


# Evaluation of Large Language Models for Intrusion Detection Systems

Grado en Ingeniería Informática

Final Bachelor's Thesis

Author:
Ismael Hernández Alarcón
Tutor:
Jorge Bernal Bernabé

Facultad Informática Universidad Murcia

30th June, 2025

# Evaluation of Large Language Models for Intrusion Detection Systems

**Author**
Ismael Hernández Alarcón

**Tutor**
Jorge Bernal Bernabé
*Department of Communications and Information Engineering*

Facultad de Informática

Grado en Ingeniería Informática

DIIC

UNIVERSIDAD DE MURCIA

Murcia, 30th June, 2025

# Agradecimientos

Este trabajo no habría sido posible sin el apoyo y el estímulo de mi tutor, Doctor Jorge Bernal Bernabé, bajo cuya supervisión me ofreció este tema de investigación. También me gustaría agradecer a Pablo Fernández Saura, estudiante de doctorado en la Universidad de Murcia, por ayudarme durante todo el estudio, especialmente con las infraestructuras durante la fase de Evaluación. Gracias a ambos por toda la ayuda que ha dado lugar a un gran aprendizaje.

No podría concluir este trabajo sin expresar gratitud a mi familia. Poniendo énfasis en mis padres, Mari Carmen y Jose Francisco, hermanos, Jose Francisco y Gonzalo, y abuelos, Paco y Carmen y Pepe y Mercedes, por su amor incondicional, compromiso y sabiduría. Gracias a ellos he tenido la estabilidad y los valores necesarios para alcanzar esta meta.

# Abstract

The cyber attacks have grown in frequency and sophistication in recent years, intrusion detection systems (IDS) have become a fundamental component in safeguarding enterprise networks. The motivation for this study arises from the pressing need to enhance the capabilities of IDS, particularly in IoT or large enterprise settings, where each device type has his own log, making very difficult to extract features from the logs.

Due to the significant surge in popularity and usage of LLMs, their integration into diverse fields has become a prominent research direction. In particular, their capacity to understand and process natural language at scale has opened new possibilities in cybersecurity, especially for log analysis and intrusion detection. Likewise, Retrieval-Augmented Generation, proposed in 2020, has emerged as a promising method to enhance model reasoning by providing relevant context. This growing relevance and potential provided the justification for selecting this topic as the focus of my thesis research.

All of that arises my motivation to evaluate different IDS models with the same dataset. Specifically, it investigates the classification capabilities of a standalone Large Language Model (LLM), a K-Nearest Neighbors (KNN) classifier and a hybrid approach combining LLM with Retrieval-Augmented Generation (RAG). The objective is to improve the detection of malicious activity in real log data by leveraging semantic similarity and natural language understanding.

A comprehensive architecture was designed and implemented, encompassing dataset preprocessing, the deployment of an LLM through a dedicated API endpoint, and the use of a Dockerized OpenSearch instance for efficient log storage and semantic retrieval. The system evaluates the incoming logs by retrieving similar past entries using KNN, enriching the LLM's input with the most similar logs. Additionally, the other two scenarios were defined to compare performance, with a strong focus on classification accuracy, precision, recall and inference time across all models.

# Resumen

En los últimos años, los ciberataques han experimentado un notable incremento tanto en frecuencia como en sofisticación, lo que ha convertido a los sistemas de detección de intrusiones (IDS, por sus siglas en inglés) en un componente esencial para la protección de redes empresariales. Esta investigación nace de la necesidad urgente de mejorar las capacidades de los IDS actuales, especialmente en entornos complejos como el Internet de las Cosas o grandes infraestructuras empresariales. En estos contextos, cada dispositivo genera su propio formato de log, lo que complica significativamente la extracción de patrones comunes y dificulta la detección de comportamientos anómalos de manera eficiente y precisa.

Paralelamente, el auge en la popularidad y uso de los grandes modelos del lenguaje (LLMs) ha abierto nuevas líneas de investigación en múltiples disciplinas. Su capacidad para comprender y procesar lenguaje natural a gran escala los convierte en herramientas particularmente útiles para la detección de intrusiones a través del análisis de logs. Además, desde su introducción en 2020, la técnica conocida como Retrieval-Augmented Generation (RAG) ha ganado atención por su capacidad de mejorar el razonamiento de los modelos al incorporar contexto relevante recuperado de fuentes externas. Esta sinergia entre modelos LLM y recuperación semántica ha demostrado ser especialmente prometedora para enfrentar los retos que plantea la clasificación de logs en entornos reales.

Dada esta evolución tecnológica y su impacto directo en el ámbito de la ciberseguridad, se consideró pertinente centrar este Trabajo Fin de Grado en la aplicación conjunta de LLMs y RAG a la detección inteligente de intrusiones. Esta elección se fundamenta en la relevancia actual del tema, su potencial práctico y las oportunidades de innovación que ofrece en el desarrollo de soluciones adaptativas, explicables y escalables.

El análisis de logs en ciberseguridad ha sido tradicionalmente abordado mediante técnicas clásicas como los sistemas basados en firmas (SBS), que detectan amenazas conocidas mediante patrones predefinidos y los sistemas basados en anomalías (ABS), que identifican comportamientos inusuales frente a un perfil base. Mientras los SBS ofrecen alta precisión frente a ataques conocidos, su dependencia de bases de datos actualizadas limita su efectividad ante nuevas amenazas. Por otro lado, los ABS son más adaptables, pero generan un mayor número de falsos positivos, especialmente en

entornos IoT con datos variados y dinámicos. Debido a esto, los métodos clásicos resultan insuficientes ante ataques modernos que modifican patrones y evaden detecciones. En respuesta, surgieron enfoques basados en aprendizaje automático, los cuales permiten identificar patrones anómalos a partir de datos históricos. Modelos como SVM o Random Forest se han utilizado ampliamente [1], pero su rendimiento depende en gran medida del diseño de extracción de características y preprocesado, lo que representa un desafío ante la heterogeneidad de los logs en grandes sistemas reales.

En los últimos años, los modelos de aprendizaje profundo [2], especialmente las redes neuronales recurrentes (RNN) y convolucionales (CNN), han mejorado la capacidad de modelar secuencias de logs. No obstante, estos enfoques requieren grandes volúmenes de datos etiquetados y una elevada capacidad computacional, lo que dificulta su aplicación en entornos reales. Por otro lado, los avances en procesamiento de lenguaje natural han llevado al uso de modelos basados en transformers, como BERT, para el análisis semántico de eventos, permitiendo representar los logs como texto estructurado y capturar relaciones complejas entre eventos.

En este contexto, los modelos de lenguaje de gran escala (LLMs) han emergido como una alternativa prometedora [3] . Estos modelos, entrenados con grandes cantidades de texto, son capaces de generalizar y razonar sobre eventos complejos sin necesidad de ajustes finos específicos. Su capacidad para realizar tareas de clasificación y generación de texto los convierte en candidatos ideales para la clasificación de logs, incluso con entradas ambiguas. A pesar de ello, su efectividad puede verse limitada cuando carecen de contexto específico del dominio o histórico del sistema analizado, lo cual ha motivado la integración de técnicas complementarias como la recuperación semántica.

La técnica conocida como Retrieval-Augmented Generation (RAG), introducida en 2020 [4], combina el poder generativo de los LLMs con la recuperación de información contextual relevante, permitiendo al modelo fundamentar sus respuestas con ejemplos previos. Este enfoque ha sido aplicado en otros dominios y comienza a explorarse en ciberseguridad [5]. La combinación de RAG con LLMs ofrece una solución robusta para la clasificación explicable de logs, al integrar el conocimiento adquirido previamente mediante el RAG con la capacidad de adaptación del modelo. Esta línea representa el estado del arte actual y motiva la propuesta implementada en este estudio.

La metodología de esta investigación se centra en evaluar el impacto de la técnica Retrieval-Augmented Generation en sistemas de detección de intrusos basados en modelos de lenguaje grandes. Para lograrlo, se establecieron fases clave que incluyen el análisis de trabajos previos, selección del LLM, Transformer y motor de búsqueda semántica, elección del algoritmo de búsqueda y métrica de distancia y finalmente, el diseño de la arquitectura y su evaluación. Cada fase fue diseñada para abordar limitaciones prácticas en entornos empresariales reales.

Se analizó cómo los LLM pueden mejorar la detección de intrusiones, especialmente mediante RAG, que permite enriquecer el contexto de los registros actuales recuperando ejemplos similares del pasado. Este enfoque supera la dependencia de características manuales y modelos entrenados previamente, ofreciendo respuestas más flexibles y contextualizadas. Además, se comparó con técnicas anteriores donde se seleccionaban manualmente ejemplos relevantes, concluyendo que el uso de búsquedas semánticas vectoriales automatizadas ofrece una solución más escalable y eficiente para entornos de seguridad en tiempo real.

