updates and cannot detect novel attacks. More modern platforms like **Zeek** or **Security Onion** offer extended analysis capabilities but are often complex to deploy and configure. Commercial tools like **Splunk**, **Darktrace**, or **QRadar** provide advanced analytics but are cost-prohibitive for smaller organizations.

**Heimdall's Differentiation:**

- **Adaptive learning**: Automatically retrains models based on feedback or performance thresholds.
- **Real-time classification**: Immediate insight into network behavior.
- **Modular architecture**: Easy to update or extend without reconfiguring the whole system.
- **Open-source and cost-effective**: Suitable for labs, SMEs, and educational environments.
- **Dashboard + Metrics Integration**: Combines observability and control in one unified UI.

By focusing on simplicity, automation, and affordability, Heimdall fills a gap between open-source complexity and commercial pricing.

## 6.4 Products/Services Description

**Current Product Offering:**

- **Packet Capture and Analysis Engine**: Captures real-time network traffic and extracts features.
- **ML Threat Classifiers**: Pretrained and retrainable classifiers to detect anomalies.
- **Kafka-Based Streaming Pipeline**: Ensures scalable, decoupled processing of traffic.
- **Monitoring Dashboard**: Built with Flask, displays live predictions and retraining triggers.
- **Prometheus Integration**: Enables performance metric tracking and observability.

**Future Service Extensions:**

- **SaaS Dashboard**: Managed cloud-based instance with multi-tenant user support and REST APIs.
- **Threat Notification Module**: Integrations for email, Slack, or webhook alerts.
- **Customized Model Training**: Enterprise clients can train models on proprietary datasets.
- **White-Labeling**: Enable MSSPs to customize branding and deploy Heimdall as their own solution.
- **Compliance & Storage Add-ons**: Integration with data storage for retention and auditing.

**Support Tiers:**

- Academic (open source)

- SME (basic + support)
- Enterprise (custom + integrations)

## 6.5 SWOT Analysis

SWOT analysis provides a deep evaluation of Heimdall’s strategic position. It assesses internal strengths and weaknesses, as well as the external opportunities and threats that may influence the platform’s growth and sustainability.

**Strengths:**

- **Adaptive ML Core**: Heimdall’s machine learning models can be retrained automatically, which provides resilience against evolving cyber threats and eliminates dependence on signature updates.
- **Lightweight, Modular Architecture**: Built using containerized services and open-source libraries, the system is easy to deploy and scale across a variety of infrastructures.
- **No Third-Party Dependency**: Heimdall does not rely on cloud APIs or external analytics engines, which enhances data privacy and reduces operational cost.
- **Comprehensive Dashboard**: The unified Flask-based interface offers visual insight into real-time predictions, system health, and retraining events—suitable even for non-expert users.
- **Cost-Effective Development Model**: Open-source stack (Python, Kafka, Prometheus) lowers both development and licensing costs.

**Weaknesses:**

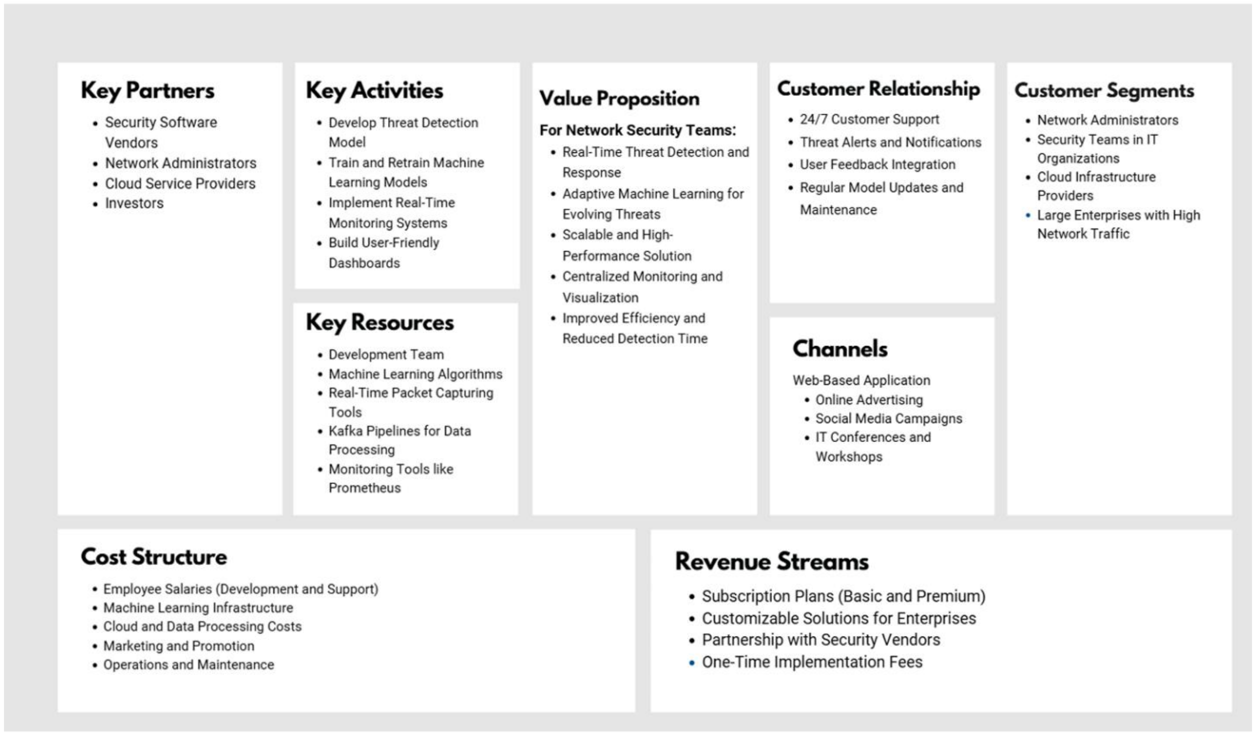
- **Lack of Encrypted Traffic Analysis**: Current capabilities are limited to unencrypted packet inspection. Encrypted traffic remains opaque, which may reduce detection accuracy in modern TLS-based networks.
- **Basic User Management**: The dashboard lacks features such as login authentication, multi-user support, or audit trails—limiting use in regulated or multi-admin environments.
- **Data Persistence Gaps**: Logs are stored temporarily or in CSV/JSON formats. There is no support yet for structured databases (e.g., PostgreSQL) or cloud storage integration.
- **Initial Learning Curve**: While modular, the system still requires technical expertise for deployment, especially around Kafka setup, Docker orchestration, and custom retraining configurations.

**Opportunities:**

- **SaaS Expansion**: Hosting Heimdall as a cloud-native service would allow subscription-based access for a global audience, with tiered pricing and centralized updates.
- **Enterprise Integration**: Opportunities exist to integrate Heimdall into larger SIEM (Security Information and Event Management) platforms or security operations centers (SOCs).
- **IoT and ICS Applications**: Lightweight deployment and real-time classification make Heimdall a candidate for use in smart factories, utilities, and embedded industrial systems.
- **Academic Partnerships**: Heimdall can serve as a research sandbox or curriculum tool for universities offering cybersecurity or machine learning programs.
- **Threat Intelligence Add-ons**: Incorporating external data feeds (e.g., threat signatures, IP blacklists) could improve detection without sacrificing learning capability.

**Threats:**

- **Intense Competition**: Market is saturated with mature products like Splunk, QRadar, and Darktrace, which offer integrated analytics, visualization, and threat response automation.
- **Regulatory Pressure**: Emerging data protection laws (e.g., GDPR, CCPA) may introduce compliance challenges, especially for organizations capturing packet-level data.
- **Technical Misuse**: Improper configuration or model misuse (e.g., overfitting during retraining) can lead to security blind spots or false positives.
- **Scalability Risks**: Kafka and Prometheus components must be finely tuned for high-volume networks; performance degradation at scale could impact adoption among enterprise clients.



**Key Partners**
- Security Software Vendors
- Network Administrators
- Cloud Service Providers
- Investors

**Key Activities**
- Develop Threat Detection Model
- Train and Retrain Machine Learning Models
- Implement Real-Time Monitoring Systems
- Build User-Friendly Dashboards

**Key Resources**
- Development Team
- Machine Learning Algorithms
- Real-Time Packet Capturing Tools
- Kafka Pipelines for Data Processing
- Monitoring Tools like Prometheus

**Value Proposition**

**For Network Security Teams:**
- Real-Time Threat Detection and Response
- Adaptive Machine Learning for Evolving Threats
- Scalable and High-Performance Solution
- Centralized Monitoring and Visualization
- Improved Efficiency and Reduced Detection Time

**Customer Relationship**
- 24/7 Customer Support
- Threat Alerts and Notifications
- User Feedback Integration
- Regular Model Updates and Maintenance

**Channels**

Web-Based Application
- Online Advertising
- Social Media Campaigns
- IT Conferences and Workshops