Para implementar la búsqueda semántica, se compararon tres motores: OpenSearch, Vespa y Weaviate. De todos estos, se eligió OpenSearch por su madurez, ser open-source y flexibilidad en configurar búsquedas vectoriales. OpenSearch ofrece herramientas de observabilidad como dashboards, facilitando la resolución de problemas y una API fácilmente usable desde Python. A parte de eso, tiene soporte para búsquedas híbridas, búsquedas vectoriales filtrando ciertos campos, y compatibilidad con el plugin k-NN. Esto lo convierte en una solución robusta para manejar millones de vectores en entornos empresariales complejos, permitiendo consultas combinadas y búsqueda basada en similitud semántica en registros de texto.

En cuanto a la técnica de indexación, se eligió HNSW por su eficiencia sin necesidad de entrenamiento previo, permitiendo una actualización dinámica del índice. Aunque Lucene es el motor por defecto de Opensearch, Facebook AI Similarity Search (FAISS) fue preferido por su soporte a una mayor compresión de los vectores, ahorrando memoria. Asimismo, se determinó que la métrica de distancia más adecuada para recuperar registros semánticamente similares es la similitud de coseno, por su robustez en espacios de alta dimensión, su extenso uso en la búsqueda por similitud de texto y su compatibilidad con los embeddings normalizados generados por transformadores como MiniLM.

La selección del dataset también fue clave. Se eligió el AIT Log Data Set V2.0, concretamente el subconjunto “Russell Mitchell”, que simula un entorno empresarial real con múltiples servicios y ataques como escaneos, subida de webshells, fuerza bruta, ejecución remota y exfiltración de datos vía DNS. Este conjunto contiene más de 11 millones de logs con etiquetas detalladas y simula cuatro días de actividad, siendo ideal para probar modelos semánticos y contextualizados. Su estructura rica en servicios y diversidad lo hace especialmente útil para evaluar modelos basados en LLMs y técnicas de búsqueda por similitud.

También, se seleccionó el modelo de embedding all-MiniLM-L6-v2 por su eficiencia, eficacia, precisión semántica y bajo consumo de recursos. Varios estudios [6] respaldan que pequeños Transformers obtienen mejores resultados que Transformers profundos

para clasificar logs. Como modelo de lenguaje principal se escogió LLaMA 3.1-8B Instruct, por su rendimiento sólido, velocidad de inferencia y tener una ventana de contexto de 8k tokens, suficiente para procesar entradas con RAG. Esta combinación permite construir un sistema de detección de intrusiones interpretable, escalable y apto para aplicaciones en sistemas reales.

La arquitectura general del sistema propuesto combina tecnologías de procesamiento semántico con modelos generativos, estructurado alrededor de una arquitectura modular basada en logs. Los registros generados por los servicios empresariales se recolectan y procesan a través de un agente central RAG, el cual los embebe usando MiniLM y los indexa en OpenSearch. Al recibir un nuevo log, el agente recupera eventos semánticamente similares y los envía junto con el evento actual a un LLM que analiza si el comportamiento es anómalo o no. Si es sospechoso, se genera una alerta explicativa para el administrador del sistema.Para evaluar dicho sistema, se diseñaron tres escenarios experimentales. El primero, basado en KNN, clasifica logs utilizando vecinos semánticamente similares en OpenSearch, sin razonamiento adicional. El segundo, basado en LLM, emplea un modelo LLaMA para interpretar individualmente los logs. El tercero integra ambos métodos en un pipeline de RAG, proporcionando al LLM un contexto enriquecido de logs pasados.

Para gestionar eficientemente los datos, se desarrolló un script de procesamiento que genera un CSV a partir del dataset AIT Log V2.0. Este archivo incluye la fuente del log, el mensaje y su etiqueta, alineadas con anotaciones JSON. En el preprocesado, no se aplicaron técnicas de limpieza para conservar la semántica original de los logs, pero dado el desequilibrio en la distribución de clases, se redujo el número de logs normales para concentrar el entrenamiento y evaluación en patrones de ataque. Finalmente, los datos se dividieron de forma estratificada para asegurar una evaluación representativa y equilibrada, con más de 6 millones de entradas en el conjunto de entrenamiento y alrededor de 50.000 entradas en el conjunto de evaluación. En este último conjunto, se hizo una reducción mayor debido al tiempo necesario para evaluar todos los modelos, pero se mantuvieron todos los logs que tenían etiqueta de ataque y al menos un log de cada fuente.

Asimismo, OpenSearch 2.19 se utiliza como base de datos vectorial, configurado con compresión escalar de 32x y almacenamiento en disco para reducir el uso de memoria. Para reducir los tiempos de las búsquedas vectoriales, la indexación se realiza con HNSW, usando el motor de búsqueda FAISS, Facebook AI Similarity Search. Finalmente, los embeddings son generados por MiniLM y se emplea similitud de coseno como métrica de distancia. La arquitectura permite consultas vectoriales en tiempo real, compatibles con filtros por metadatos como la fuente del log. Además, se despliega un pipeline de OpenSearch que genera embeddings en tiempo real mediante llamadas REST, eliminando la necesidad de procesamiento externo y facilitando una integración

eficiente y escalable del sistema de detección. Tras esto, se presentan los tres escenarios a evaluar.
Primero, en el escenario basado en KNN se emplea Opensearch con la configuración antes mencionada como base de datos y motor de búsqueda vectorial. La fase de entrenamiento consiste en leer un archivo CSV con logs preprocesados, extraer el mensaje, su origen y la etiqueta, y procesarlos por lotes para su inserción en OpenSearch, utilizando el pipeline que genera automáticamente los vectores durante la ingesta, lo que permite construir un índice semántico escalable y reutilizable para otros escenarios como el RAG. Durante la evaluación, cada log se embebe de forma similar y se compara contra los vectores existentes utilizando la similitud del coseno. La búsqueda k-NN devuelve los logs más parecidos junto con sus etiquetas y puntuaciones, y se aplica una política de votación ponderada para asignar una etiqueta al nuevo log, eligiendo la etiqueta con mayor puntuación ponderada. Esta política considera la cercanía semántica, es decir, la puntuación, en lugar de un simple voto mayoritario, mejorando la precisión de clasificación.

El segundo escenario, basado en LLM, opera mediante un servidor que expone una API REST para clasificar logs utilizando Llama-3.1-8B-Instruct. Además, utilizamos un cliente Python para enviar peticiones con prompts estructurados que incluyen el mensaje del sistema y usuario, en el mensaje del sistema se detalla la tarea, dando órdenes concretas de cómo hacerlo y la estructura de la salida del LLM y en el del usuario se manda el log y su fuente. El modelo responde con una etiqueta y una explicación, ambas en formato JSON. También se controla el comportamiento del modelo mediante parámetros como temperatura, longitud máxima de la respuesta y estructura de salida. Esta configuración asegura que las respuestas sean coherentes y ajustadas a los requisitos de la aplicación. Las respuestas erróneas o ambiguas se registran para análisis posterior, mejorando la trazabilidad y depuración del sistema.

En el último escenario, el sistema combina LLM con RAG. A cada nuevo log se le asocian K registros previos recuperados mediante KNN, lo cual forma el contexto que el LLM utiliza para tomar decisiones. El mismo índice de Opensearch utilizado en el primer escenario es usado para recuperar los logs más similares con KNN, por lo que no requiere de un nuevo entrenamiento ni nuevas inserciones a la base de datos. El prompt es enriquecido con este historial y diseñado para que el modelo justifique si su decisión se basa en los logs contextuales o no. Esta arquitectura permite decisiones más informadas y justificadas, especialmente en patrones de ataque menos evidentes. Además, el sistema actualiza el índice con cada nueva clasificación, añadiendo el log etiquetado a la base de datos, mejorando su capacidad de recuperación y adaptación continua frente a nuevas amenazas en el tiempo.
La evaluación de los diferentes sistemas se basa en métricas estándar como accuracy, precisión, recall y F1-score, utilizando promedios ponderados para corregir el desequilibrio de clases. Se emplea también una matriz de confusión y un informe de clasi-

ficación por etiqueta. Las clases con pocos ejemplos fueron excluidas de las gráficas, pero sí consideradas en las métricas globales, asegurando así un análisis completo y representativo de todos los escenarios evaluados.

En el escenario basado en KNN, se determinó que k=3 es el valor óptimo al ofrecer una alta exactitud (96,2%) y tiempos de consulta bajos. Este modelo muestra un rendimiento robusto, con puntuaciones F1 por encima de 0,90 en clases frecuentes como `normal_log` y `service_scan`. Sin embargo, sufre con clases poco representadas, como `network_scan`, donde el recall fue solo 0,28, reflejando una tendencia conservadora y sesgo hacia las clases mayoritarias.