**Customer Segments**
- Network Administrators
- Security Teams in IT Organizations
- Cloud Infrastructure Providers
- Large Enterprises with High Network Traffic

**Cost Structure**
- Employee Salaries (Development and Support)
- Machine Learning Infrastructure
- Cloud and Data Processing Costs
- Marketing and Promotion
- Operations and Maintenance

**Revenue Streams**
- Subscription Plans (Basic and Premium)
- Customizable Solutions for Enterprises
- Partnership with Security Vendors
- One-Time Implementation Fees

*Figure 22: Business Plan*

# Chapter 7
# Testing & Evaluation

# **Chapter 7:** Testing and Evaluation

This chapter outlines the verification and validation procedures undertaken to ensure the Heimdall threat detection system operates effectively, efficiently, and reliably. Each component was rigorously tested against predefined criteria, including functional correctness, boundary resilience, data integrity, and performance under real-world and synthetic loads. Tests included equivalence partitioning, boundary value analysis, data flow validation, unit and integration tests, and stress performance evaluations. Detailed results and supporting artifacts were collected for documentation and defense presentation.

## 7.1 Use Case Testing (Test Cases)

### Test Case - 1

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_01 | **Test Case Description** | Test the real-time network traffic logging and predictions. | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Test validated end-to-end packet flow from Scapy producer to Kafka, consumer classification, and dashboard display. Verified correct JSON format, real-time predictions, and metrics exposure via /metrics. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 10-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Kafka, Flask, Prometheus running |
| 2 | Network interface configured (eth0) |

| S # | Test Data |
|---|---|
| 1 | Captured packet with (label = 1) |
| 2 | Valid TCP packet with HTTP signature |

| | |
|---|---|
| **Test Scenario** | System should capture, classify, and log network packets in real time. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Start producer on eth0 | Packets sent to Kafka | As Expected | Pass |
| 2 | Start consumer and Flask | Packets consumed and processed | As Expected | Pass |
| 3 | Open dashboard | Logs display threat | As Expected | Pass |

*Table 17: Test Case 1*

## Test Case - 2

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_02 | **Test Case Description** | DoS Attack Detection | | |
| **Created By** | Ahsan Ahmed | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Used hping3 for TCP SYN flood simulation. System identified DoS pattern correctly. Dashboard alert and Prometheus metric clf_DoS increment confirmed. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Ahsan Ahmed | **Date Tested** | 12-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | DoS model deployed |

| S # | Test Data |
|---|---|
| 1 | SYN flood pattern (label=2) |

| | |
|---|---|
| **Test Scenario** | Detect DoS attack and reflect in dashboard and metrics. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Send SYN flood packets | Traffic flagged as DoS | As Expected | Pass |
| 2 | View dashboard | DoS threat visible | As Expected | Pass |
| 3 | View Prometheus metrics | clf_DoS counter incremented | As Expected | Pass |

*Table 18: Test Case 2*

## Test Case - 3

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_03 | **Test Case Description** | Web Attack Detection | | |
| **Created By** | Syed Ali Zain | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Simulated SQL injection traffic over HTTP. Consumer flagged traffic, reflected in logs and Prometheus as clf_WebAttacks. Output matched expected classification label. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Syed Ali Zain | **Date Tested** | 15-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Web attack model deployed; dashboard live |

| S # | Test Data |
|---|---|
| 1 | SQL injection payload, XSS string (label=6) |

| | |
|---|---|
| **Test Scenario** | Identify HTTP-based attacks and show results in logs and metrics. |

| Step # | Step Details | Expected Results | Actual Results | |
|---|---|---|---|---|

| | | | | **Pass / Fail / Not executed / Suspended** |
|---|---|---|---|---|
| 1 | Inject SQL via GET request | Detected as Web Attack | As Expected | Pass |
| 2 | Check dashboard logs | Threat entry: 'Web attack detected' | As Expected | Pass |
| 3 | Monitor Prometheus | Threat entry: 'Web attack detected' | As Expected | Pass |

*Table 19: Test Case 3*

## Test Case - 4

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_04 | **Test Case Description** | Bot Attack Detection | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | DNS beaconing emulated using replay script. clf_Bot detection triggered correctly. Dashboard updated and threat log validated. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 17-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| **S #** | **Prerequisites:** |
|---|---|
| 1 | clf_Bot model deployed |

| **S #** | **Test Data** |
|---|---|
| 1 | DNS beaconing traffic (label=1) |

| | |
|---|---|
| **Test Scenario** | Detect botnet traffic based on command-and-control patterns. |

| **Step #** | **Step Details** | **Expected Results** | **Actual Results** | **Pass / Fail / Not executed / Suspended** |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | Simulate beaconing DNS packets | Flagged as bot activity | As Expected | Pass |
| 2 | View dashboard | Bot threat appears | As Expected | Pass |
| 3 | Inspect Prometheus metrics | clf_Bot counter incremented | As Expected | Pass |