En el escenario LLM sin contexto, se obtuvo una accuracy del 88%, influenciada por el dominio de la clase `normal_log`. Aunque las métricas ponderadas son elevadas, el promedio macro revela debilidades importantes en clases minoritarias. El modelo falla por completo en detectar algunas etiquetas, como `network_scan`, y muestra escasa generalización. Sin ejemplos históricos, el modelo tiende a etiquetar como `normal_log`, reduciendo su utilidad práctica en detección de intrusiones.

La integración de RAG con LLM mejora sustancialmente los resultados: la accuracy alcanza el 98,1%, y todas las métricas por encima del 98%, superando así ampliamente a los otros modelos. Incluso clases complicadas como `network_scan` muestran mejoras significativas en precisión. Este enfoque permite que el LLM razone mejor con contexto, logrando altos F1-scores en la mayoría de las etiquetas. A pesar del aumento en su tiempo de consulta, este coste se justifica por la alta mejora en precisión y recall.

En resumen, los resultados de la evaluación demuestran que el modelo híbrido LLM + RAG supera consistentemente a los enfoques LLM y KNN por separado. Aunque los otros modelos presentan tiempos de inferencia rápidos, su precisión y capacidad de detección son limitadas, especialmente ante ataques raros u ofuscados. La búsqueda por similitud usando KNN ofrece buenos rendimientos, pero la combinación con un LLM proporciona mayor equilibrio y exactitud, destacando en etiquetas críticas como `dnsteal`. Además, el modelo LLM + RAG es el más eficaz detectando ataques extremadamente raros, lo que confirma su aplicabilidad real y su gran potencial en el ámbito de la ciberseguridad.

Para investigación futura, se propone mejorar las estrategias de recuperación en arquitecturas RAG combinando representaciones densas y dispersas, se podría utilizar, por ejemplo, técnicas de aprendizaje de clasificación para mejorar la calidad de los logs recuperados. También se destaca la necesidad de abordar el desbalance de clases en los logs, optimizar el procesamiento en tiempo real y aumentar la explicabilidad de las decisiones del LLM. Estas mejoras pueden incrementar la precisión, eficiencia y confianza en entornos reales.

# Contents

# List of Figures

# List of Tables

# Listings

# 1 Introduction

## 1.1 Context

As cyber threats become increasingly complex and frequent, especially in the context of Internet of Things (IoT) ecosystems, the need for advanced, adaptive, and scalable security mechanisms has become more urgent than ever. Traditional intrusion detection systems (IDS), based on static rules or predefined attack signatures, often fall short when confronted with novel or subtle attack patterns, particularly in distributed and heterogeneous environments.

At the same time, the field of Natural Language Processing (NLP) has undergone a major shift with the emergence of Large Language Models (LLMs). These models have demonstrated remarkable performance in tasks requiring semantic understanding and context awareness—capabilities that are increasingly relevant for analyzing logs and system activity data in security contexts.

In parallel, distributed search and analytics suite like OpenSearch now offers native support for high-dimensional vector indexing and different methods to search, such as K-Nearest Neighbors. This enables efficient retrieval of semantically similar logs when combined with LLM-based embeddings, creating a promising new paradigm for intrusion detection. By transforming raw log messages into dense vector representations and leveraging OpenSearch's retrieval infrastructure, it becomes possible to implement a real-time IDS framework capable of detecting both known and previously unseen threats based on log similarity.

In the field of intrusion detection, this paradigm takes on special importance as it provides interesting benefits such as a natural language response to events, a higher accuracy in unseen attacks, or even a reduction in attack detection time, among other aspects.

## 1.2 Motivation

The growing potential of machine learning, particularly LLMs, has been useful in a wide range of areas, among which the chatbots takes a special importance. Given

that, we are going to use this ability to semantically interpret and classify logs in high-volume log environments such as those found in IoT networks.

While much research has been devoted to improving intrusion detection through traditional machine learning classifiers or statistical techniques, these methods often require extensive feature engineering and struggle with generalization. LLMs, on the other hand, offer a way to encode the full textual context of log entries into meaningful embeddings that can be used for flexible, scalable retrieval and classification.

Our main motivation and main objective is to evaluate a Large Language model (LLM) with Retrieval Augmented Generation(RAG) as an IDS in order to see which of these benefits can be achieved, what problems arise and how we can solve them.

By studying their effectiveness and deployment feasibility, this work seeks to contribute a practical, modern approach to intrusion detection that leverages the strengths of both large-scale retrieval systems and advanced language models.

## 1.3 Objectives

To achieve the main objective of evaluating the effectiveness and feasibility of the three intrusion detection models within the OpenSearch framework, the following specific objectives have been defined:

- **Objective 1**.Analyze the state of the art and the related work, taking into account the role of Large Language Models (LLMs) in semantic understanding and anomaly detection, especially in the context of log data.
- **Objective 2**. Analysis and selection of the semantic search engine, dataset and LLM needed for the system.
- **Objective 3**. Design the architecture solution, focusing on how context enhancement can improve LLMs.
- **Objective 4**. Implement the proposed solution, including a framework that supports vector search and data ingestion pipeline.
- **Objective 5**. Evaluate the implemented solution. Three different approaches are proposed and compared: A baseline intrusion detection model using Machine Learning methods, another one based on LLM and finally the LLM with RAG system.

- **Objective 6**. Identify the main challenges encountered during the implementation and evaluation process and propose future research directions based on the findings.

## 1.4 Outline

This document is composed firstly of an abstract in english, an extended abstract in spanish, and 6 chapters. Below, each of these chapters are briefly described:

- **Chapter 1** Introduction. Presents the context, motivation, objectives and structure of the work.
- **Chapter 2** State of the Art & Background. Introduction to Intrusion Detection Systems, discusses the capabilities of LLMs with RAG in NLP and how to use it for IDS, so, objective 1 is defined.
- **Chapter 3** Analysis & Methodology. Several points such as semantic search framework, intrusion detection datasets or search algorithms are analyzed, implementing objective 2. In addition, the methodology for this work is presented.
- **Chapter 4** Design & Implementation. Objectives 3 and 4 are fulfilled, detailing the ingestion pipeline, the semantic search configuration, the models implemented and the evaluation environment.
- **Chapter 5** Evaluation results. For each considered scenario, a performance quantitative analysis is presented, therefore, objective 5 is satisfied.
- **Chapter 6** Conclusions, challenges and future trends. This last chapter provides an overview of the results, description of the challenges encountered and future trends to be considered for further works, fulfilling objective 6.

# 2 State of the Art & Background

## 2.1 Introduction

This chapter provides a comprehensive review of the foundational concepts and advances relevant to this thesis. It begins by exploring traditional and machine learning-based intrusion detection systems, focusing on their types, mechanisms, and limitations. Next, related work about LLM based IDS is summarized. Finally, introduces the proposed approach of leveraging LLMs with Retrieval-Augmented Generation to enhance intrusion detection.

## 2.2 Intrusion Detection Systems

An intrusion detection system is a network security tool that monitors network traffic and devices for known malicious activity, suspicious activity, or security policy violations. They are the most efficient way of defending against network-based attacks aimed at computer systems and are widely used in almost all large-scale IT infrastructure.

### 2.2.1 Traditional IDS

Historically, there were two main types of intrusion detection systems:

- Signature-based Intrusion Detection System (SBS).
- Anomaly-based Intrusion Detection System (ABS).

SBS systems rely on a database of predefined attack signatures or characteristics of known threats, such as specific byte sequences in malware or known exploit code.These systems compare network traffic and system activity against this database to identify matches, offering high accuracy for detecting known attacks with low false positive rates. Signature-based systems normally use pattern matching algorithm such as Aho-Corasick algorithm used in tools like Snort[7]. Also, these algorithms are efficient for multi-pattern string matching, so, the systems detect the attacks really quickly.

However, signature-based IDS have significant limitations. They only detect attacks whose signatures are previously stored in the database, leading to a high inefficiency

against novel attacks threats. Moreover, as we see in figure 2.1, the database requires constant updates from a system expert because the system needs a signature for every attack. In IoT environments, where devices generate diverse and high-volume data, maintaining an up-to-date signature repository becomes impractical.

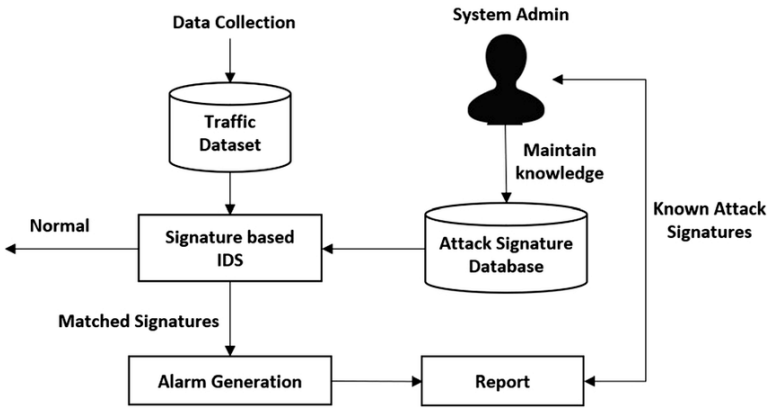


**Figure 2.1:** Signature Based IDS Architecture.

Anomaly-based IDS address the shortcomings of signature-based systems by establishing a baseline of normal behavior for a network or system. Deviations from this baseline, such as unusual traffic patterns or unexpected system calls, are flagged as potential intrusions. In the architecture figure 2.2, we see that the system administrator does not need to maintain any database as in the SBS, but instead a normal profile dataset is needed.

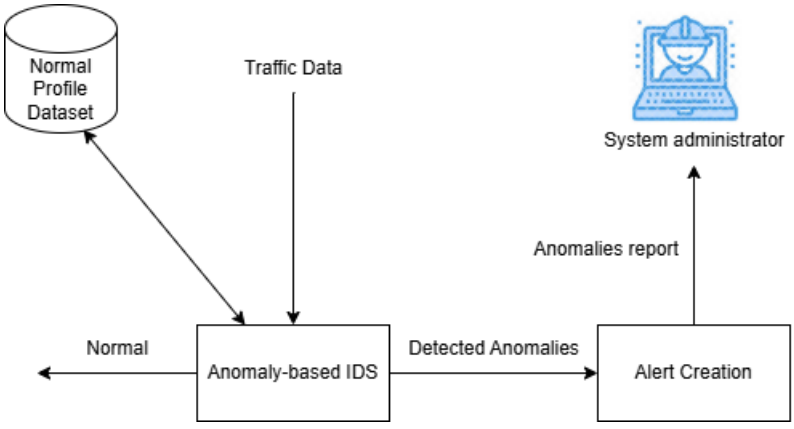


**Figure 2.2:** Anomaly Based IDS Architecture.

While anomaly-based IDS excel at identifying novel or zero-day attacks, they suf-

fer from high false positive rates, as legitimate but rare behaviors may be mistaken for threats. In IoT ecosystems, the heterogeneity of devices and dynamic network conditions exacerbate this issue, making it challenging to define a consistent baseline. Additionally, training anomaly detection models requires substantial data, which is often scarce or incomplete in real-world settings.

## 2.3 Related Work

### 2.3.1 ML-Based IDS

In this section, the past studies of Machine Learning based IDS are presented. Diro et al. [8] proposed a distributed deep learning-based IoT attack detection system. The results of their work gave 96% accuracy.

Bostani and Sheikhan [9] introduced an altered K-means strategy for shrinking the training dataset and balancing the data used to train ELMs and SVMs. The experimental findings of the proposed model gave 96.02% percent accuracy and a 5.92 percent false alarm rate. The clustering with self-Organized Ant Colony Networks (CSOACN) was used to categorize network traffic as benign or unsual traffic.

In [10] Kayode and Idris evaluated six different classifiers using the UNSW-NB-15 dataset. First, the data was preprocessed and normalized using Min-Max technique.After that, dimensionality reduction was performed with PCA. The XG-Boost, Cat Boost, KNN and SVM classifiers achieved a 99.9% accuracy and a precision of 1, while the Naive Bayes accuracy was 97.14%. The main drawback of this detection system is the feature extraction and the preprocessing of the data before being fed to the classifier. This could become too complex and resource intensive in real-time detection systems.

### 2.3.2 Limitations in Real Environments

IDS face unique challenges in real enterprise ecosystems. The sheer volume of data generated by interconnected devices demands scalable processing, while resource constraints restrict the deployment of computationally intensive models. Furthermore, the heterogeneity of new IoT devices, each producing logs in different formats, complicates feature extraction and unified analysis. These limitations highlight the need for an adaptive and intelligent IDS capable of handling diverse data and detecting novel threats without a complex manual configuration.

### 2.3.3 LLM-Based IDS

Unlike network-based systems, Host-based IDS work with text sources like system call records, log files, and application or memory traces. They use Natural Language Pro-

cessing methods to examine the sequence and semantic interactions of those calls and logs, spotting suspicious patterns as they happen. By treating these data, HIDS can detect intrusions quickly and accurately. Several NLP-based solutions have been developed before the rise of today's large language models[11].

LLMs are increasingly being adapted for network traffic detectors, but typically with architectural changes. Instead of the usual sequence-to-sequence head, which predicts the next token, these models uses a classification layer whose size matches the number of target classes, obtaining a deterministic output to evaluate it. Lira et al. [12] evaluated GPT-3.5, GPT-4, and ADA on the CICIDS2017 and Urban IoT datasets for DDoS detection. They showed that fine-tuning or even just a few labeled examples, these LLMs can achieve high accuracy in spotting DDoS attacks, highlighting their flexibility and velocity in adapting to security tasks.

Manocchio et al. in [6] evaluated four Transformers models, including BERT and GPT 2.0, with six different classification heads using common datasets, such as UNSW_NB15 and NSL-KDD. The findings are that the shallow transformers, with many fewer parameters, performs better than the deep Transformers with a F1-score of 97% versus the 96% of the GPT decoder.

Among these, Michael et al. [13] is the only study which employed LLMs, which are pre-trained on natural language and possess a sequence-to-sequence head producing sequences of natural text tokens, to detect DDoS attacks by fine-tuning OpenAI's Ada model. They evaluated 3 approaches: a LLM without context, a LLM with the labeled top k training data samples in the context and a LLM fine-tuned with the training data. The LLm with top K surpass mostly all the other configurations, including the MLP model used as the baseline. Only the fine-tuned LLM achived higher accuracy in certain cases. Although this IDS is using IoT datasets, It helped me a lot to clarify which models to evaluate and how I can configure them.

Other field of study is **explainable IDS**, usually employing existing explainable AI methods.For instance, Mallampati et al. in [14] applies the SHapley Additive exPlanations (SHAP) model across different network scenarios, quantifying how much each feature contributes to a given prediction. By assigning each feature a precise impact score on the final classification, this technique makes it possible to design real-time threat-response algorithms. Large language models can offer human operators intuitive, text-based justifications that go beyond merely highlighting statistically unusual features flagged by the NIDS. This will allow to facilitate threat response and a better understanding of their NIDS functionality or potential malfunctioning. In this study, we further investigate the capability of LLMs to identify malicious activity while explaining their reasoning. Houssel et al.[5] fine-tuned the GPT-4 and LLama3 models with ORPO and KTO methods to try these, achieving a precision and recall of nearly

50%, concluding that using the LLMs as IDS has notable limitations now.

The newest state of the art LLMs based IDS are the IDS agent. Introduced by Li et al.[15], proposed a LLM agent that has a toolbox. The agent iterates through 3 steps: reasoning, action generation and observation update after the toolbox executes the action. Although the classification is done in the toolbox by ML-based IDS, the final output is infered by the LLM with the classification scores and the knowledge retrieved from other tools. Moreover, the agent was compared with a normal LLM based IDS and a Random Forest based IDS, it outperforms all the other approaches.

## 2.4 Proposal

The previous analysis of the state-of-the-art about LLM-based Intrusion Detection Systems demonstrates that this area of research is growing exponentially in recent years. Most of the described works are focused on IoT environments, particularly relying on packet-level inspection and feature-based learning. However, packet tracking can be highly volatile and requires real-time processing capabilities, which is often not feasible or scalable for many enterprises. Moreover, most packet-level systems lack semantic interpretability, making it harder for human analysts to understand and act on their predictions.

This thesis proposes an alternative architecture by leveraging Large Language Models with Retrieval-Augmented Generation to analyze logs rather than raw network packets. Logs inherently capture a wide range of system-level and user-level activities, such as authentication attempts, system calls, scheduled tasks, and firewall alerts. These logs provide rich textual data, which makes them highly suitable for NLP-based modeling, particularly with LLMs.

# 3 Analysis & Methodology

## 3.1 Methodology

As described in Section 1.3, the main objetive of this research is to perform a quantitative analysis on the impact of Retrieval-Augmented Generation on the performance of a LLM-based Intrusion Detection System. Furthermore, how the variation of certain parameters, such as the prompt and the RAG, affect the overall accuracy. Nevertheless, before adressing this task, we define the next methodology phases in order to meet all the proposed objectives:

1. **Analysis of previous approaches in attack detection**. Review recent research in the field of intrusion detection using LLMs and identify their weak points.