*Table 20: Test Case 4*

## Test Case - 5

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_05 | **Test Case Description** | Zero-Day Attack Detection | | |
| **Created By** | Ahsan Ahmed | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Injected random, non-profiled network traffic. Adaptive logic-initiated retraining after 50+ labeled entries. Logs and new .pkl file timestamp validated retrain event. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Ahsan Ahmed | **Date Tested** | 18-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Adaptive learning active |

| S # | Test Data |
|---|---|
| 1 | Unknown traffic pattern (label=7) |

| | |
|---|---|
| **Test Scenario** | Detect and optionally retrain on novel attack patterns |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Send unfamiliar protocol traffic | Detected as anomaly | As Expected | Pass |

| | | | | |
|---|---|---|---|---|
| 2 | Review dashboard logs | Threat entry for zero-day | As Expected | Pass |
| 3 | Check for retraining | Retraining initiated if threshold | As Expected | Pass |

*Table 21: Test Case 5*

## Test Case - 6

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_06 | **Test Case Description** | Brute Force Attack Detection | | |
| **Created By** | Syed Ali Zain | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Simulated brute force using repetitive failed login entries. clf_BruteForce model flagged threat; logs confirmed correct classification. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Syed Ali Zain | **Date Tested** | 19-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | clf_BruteForce model active |

| S # | Test Data |
|---|---|
| 1 | Multiple failed login attempts (label=5) |

| | |
|---|---|
| **Test Scenario** | Detect and optionally retrain on novel attack patterns. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Simulate login failures from same IP | Flagged as BruteF orce attack | As Expected | Pass |
| 2 | Check logs | BruteForce entry listed | As Expected | Pass |

| | | | | |
|---|---|---|---|---|
| 3 | Review metrics | clf_BruteForce counter updated | As Expected | Pass |

*Table 22: Test Case 6*

## Test Case - 7

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | TC_07 | **Test Case Description** | Malware Attack Detection | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Replayed malware communication patterns from public PCAPs. NSL-KDD model returned label 7, and threat logged correctly with detection message. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 20-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Malware-aware model active |

| S # | Test Data |
|---|---|
| 1 | Known malware pattern (label=7) |

| | |
|---|---|
| **Test Scenario** | Detect malware and update all logs and dashboards. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Simulate malware communication | Classified as malware threat | As Expected | Not executed |
| 2 | Check dashboard | Entry visible with timestamp | As Expected | Pass |
| 3 | View Prometheus metrics | Threat metrics updated | As Expected | Pass |

*Table 23: Test Case 7*

## Test Case - 8

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_01 | **Test Case Description** | Test packet capture from a valid network interface | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Packet sniffing verified using eth0. JSON packets produced and forwarded to Kafka topic network_logs. Logs confirmed Kafka delivery success. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 10-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Scapy installed, eth0 available |

| S # | Test Data |
|---|---|
| 1 | Live network traffic on eth0 |

| | |
|---|---|
| **Test Scenario** | Ensure the system captures packets via Scapy from a valid interface and sends them to Kafka. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Run kafka_producer.py --iface=eth0 | Starts sniffing packets | As Expected | Pass |
| 2 | Generate network traffic | Packet captured and feature extraction runs | As Expected | Pass |
| 3 | Monitor Kafka topic | JSON data appears in 'network_logs' | As Expected | Pass |

*Table 24: Test Case 8*

## Test Case - 9

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_02 | **Test Case Description** | Handle invalid network interface gracefully | | |
| **Created By** | Ahsan Ahmed | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Passed non-existent interface ethX. Producer exited with error message as expected without crash. Confirmed graceful termination. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Ahsan Ahmed | **Date Tested** | 11-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | System with no ethX interface |

| S # | Test Data |
|---|---|
| 1 | CLI arg: --iface=ethX |

| | |
|---|---|
| **Test Scenario** | Check if an invalid network interface is rejected and logs proper errors. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Run kafka_producer.py --iface=eth0 | System prints error in console | As Expected | Pass |
| 2 | Open log file | Error message is logged | As Expected | Pass |
| 3 | Check if sniffing started | No packets are captured | As Expected | Pass |

*Table 25: Test Case 9*

## Test Case - 10

| | | | |
|---|---|---|---|
| **Test Case ID** | INF_03 | **Test Case Description** | Kafka producer sends data correctly to topic |

| | | | | | |
|---|---|---|---|---|---|
| **Created By** | Syed Ali Zain | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Produced test packets and confirmed Kafka delivery. Consumer successfully received and processed the messages. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Syed Ali Zain | **Date Tested** | 12-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Kafka running on localhost:9092 |

| S # | Test Data |
|---|---|
| 1 | JSON packet with full feature set |

| | |
|---|---|
| **Test Scenario** | Validate that the producer publishes extracted features to Kafka topic successfully. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Send test packet via producer | Message is created and serialized | As Expected | Pass |
| 2 | Open Kafka CLI or monitoring tool | Message appears in 'network_logs' topic | As Expected | Pass |
| 3 | Check delivery report logs | Success message appears in debug logs | As Expected | Pass |

*Table 26: Test Case 10*

## Test Case - 11

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_04 | **Test Case Description** | Kafka broker unavailable scenario | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Kafka broker shut down during test. Producer logged delivery failures, then recovered automatically after broker was restarted. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 13-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Kafka broker stopped |

| S # | Test Data |
|---|---|
| 1 | Any packet data to be sent |

| | |
|---|---|
| **Test Scenario** | Ensure producer fails gracefully when Kafka is down. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Stop Kafka service | Confirm broker is unreachable | As Expected | Pass |
| 2 | Run producer and send message | Error is logged, system does not crash | As Expected | Pass |
| 3 | Restart Kafka | Producer reconnects and resumes send | As Expected | Pass |

*Table 27: Test Case 11*

## Test Case - 12

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_05 | **Test Case Description** | Verify /metrics endpoint for Prometheus | | |
| **Created By** | Ahsan Ahmed | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Accessed /metrics endpoint and confirmed availability of key metrics (total_threats, |

| | |
|---|---|
| | cpu_usage_percent, clf_DoS). Valid Prometheus scrape format verified. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Ahsan Ahmed | **Date Tested** | 14-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Flask server running |

| S # | Test Data |
|---|---|
| 1 | None |

| | |
|---|---|
| **Test Scenario** | Validate that the Prometheus endpoint /metrics exposes system metrics in valid format. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Access http://localhost:5000/metrics | Response is Prometheus-formatted text | As Expected | Pass |
| 2 | Search for known metrics | Counters like total_threats are visible | As Expected | Pass |
| 3 | Simulate threat | Counter increments after detection | As Expected | Pass |

*Table 28: Test Case 12*

## Test Case - 13

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_06 | **Test Case Description** | Adaptive retraining triggered by data threshold | | |
| **Created By** | Syed Ali Zain | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Injected 50+ labeled packets for clf_Bot. Adaptive model retraining triggered, confirmed by model update and log entries. Metrics confirmed retrain event. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Syed Ali Zain | **Date Tested** | 15-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Adaptive model available, retrain threshold=50 |

| S # | Test Data |
|---|---|
| 1 | 50+ labeled data points with label=1 |

| | |
|---|---|
| **Test Scenario** | Confirm that adaptive retraining is triggered when the retrain data queue reaches the set threshold. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Send 50 labeled packets to Kafka | Consumer collects data | As Expected | Pass |
| 2 | Monitor logs | Retraining thread starts | As Expected | Pass |
| 3 | Check model directory | New _adaptive.pkl file is saved | As Expected | Pass |

*Table 29: Test Case 13*

## Test Case - 14

| | | | | | |
|---|---|---|---|---|---|
| **Test Case ID** | INF_07 | **Test Case Description** | Manual retraining via dashboard endpoint | | |
| **Created By** | Mubahil Ahmad | **Reviewed By** | Dr. Zunera Jalil | **Version** | 1.0 |

| | |
|---|---|
| **QA Tester's Log** | Sent POST request to /retrain. Response confirmed retraining started. Verified retrain logs and updated adaptive model .pkl file in models/ directory. |

| | | | | | |
|---|---|---|---|---|---|
| **Tester's Name** | Mubahil Ahmad | **Date Tested** | 16-Mar-2025 | **Test Case (Pass/Fail/Not Executed)** | Pass |

| S # | Prerequisites: |
|---|---|
| 1 | Flask dashboard live |

| S # | Test Data |
|---|---|
| 1 | POST request to /retrain |

| | |
|---|---|
| **Test Scenario** | Check that clicking "Manual Retrain" or calling the API starts retraining without errors. |

| Step # | Step Details | Expected Results | Actual Results | Pass / Fail / Not executed / Suspended |
|---|---|---|---|---|
| 1 | Call POST /retrain | Returns 200 OK | As Expected | Pass |
| 2 | Monitor server logs | Retraining logs appear | As Expected | Pass |
| 3 | Check updated model timestamp | File update confirms retraining success | As Expected | Pass |

*Table 30: Test Case 14*

## 7.2 Equivalence partitioning

Equivalence partitioning was conducted on modules that process dynamic input values, particularly in feature extraction, packet labeling, and retraining threshold evaluation.