2. **Analysis and selection of the Semantic Search Engine**. Investigate available semantic search engines. Compare their vector indexing capabilities, scalability, and select the most suitable option for log-based anomaly detection.

3. **Analysis and selection of Search algorithm**. Research about the most interesting search algorithms for our use case and analyze them.

4. **Analysis and selection of distance metric algorithm**. Explore various distance metrics and justify the selection of the metric best aligned with the requirements of our log retrieval.

5. **Analysis and selection of the Sentence Transformer**. Investigate the most suitable Transformer for our objetive.

6. **Analysis of the Log Dataset**. Study the characteristics of the chosen dataset, including the diversity of logs, labeling strategy and volume.

7. **Design & Implementation**. Once the implementation engine and algorithms have been selected, adapt it to our needs and propose an architecture design.

8. **Evaluations**. Test and compare how different scenarios and configurations affect the overall detection accuracy.

## 3.2 LLM applied to Intrusion Detection

While recent studies have shown that LLMs can achieve high accuracy in intrusion detection when fine-tuned or guided with labeled examples, they still face key limitations, particularly when deployed in dynamic or noisy environments. Many works, adapt transformer architectures for classification tasks, which require substantial pretraining or fine-tuning on specialized datasets. Moreover, these models often lack contextual awareness when processing isolated log entries or traffic records, and their predictions are inherently limited to the scope of their training.

This is where **Retrieval-Augmented Generation** becomes an impactful complement. Instead of relying solely on the model's internal knowledge or requiring exhaustive fine-tuning, RAG allows the system to *dynamically retrieve relevant examples* from a log database based on semantic similarity. For example, when a suspicious login or command execution log is processed, the RAG pipeline can retrieve historical entries that exhibit similar behavior, including previously labeled intrusions. These retrieved samples serve as *real-time, contextual memory* that enrich the model's input and enable more accurate, informed decisions.

In comparison to the work by Michael et al. [13], where context windows were manually populated with top-K examples, a RAG system automates this process using dense embeddings and vector similarity search scaling the concept for real-time deployments, updating the RAG Database and filtering the Knn search. Furthermore, this architecture removes the dependency on handcrafted features or preprocessing pipelines described in classical ML models, enabling fully text-native intrusion analysis.

Regarding **explainability**, models such as those using SHAP to visualize the contribution of features in structured data. In contrast, an LLM + RAG approach can provide rich, natural language justifications for its classifications, helping human analysts quickly understand why a log was flagged and the type of attack, particularly useful in Security Operations Center environments. This represents a key evolution in explainable IDS: not just scoring feature weights, but reasoning through similar cases retrieved from past experience.

Even the promising **LLM agents** whose normally integrates RAG modules. Their architecture uses an external toolbox for classification but leaves contextual grounding to the agent's internal reasoning. Adding semantic retrieval enables the agent to make decisions informed by the most relevant, real-world examples stored in the log corpus. Furthermore, the agents tend to be very slow at predicting in real-time scenarios that is why using only a LLM with RAG could be more suitable for those scenarios.

In summary, RAG enhances the adaptability, scalability, and interpretability of

LLM-based IDS by:

- Enabling retrieval of semantically relevant historical logs for better context.
- Reducing reliance on extensive fine-tuning or manually labeled datasets.
- Allowing flexible, few-shot reasoning even in novel or ambiguous situations.
- Generating human readable and context aware justifications for security alerts.

These improvements position LLM + RAG architectures as a promising next step in the evolution of intelligent and transparent intrusion detection systems.

## 3.3 Analysis of Semantic Search Engine: Opensearch

To implement semantic search within our RAG-based IDS pipeline, it is crucial to select an efficient and scalable vector database capable of indexing and retrieving dense embeddings generated from logs. Three of the most prominent solutions in this domain are **OpenSearch**, **Vespa**, and **Weaviate**. Each has strengths and trade-offs in terms of open-source support, indexing capabilities, and integration with LLM workflows.

| Feature | OpenSearch | Vespa | Weaviate |
|---|---|---|---|
| License | Apache 2.0 | Apache 2.0 | Business Source License |
| Semantic Search Support | Yes (k-NN plugin) | Yes (ANN support) | Yes (native hybrid search) |
| Vector Indexing Capabilities | FAISS, HNSW, Lucene | HNSW, brute-force | HNSW, PQ, IVF |
| Custom Pipelines / LLM Integration | Manual setup via APIs or plugins | Advanced support via functions | Direct API |
| Best Use Case | Elasticsearch-compatible systems | High-scale, low-latency | Fast prototyping with built-in ML |

**Table 3.1:** Comparison of Semantic Search Engines

**OpenSearch** is a powerful fork of Elasticsearch maintained by the Linux Foundation and Amazon. It supports vector search through its `k-NN` plugin, compatible with ANN algorithms like FAISS and HNSW. Moreover, AI search methods such as multimodal, hybrid or neural sparse search, are supported and some Hugging Face pretrained models can be used to do so. Other advantages lies in its easy integration with existing Elasticsearch pipelines, and recent versions support native Lucene-based ANN indices. While integration with LLMs must be done manually via APIs or plugins, OpenSearch's maturity and extensive documentation make it ideal for heavy systems due to its index capabilities such as on disk search or FAISS integration.

**Vespa**, developed by Yahoo, offers high-performance vector search built into its core engine. It supports hybrid retrieval and can execute custom ranking functions directly at query time. Vespa's performance at scale and its ability to serve low-latency inference make it a good candidate for production systems but require a steeper learning curve and infrastructure overhead.

**Weaviate** is a relatively newer but LLM-focused vector search engine. It provides out-of-the-box support for semantic search and comes bundled with modules to embed documents using OpenAI, Cohere, or HuggingFace models. It also features a GraphQL API, simplifying integration. However, its BSL license may restrict commercial deployments, and while great for prototyping, Weaviate may lack fine-grained operational control needed for production-grade intrusion detection.

After comparing the options, **OpenSearch** is selected for this thesis due to its robust open-source support, flexibility in configuring vector-based search and the observability dashboard that makes it easy to deal with problems. Its ability to coexist with classical keyword-based indexing also allows hybrid pipelines that combine exact matches and semantic relevance.

### 3.3.1 Semantic Search Algorithm

To perform semantic retrieval of similar log entries in our RAG-based IDS architecture, two key components must be considered: the retrieval algorithm (semantic search method), and the similarity function (distance metric) used to compare embeddings. Each influences both performance and relevance of retrieved context.

Due to the support and versatility in OpenSearch, the $k$-Nearest Neighbors algorithm has been chosen as semantic search algorithm. In the context of vector-based search, it allows the system to identify the $k$ most semantically similar entries to a query embedding in a high-dimensional vector space which is particularly useful for log-based intrusion detection, where sentence embeddings created by the Transformers capture the semantic meaning of log entries. The core idea is to retrieve the vectors closest to the query vector based on a chosen distance metric.

The mathematical formulation for k-NN is as follows: given a query vector $q$ and a dataset of vectors $X = \{x_1, x_2, \ldots, x_n\}$, the goal is to find a subset $S$ such that:

$$S = \{x_i \in X \mid d(q, x_i) \text{ is among the } k \text{ smallest distances}\}$$

where $d(q, x_i)$ is a distance function such as cosine similarity or Euclidean distance. OpenSearch provides efficient implementations of approximate k-NN to scale this search to millions of vectors, which is critical for real-time use in large-scale intrusion detection systems. The support of multiple engines and indexing methods allows flexible adaptation to different performance, precision, and scalability requirements.

### 3.3.2 Optimizing Semantic Search

Efficient retrieval of semantically similar log entries is crucial for intrusion detection systems. OpenSearch enhances k-Nearest Neighbors search scalability and performance

through various indexing methods and engines. This section delves into the supported methods, Hierarchical Navigable Small World (HNSW) and Inverted File Index (IVF), and engines, Lucene, FAISS and NMSLIB, highlighting their mechanisms, and suitability for log data.

Among the indexing methods supported, **HNSW** is widely regarded as the most efficient for approximate nearest neighbor (ANN) search, introduced by Malkow in 2018 [16],based on the concept of multi-layer navigable graphs. It organizes data points into a hierarchy of layers, where all data points are on the bottom layer and each upper layer contains a sparser subset of the dataset. Each node,in our case log embeddings, connects to a fixed number of neighbors, exploiting the small-world network property to ensure short paths between any two points. Search begins from the top layer, using greedy traversal to move toward the query's nearest neighbor by evaluating only a limited number of candidates at each step, controlled by a variable. Once the closest point at a given level is found, the search proceeds to the next lower level, refining the results progressively until the base layer, which contains all points, and then the algorithm sorts all visited nodes based on similarity to the query vector and returns the top k nodes. The insertion process is similar to the search, the point finds its closest neighbor in each level and connects to them, but the point is only added in a randomly assigned maximum number of layers **L**, it will be added into all the layers from level 0 up to **L**. This technique is particularly well-suited for log data embedded using transformer models, as such embeddings lie in high-dimensional space and require efficient nearest-neighbor navigation.