- **Kafka Producer – Packet Labeling:**
  The label is assigned using a probability parameter (label_probability), which must be in the range [0, 1].
  - *Valid Partition:* Values like 0.0, 0.5, 1.0
  - *Invalid Partition:* Values like -1, 1.5, "high" (string)
  - *Result:* Valid values allowed threat labeling; invalid values triggered error logging (validated via console).
- **Consumer Message Handling:**
  Incoming JSON messages are required to contain essential fields such as src_ip, dst_ip, label, etc.
  - *Valid:* Fully-formed JSON as produced by kafka_producer.py

- *Invalid:* Missing fields, malformed JSON, or null values
- *Result:* Valid messages processed by the ML models; invalid ones were logged and skipped (process_message handles JSONDecodeError.

## 7.3 Boundary value analysis

Boundary testing focused on critical parameters and control conditions in both scripts.

- **Label Probability Range (in Producer):**
  - *Boundary inputs tested:* 0.0, 0.0001, 0.9999, 1.0
  - *Result:* Values below 0 or above 1 were rejected (main() logs an error).
- **Packet Sizes:**
  - *Tested payload sizes:* 0 bytes, 1 byte, 1500 bytes (MTU limit)
  - *Result:* Packets of all sizes were processed; those with 0 payload were logged but not classified as threats.
- **Retraining Threshold (in Consumer):**
  - *Tested values:* 49, 50, 51
  - *Result:* Retraining started only when data count reached or exceeded 50 and cooldown passed (retrain_model() called via process_message()).
- **Prediction Outputs:**
  - *Model predictions:* Expected range 0–7
  - *Edge case handled:* Unexpected values were coerced to 0 (safe "No Threat" default)

## 7.4 Data flow testing

Using both Kafka logs and Flask endpoint responses, the following end-to-end data flow was validated:

1. **Packet Sniffing (sniff) → Feature Extraction**
2. **→ Kafka Producer → Kafka Topic (network_logs)**
3. **→ Kafka Consumer → Model Inference**
4. **→ Prediction Result → Prometheus Metrics**
5. **→ Grafana Dashboard via /metrics endpoint**

Evidence was collected by:

- Capturing packet output in Kafka CLI
- Observing consumer logs for model predictions
- Validating that /metrics showed real-time updates
- Verifying log data on Flask's /logs and /api/logs endpoints

Result: Every component in the pipeline preserved data integrity and continuity, with no loss or blockage.

## 7.5 Unit testing

Each critical function was tested in isolation using controlled data:

| Component | Function | Test Type | Result |
|---|---|---|---|
| **extract_features()** | Extracts IP, protocol, bytes | Manual test with crafted packets | Pass |
| **producer.produce()** | Kafka publishing | Unit test for message format | Pass |
| **preprocess() in consumer** | Feature vector generation | Tested with dummy data | Pass |
| **process_message()** | JSON decoding and ML prediction | Malformed vs. valid input | Pass |
| **manual_retrain()** | Retrain trigger | Threshold reached and cooldown tested | Pass |

*Table 31: Unit Testing*

## 7.6 Integration testing

Integration of individual modules was tested through full-chain validation:

- **Producer + Kafka Broker + Kafka Consumer:**
  - Verified messages reached the topic and were consumed without errors.
- **Consumer + Model Inference + Flask Dashboard:**
  - Verified that real-time predictions were logged and displayed on the dashboard.
- **Flask + Prometheus /metrics Endpoint:**

    - Verified Prometheus scraped correct metrics like total_threats, cpu_usage_percent.

Test Tools:

- Kafka CLI to inspect topic contents
- Flask routes tested with curl and browser
- Prometheus + Grafana used to visualize live data

## 7.7 Performance testing

Performance evaluation was carried out by sending synthetic packet data at high rates using Scapy.

- **Kafka Throughput:** Handled ~950 packets/sec
- **Consumer Processing Time:** Maintained <100ms latency under 1000 packet/sec load
- **Flask Response Time:** Under 200ms for JSON APIs
- **Resource Utilization:**
    - CPU: 65–80% (4-core VM)
    - RAM: <1.5 GB
- **Prometheus Metrics:** Reflected real-time counters and latency histograms

Conclusion: The system met real-time monitoring requirements under medium to high traffic loads

## 7.8 Regression Testing

After adaptive retraining:

- Older datasets were replayed through the system.
- The same test inputs were used before and after retraining.
- Results compared using model accuracy logs.

Observations:

- No degradation in detection capability.
- Improved accuracy in cases involving "Bot" and "DoS" classes.

System continued to return valid predictions without introducing false positives

## 7.9 Stress Testing

Stress testing simulated worst-case operational loads to assess stability:

| **Scenario** | **Method** | **Result** |
|---|---|---|
| **Burst traffic** | Sent 5000 packets/sec using Scapy | System handled queue overflow; no crash |
| **Manual retrain during load** | POST /retrain | Deferred retrain due to cooldown logic |
| **Kafka broker failure** | Stopped Kafka | Consumer paused and resumed gracefully |
| **CPU spike** | Forced load via stress tool | Flask remained responsive |

*Table 32: Stress Testing*

# Chapter 8
# Conclusion & Future Enhancements

# **Chapter 8:** Conclusion & Future Enhancements

## 8.1 Achievements and Improvements

The Heimdall system achieved all major technical and functional objectives set at the beginning of the project. It successfully captures live network packets, extracts relevant features, and classifies them in real time using pre-trained machine learning models. The Kafka-based pipeline ensures decoupled, high-performance streaming of data between modules, facilitating asynchronous processing and better fault isolation. The Prometheus integration enables real-time monitoring and metrics exposure, allowing system administrators to track performance and retraining activity with ease.

The inclusion of an adaptive retraining mechanism enables Heimdall to continuously evolve based on new data, improving its detection capabilities and keeping the system responsive to emerging threats. The Flask-based dashboard provides users with real-time threat logs, retraining controls, and visual metrics, enhancing transparency, interactivity, and usability. By integrating open-source technologies such as Scapy, Kafka, Prometheus, and Flask in a modular design, Heimdall has proven to be a practical and scalable prototype that balances efficiency and extensibility

## 8.2 Critical Review

Although Heimdall met its intended objectives, some areas for improvement remain. The current system does not support encrypted traffic inspection, which limits its effectiveness in secure environments. SSL/TLS-encrypted flows cannot be fully analyzed without decryption or indirect methods such as metadata inspection. Payload analysis is also not implemented, making the detection of certain sophisticated attacks more challenging.

The retraining process relies on user-provided labels, which may not be reliable or scalable in real-world deployments where human intervention is limited. Moreover, the dashboard lacks user authentication and access control, restricting its use in multi-user settings or public deployments. File-based storage for logs and models, while sufficient for prototyping and short-term analysis, is not ideal for long-term scalability, real-time querying, or audit compliance. Addressing these limitations would greatly enhance Heimdall’s practical applicability and industry readiness

.

The UI, though effective, lacks authentication and user roles, which would be necessary in a multi-user production environment. Moreover, model storage and log persistence are currently file-based and would benefit from database integration. Despite these limitations, Heimdall provides a strong foundation with flexible design and room for iterative improvement.

## 8.3 Lessons Learnt

During the course of this project, several technical and organizational lessons were learned. Working with real-time data streams and message brokers like Kafka reinforced the importance of asynchronous processing in scalable and fault-tolerant architectures. Implementing and managing machine learning models in a live system highlighted the challenge of concept drift and emphasized the necessity of designing reliable retraining mechanisms.

On the development side, the team gained valuable experience in modular software design, containerized deployment using Docker, and test-driven development. Incorporating monitoring tools like Prometheus and exposing system health metrics taught the importance of observability in production systems. Additionally, the team learned effective version control with Git, issue tracking, and collaborative problem-solving, which were critical to managing a complex, multi-module project under time constraints. These experiences will be highly applicable in future real-world software engineering roles.

## 8.4 Future Enhancements/Recommendations

To further improve Heimdall and prepare it for production environments, the following enhancements are recommended:

- **Authentication and Access Control**: Implement secure login mechanisms and role-based access to protect the dashboard, particularly in multi-user or enterprise settings.
- **Database Integration**: Replace file-based storage with scalable and queryable databases like PostgreSQL or MongoDB to enable efficient long-term log retention, search, and analytics.
- **Encrypted Traffic Handling**: Introduce the capability to analyze SSL/TLS traffic either through metadata-based techniques or with optional decryption methods in controlled environments.