**Key Parameters:**

- `ef_construction`: Controls the max number of candidate nodes explored per layer during insertion. Higher values result in a more accurate graph but slower indexing speed.
- `ef_search`: Controls the max number of candidate nodes explored per layer during search. Higher values boost accuracy but increase query latency.
- `m`: Determines the maximum number of bidirectional neighbors per node, higher values enhance recall at the cost of increased memory usage. Also, it directly affects both the graph's quality and the time needed for insertion.

**Search Complexity:** $O(\log N)$ due to its layered structure, which efficiently narrows the search area, where $N$ is the number of vectors in the index.

On the other hand, **IVF** method clusters the vector space into partitions during indexing. Each vector is assigned to a cluster based on its proximity to a centroid. At search time, only a subset of clusters determined by proximity to the query is used in the search. Because of this, search time is drastically reduced, particularly in datasets

with millions of entries. The efficiency of IVF depends on two main parameters: `nlist`, the number of clusters, and `nprobe`, the number of clusters to explore at search time. While IVF is extremely efficient and scalable, it generally performs best when the vector space is evenly distributed and the clustering is well-formed. In highly variable data environments like system logs—where the semantics of messages can vary significantly between sources—IVF may lead to lower recall unless carefully tuned.

**Key Parameters:**

- `nlist`: Number of clusters. Higher values may increase accuracy but also increase memory and training latency.
- `nprobe`: Number of buckets to search during a query. Higher values result in more accurate but slower searches.

**Search Complexity:** Approximately $O(nprobe \times \log(N/nlist))$, where N is the total amount of vectors. This search complexity changes a lot because we assume that all the clusters have the same amount of data but in practice it is not the case.

I have selected the **HNSW** method primarily due to its efficiency and simplicity. Unlike IVF, HNSW requires no prior training phase, making it ideal for our intrusion detection system where new data is constantly added, reducing indexing time and computation complexity. Furthermore, HNSW offers excellent support for filtering, especially with the Lucene engine, allowing hybrid queries that combine vector similarity with metadata based filters like timestamp, or log source. This flexibility, along with its high recall and scalability, makes HNSW the most practical choice for semantic log retrieval.

Supporting these methods are three main engines: Lucene, FAISS, and the now-deprecated NMSLIB. **Lucene**, being OpenSearch's native engine, provides a robust and stable implementation of HNSW. One of Lucene's main advantages is its support for filtering. This is particularly relevant in log search scenarios, where the user might want to restrict the search to logs from a specific host, service, or time range. Moreover, it is tightly integrated with the rest of OpenSearch's query DSL, allowing for hybrid queries that combine vector search with full text search or filtering. What is more, the Lucene engine support Scalar Quantization, that can decrease the memory footprint by a factor of 4 but in exchange of some loss in recall.

Second, Facebook AI Similarity Search, **FAISS**, is a highly specialized engine designed for high speed and large scale vector search. Both HNSW and IVF are supported by FAISS, along with advanced features like vector compression via Product or Scalar Quantization and optional SIMD (Single Instruction Multiple Data) optimization. This engine is particularly effective when indexing millions of dense embeddings, as is often

the case in large scale enterprise networks with lots of hosts and services. Although FAISS lacks native filtering capabilities compared to Lucene, its raw performance especially when used with Scalar Quantization and SIMD opstimization, makes it the preferred choice when retrieval speed and scalability are top priorities. It is widely adopted in deep learning pipelines where the vectors are generated in real time and pushed into a fast approximate search index.

Finally, Non-Metric Space Library, **NMSLIB**, once a competitive engine for ANN search, also implements HNSW but is now deprecated in OpenSearch. While it offered solid performance during its active period, it lacks many of the modern optimizations found in FAISS and does not support advanced filtering or integration with OpenSearch's internal mechanisms. For this reason, its use is no longer recommended for production systems.

Taking into account the proposed engines, FAISS is the most suitable engine for our research because it has better performance in larger networks, although Lucene supports hybrid queries in a native way, FAISS supports them too. Moreover, Scalar Quantization can be used in both and OpenSearch has different memory-optimized methods where the vectors are compressed and a disk-based search is supported in order to reduce memory usage and latency, but FAISS brings the option to a higher compression, saving more memory.

### 3.3.3 Distance Metrics

Each engine supports a variety of distance metrics, and choosing the correct one is critical for ensuring meaningful semantic retrieval. OpenSearch supports three main types: cosine similarity, Euclidean distance, and dot product.

**Cosine similarity** is the most commonly used metric in natural language processing because it captures the angular difference between two vectors regardless of their magnitude. This is especially effective when working with sentence embeddings from models like MiniLM or BERT, which are often normalized.

$$\text{cosine\_similarity}(x, y) = \frac{x \cdot y}{\|x\|\|y\|}$$

where $x \cdot y$ is the dot product of the vectors and $\|x\|$ and $\|y\|$ are their magnitudes. Cosine similarity returns values in the range $[-1, 1]$, with 1 indicating perfect alignment (i.e., semantic similarity).

**Euclidean distance** computes the straight-line distance between two vectors in space and is defined as:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

While intuitive and interpretable, Euclidean distance is sensitive to the magnitude of vectors. In high-dimensional spaces, particularly those generated by NLP models where vectors may not be standardized, Euclidean distance can perform poorly. It fails to capture semantic similarity effectively when differences in vector length do not correspond to differences in meaning, which is often the case with natural language embeddings.

**Dot product**, on the other hand, measures the projection of one vector onto another and is calculated as:

$$\text{dot}(x, y) = \sum_{i=1}^{n} x_i \cdot y_i$$

This metric is sensitive to both direction and magnitude, meaning that two vectors with similar orientation but very different lengths could yield a high or low score depending on their energy. In practice, this can lead to biased similarity assessments when vector magnitudes vary, which is undesirable when dealing with semantic embeddings that aim to capture meaning rather than intensity.

In this work, we use sentence embeddings generated by the `all-MiniLM-L6-v2` transformer model, which produces 384-dimensional vectors that are L2-normalized. This normalization ensures that each embedding has a unit norm, i.e., its vector magnitude equals 1. Under this condition, the distance metrics effectively capture angular relationships between vectors without being affected by embedding length. Despite this equivalence, cosine similarity remains the preferred metric for semantic retrieval tasks due to its intuitive interpretation, widespread use in NLP applications, and a safer choice if the index engine does not assume normalized vectors. Therefore, cosine similarity is selected as the distance metric for our semantic log retrieval system.

## 3.4 Analysis of Intrusion Detection dataset

Dataset selection and analysis are key aspects in every machine learning case. It will define how the final model will behave. Due to that, the dataset justification is going to be explain in this section. Also, a deep analysis of the dataset is going to be carried out.

The dataset chosen for the research is the AIT Log Data Set V2.0 developed by Max Landauer at al. [17], which is a comprehensive and realistic dataset designed to support research in intrusion detection and cyber attack analysis. Developed as part of a simu-

lated enterprise environment, it captures logs from various services and devices over a multi-day period and includes detailed annotations for attack phases. Unlike synthetic or oversimplified datasets, AIT Log V2.0 simulates real user and attacker behavior, including normal operations and different attacks. Each scenario spans several days and includes normal behavior from legitimate users, thus closely mimicking real world IT environments. The dataset includes logs from different system components such as web servers, mail servers, DNS, VPN, firewalls, and more, offering rich multi-source observability. The project was partially funded by the EU, so, the dataset is part of *EU Open Research Repository*,this is a key aspect because our research dataset should be free. On the other hand, the logs collection and simulation environment has been improved and now the enterprise network is larger and the types of attacks are also grater than the previous dataset called AIT-LDS-v1.x developed by the same group [18].

The AIT Log Data Set V2.0 comprises a diverse set of attacks to emulate realistic intrusion attempts.

1. **Network and Web Scanning**
   Attackers perform active reconnaissance to discover reachable hosts, open ports, services, and endpoints. In the simulation, these three tools are used, `nmap` send crafted packets to map network topology and infer operating systems. Web-focused scanners such as `WPScan` enumerate WordPress plugins, themes, and user accounts for known vulnerabilities. Directory brute-forcers like `dirb` probe for hidden files or directories by issuing HTTP requests for common paths.

2. **Webshell Upload**
   Adversaries exploit file-upload or plugin vulnerabilities, in our case, CVE-2020-24186 to upload files such as PHP or shell scripts to the server and execute them using simple HTTP POST/GET requests.