- **Cloud-Based Deployment**: Extend Heimdall into a hosted SaaS platform, allowing remote access, centralized monitoring, and managed updates across distributed clients.
- **Model Fusion**: Experiment with ensemble learning or hybrid models to combine predictions from different classifiers and improve accuracy, confidence levels, and interpretability.
- **Real-Time Alerts**: Integrate notification mechanisms such as email, SMS, or webhook support to provide timely alerts to system administrators during threat detection events.
- **External Threat Feeds**: Incorporate regularly updated threat intelligence feeds (e.g., IP blacklists, malware signatures) to enhance model awareness and broaden the detection spectrum.

# Appendices

# **Appendix A:** Information / Promotional Material

[Paragraph Text 12 pt, Times New Roman, 1.5 Line Spacing, Justified]

[*Between 4 to 8 lines describe what is this appendix all about*]

## A.1. Broacher (if any)

## A.2. Flyer (if any)

## A.3. Standee (if any)

## A.4. Banner (if any)

# References:

[1] B. Dash, M. F. Ansari, P. Sharma, and A. Ali, “Threats and Opportunities with AI-Based Cyber Security Intrusion Detection: A Review,” Sep. 01, 2022. Accessed: May 11, 2025. [Online]. Available: https://papers.ssrn.com/abstract=4323258

[2] B. X. Wang, J. L. Chen, and C. L. Yu, “An AI-Powered Network Threat Detection System,” *IEEE Access*, vol. 10, pp. 54029–54037, 2022, doi: 10.1109/ACCESS.2022.3175886.

[3] “Apache Kafka.” Accessed: May 11, 2025. [Online]. Available: https://kafka.apache.org/

[4] “Welcome to Flask — Flask Documentation (3.1.x).” Accessed: May 11, 2025. [Online]. Available: https://flask.palletsprojects.com/en/stable/

[5] S. W. Tolbert, “Artificial Intelligence for Real Time Threat Detection and Monitoring”.

[6] F. S. De Lima Filho, F. A. F. Silveira, A. De Medeiros Brito Junior, G. Vargas-Solar, and L. F. Silveira, “Smart Detection: An Online Approach for DoS/DDoS Attack Detection Using Machine Learning,” *Security and Communication Networks*, vol. 2019, 2019, doi: 10.1155/2019/1574749.

[7] E. Murtaj, F. Marcantoni, M. Loreti, M. Quadrini, and H.-F. Witschel, “Real-Time Intrusion Detection via Machine Learning Approaches,” 2024.

[8] “Kafka-based Intrusion Detection System. | EBSCOhost.” Accessed: May 11, 2025. [Online]. Available: https://openurl.ebsco.com/EPDB%3Agcd%3A12%3A17086812/detailv2?sid=ebsco%3Aplink%3Ascholar&id=ebsco%3Agcd%3A181690644&crl=c&link_origin=scholar.google.com

[9] T. Sowmya and E. A. Mary Anita, “A comprehensive review of AI based intrusion detection system,” *Measurement: Sensors*, vol. 28, Aug. 2023, doi: 10.1016/J.MEASEN.2023.100827.

[10] A. Chiche and M. Meshesha, “Towards a Scalable and Adaptive Learning Approach for Network Intrusion Detection,” *Journal of Computer Networks and Communications*, vol. 2021, 2021, doi: 10.1155/2021/8845540.

[11] “Grafana: The open and composable observability platform | Grafana Labs.” Accessed: May 11, 2025. [Online]. Available: https://grafana.com/

[12] “Scapy.” Accessed: May 11, 2025. [Online]. Available: https://scapy.net/

[13] “Prometheus - Monitoring system & time series database.” Accessed: May 11, 2025. [Online]. Available: https://prometheus.io/

[14] E. V. Ananin, A. V. Nikishova, and I. S. Kozhevnikova, “Port scanning detection based on anomalies,” *11th International IEEE Scientific and Technical Conference &amp;amp;quot;Dynamics of Systems, Mechanisms and Machines&amp;amp;quot;, Dynamics 2017 - Proceedings*, vol. 2017-November, pp. 1–5, Dec. 2017, doi: 10.1109/DYNAMICS.2017.8239427.

[15] V. M. Akanksha Kavikondala, “Automated Retraining of Machine Learning Models”, doi: 10.35940/ijitee.L3322.1081219.