3. **Password Cracking**
   Attackers obtain user password hashes from compromised systems and attempt to crack them offline using tools like John the Ripper, used in this simulation. It normally uses a list of passwords and systematically test common or generated passwords against the hash until a match is found. Successful password cracks enable the use of valid credentials for further intrusion.

4. **Privilege Escalation**
   Once initial access is achieved, adversaries elevate privileges by exploiting OS misconfigurations or vulnerabilities. Common methods include abusing `sudo` misconfigurations, setuid binaries, kernel exploitation, `su` invocations, and modifications to user/group permissions.

5. **Remote Command Execution**
   RCE vulnerabilities permit attackers to execute arbitrary system commands on a remote host. Vectors include unsanitized input passed to shell calls in web applications or exposed APIs, frequently, HTTP requests containing shell syntax and executing them on the server.

6. **Data Exfiltration via DNS**
   DNS steal attacks attempt to disrupt the functionality of DNS servers as well as the resolution of domain names to IP addresses to redirect users to malicious websites or intercept their internet traffic to gain unauthorized access.

Moving on to data organization, logs in AIT Log V2.0 are organized under a mirrored directory structure: `gather/<host_name>/logs/` contains raw log files, and `labels/<host_name>/logs/` holds line-indexed JSON annotations indicating attack steps [17]. The dataset captures diverse logs from different sources sucha as Apache access and error logs on the WordPress server, Exim mail logs, VPN connection records, DNS query/responses, and Suricata IDS alerts. System events are recorded via syslog and auditd (Linux audit), while application-level activity emerges from Horde webmail and file share services. The `attacks.log` in `gather/attacker_0/logs/` enumerates the exact timestamps of each attack phase. Configuration manifests under `gather/<host_name>/configs/` and `facts.json` store host attributes (IP, OS, services). There is another three directories with additional data: the `processing` directory contains the source code that was used to generate the labels, the `rules` directory contains the labeling rules and the `environment` directory contains the source code that was used to deploy the testbed and run the simulation using the Kyoushi Testbed Environment.

In addition, each log file follows a standard format native to the respective service. For example, Apache access logs use the common log format with fields for IP address, timestamp, HTTP method, requested resource, and response code. Audit logs, in contrast, are composed of structured key-value messages, where event types such as `USER_AUTH` or `CRED_ACQ` encode user session behaviors. DNS logs document query types and resolving hosts, and Suricata alerts summarize packet-level detections in a readable and parsable JSON format. This diversity in format and content makes it really difficult to perform a deep preprocessing, due to that, we leave these patterns extraction to the Sentence Transformer. Although the preprocessing is going to be light, we assume that in real scenario we know the host or service that is sending the log to the IDS, so we extract this feature from the path of the log in our simulation, in next sections, this is explained.

Now that the introduction to the AIT Log Data Set V2.0, this dataset is divided in 8 different subsets with a certain simulation time and attack types. All of them are big enough to do the research, so, I choose the russellmitchell subset because every type

of attack is simulated and is the lightest one, around 14 GB and more than 11 million logs.

### 3.4.1 Russell Mitchell Subset

In this section we are going to explain the Russel Mitchell subset. First, the simulation is 4 days long, from the $2022-01-21T00:00:00$ till the $2022-01-25T00:00:00$ and all the logs from every service and host are collected. The simulated network, as seen in 3.1, is bounded and protected by a firewall and interacts with multiple public services including a mail server, VPN server, cloud file share, and a web server, all with public IP addresses. These components provide legitimate entry points into the internal network and simulate the attack surface of a real organization. The architecture supports all the mentioned intrusion scenarios such as web exploitation, credential brute force crack, DNS tunneling for data exfiltration, and lateral movement through privilege escalation. Attack paths generally begin from the public services, proceed through credential compromise or remote code execution, and eventually target critical systems inside the internal network. This setup offers a highly realistic and diverse environment for intrusion detection experiments, especially for evaluating behavior-based and context-aware models such as those powered by LLMs and vector search methods.

The network architecture is divided into two main network segments: `10.143.0.0/16` and `192.168.0.0/16`. The first segment simulates an internal corporate LAN composed of employee devices, an intranet server, a file share service, and a monitoring system. This segment reflects the core of a typical enterprise network where users interact with internal services and perform daily operations. Logs generated here reflect employee activities, service access patterns, and potential signs of internal compromise. The `192.168.0.0/16` segment simulates a private network where all the other entities are going to be connected and interact with the company services. External users, a DNS server, and email infrastructure including `Morris_mail` and `Davey_mail` compose the network. Additionally, multiple `Remote_employee` nodes and an `Attacker_0` node are included, which simulate remote workforce and malicious actors respectively.

Each log file follows a standard format native to the respective service. For example, Apache access logs use the common log format with fields for IP address, timestamp, HTTP method, requested resource, and response code. Audit logs, in contrast, are composed of structured key-value messages, where event types such as `USER_AUTH` or `CRED_ACQ` encode user session behaviors. DNS logs document query types and resolving hosts, and Suricata alerts summarize packet-level detections in a readable and parsable JSON format. This diversity in format and content makes it really difficult to perform a deep preprocessing, due to that, we leave these patterns extraction to the Sentence Transformer. Although the preprocessing is going to be light, we assume that in real

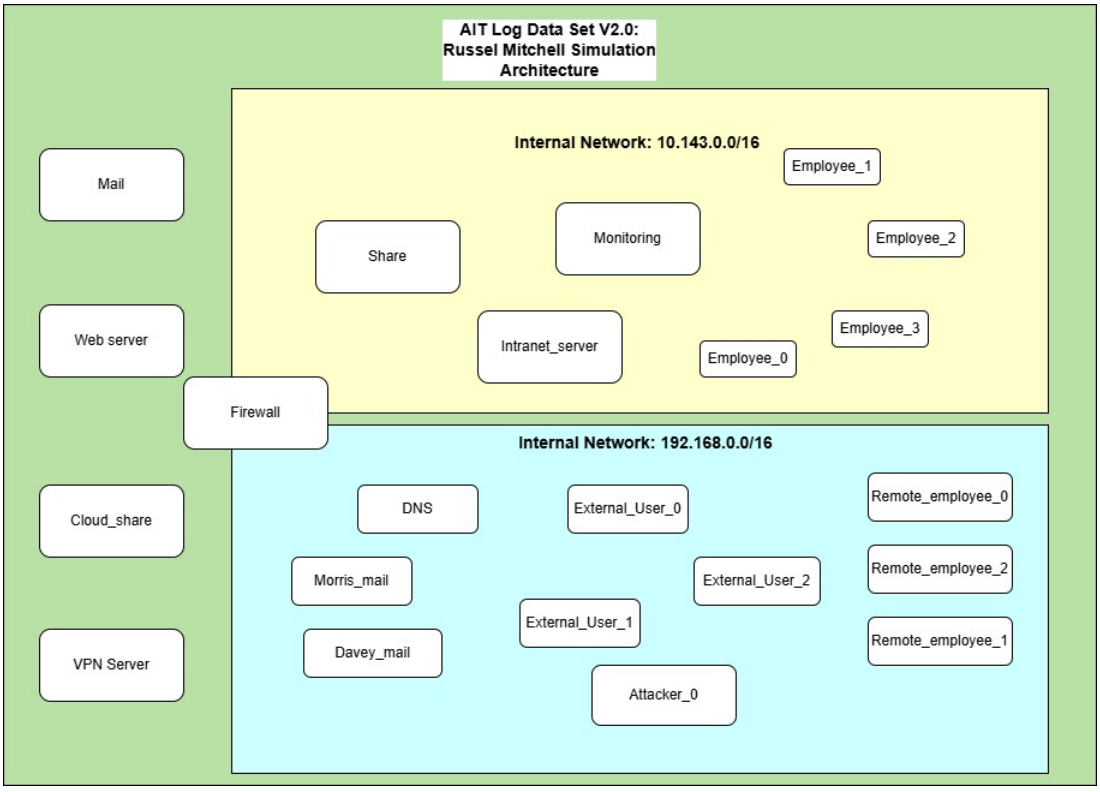


**Figure 3.1:** AIT Log Data Set V2.0 Simulation Architecture

scenario we know the host or service that is sending the log to the IDS, so we extract this feature from the path of the log in our simulation, in next sections, this is explained.

In conclusion, the AIT Log Data Set V2.0, particularly the Russell Mitchell subset, offers a uniquely rich testing ground for intrusion detection experiments using LLMs and vector search. Its authentic log formats, labeled attack scenarios, and structured organization directly support the goals of semantic retrieval and explainable reasoning in a RAG based IDS pipeline.

## 3.5 Analysis and selection of transformer & LLM

The transformer encoder selected for semantic log embedding is `all-MiniLM-L6-v2`, a lightweight and high-performance model available in the OpenSearch ML Commons plugin. This model generates 384-dimensional embeddings and is fine-tuned for semantic similarity tasks, making it ideal for use in semantic retrieval pipelines such as RAG. It is significantly smaller than alternatives like BERT-base, which has 110 million parameters and produces 768-dimensional embeddings. Despite its compact size of approximately 22 million parameters, MiniLM-L6 achieves competitive performance

while maintaining 5× faster inference time compared to larger models.

On the other hand, BERT, which is not natively supported by OpenSearch, has worse integration and higher latency. In the context of log-based intrusion detection, logs are parsed into structured textual templates before embedding. MiniLM's training on massive sentence pairs ensures that it captures semantic closeness between similar templates, which is critical for indexing and information retrieving. Although more expressive in theory, BERT often requires fine-tuning to perform similarly in semantic tasks and doubles storage and compute due to larger embedding size [19]. Therefore, MiniLM-L6-v2 is selected for its deployment readiness, low memory footprint, and strong semantic capabilities in log contexts.

For the inference and output generation in our IDS, we compare three competitive open-source LLMs: LLaMA 3 8B Instruct, Mistral 7B, and DeepSeek-R1 Distill (8B). All of them provide instruction-tuned models and natural language inference.

First, **LLaMA 3 8B** by Meta has strong performance in general tasks and was fine-tuned using Reinforcement Learning from Human Feedback (RLHF). Consequently, it produces fluent and safe output, supporting up to 8K context tokens. However, it offers slightly weaker reasoning scores in complex tasks compared to the other candidates. Nonetheless, its suitability for general-purpose use and its inference speed render it an optimal choice for real-time log assessment within RAG pipelines.

Second, **Mistral 7B** offers a strong balance of performance and efficiency. This is achieved through its implementation of grouped-query and sliding window attention mechanisms, which facilitate rapid inference and extensive context handling (32K tokens). Furthermore, benchmarks indicate that it surpasses larger models, such as LLaMA 13B, in both reasoning and mathematical tasks.

Third, **DeepSeek-R1 Distill (8B)** is a variant specifically engineered for reasoning, having been trained on chain of thought and computationally intensive mathematical problems. Moreover, it supports an impressive context window of 128K tokens and achieves state-of-the-art performance in logic and question-answering benchmarks. While it operates at a slower pace than Mistral, its exceptional proficiency in complex multi-hop inference makes it uniquely suited for in-depth forensic analysis across extended log sequences.

Finally, LLama 3.1-8b- Instruct has been selected for the study because it is lightweight and the inference time is really short. Also, larger context windows are not needed for our system because a log usually has less than 100 tokens and with 8k tokens, the RAG can be large enough to produce the best results. Also, the output can be easily customised in a more secure way.

# 4 Design & Implementation

This chapter presents the structural and technical blueprint of the proposed Intrusion Detection System. Both the conceptual design and its practical realization are covered, detailing the integration of KNN, large language models, and retrieval-augmented generation to process and analyze logs effectively in diverse detection scenarios.

## 4.1 Design

The design section outlines the architecture and core components of the system, focusing on how each module contributes to accurate intrusion detection. Furthermore, it provides a systematic overview of log collection, preprocessing, and the application of multiple detection strategies, emphasizing scalability, flexibility, and integration across detection systems.

### 4.1.1 General Architecture

High-level architecture diagram 4.1 illustrates a log-based Intrusion Detection System where logs generated by infrastructure components such as web servers, mail servers, DNS server, VPN and other monitored assets, are collected and forwarded to a centralized Preprocessor. This module standardizes and cleans the raw logs to ensure consistent formatting, enabling effective downstream analysis.

Once preprocessed, logs with the source are directed to one of three approaches. In the ML-Based Detection Engine, logs are sent to `OpenSearch` where they are first converted into semantic embeddings using the `MiniLM-L6-v2` Transformer model. Then, these embeddings are stored and later queried using `HNSW` indexing for carrying out an approximate nearest neighbor search. During inference, the model uses `cosine similarity` to retrieve the most semantically similar logs and applies a weighted voting classification policy based on the retrieved neighbors and similarity scores to predict the label of the log.

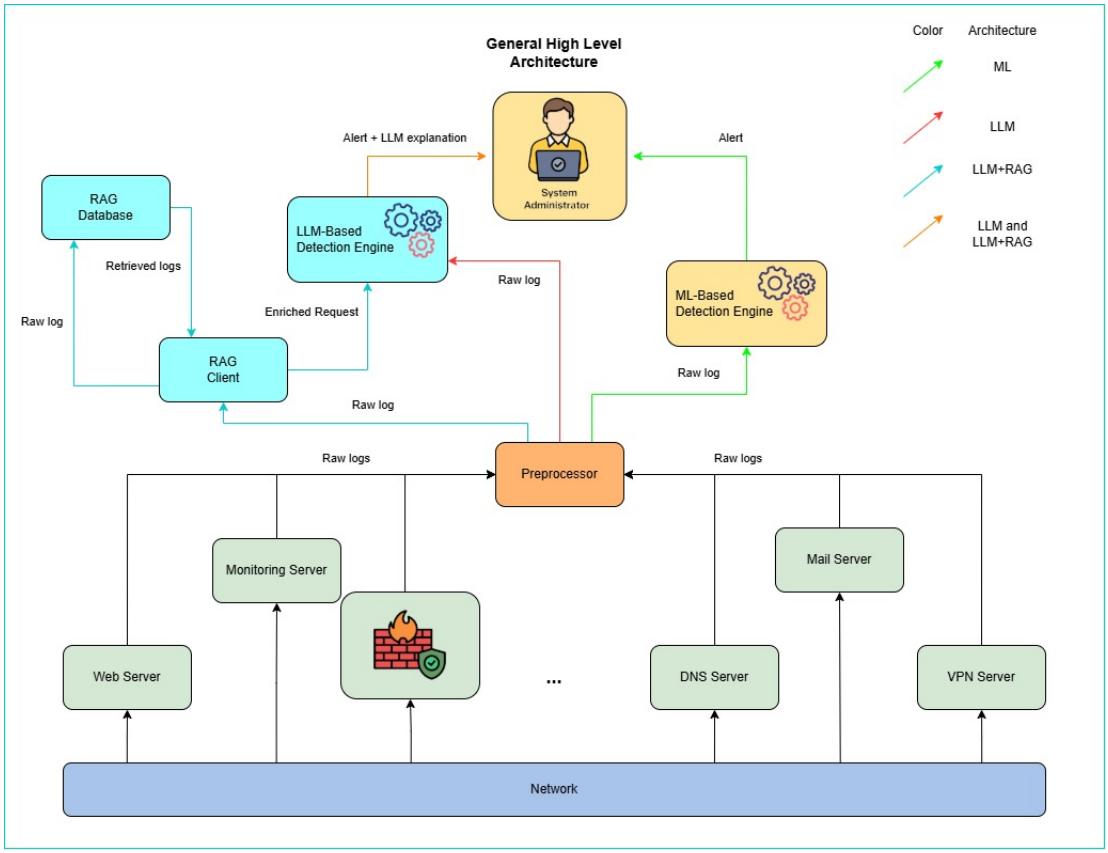


**Figure 4.1:** High Level Architecture Concept.

On the other hand, the LLM-Based Detection Engine processes logs directly using `LLaMA 3.1-8B Instruct`, relying solely on its pre-trained knowledge without external context. The model predicts the label and provides an explanation for the decision.

Third, the LLM + RAG Engine, enriches the raw log input by integrating contextual examples. Logs are first feed to a RAG Database that is `Opensearch` with the same configuration as in the ML approach. After that, the K most similar past logs are fetched and combined with the current log to create an enriched prompt for the LLM. The output is the predicted label with a decision explanation.

Finally, all detection engines can trigger alerts to the system administrator when an anomaly is inferred, either only the label of the log, in the ML approach or the label with a human-readable explanation generated by the model, on the LLM systems.

### 4.1.2 Scenarios Architecture

To evaluate the performance of different intrusion detection methods, three experimental scenarios were implemented. Each scenario leverages the same log dataset but differs in how the embedded logs are interpreted and classified. The aim is to measure how the addition of semantic models and contextual reasoning mainly affects detection accuracy and recall.

- **Scenario 1: KNN-Based Detection**
  In this baseline scenario, logs are embedded using the MiniLM model and stored in OpenSearch. At query time, the k-nearest neighbor algorithm retrieves the most similar historical logs based on cosine similarity. Detection decisions are made by comparing labels or patterns from the retrieved neighbors.
- **Scenario 2: LLM-Based Detection**
  In this setup, each new log is passed through the same embedding process, and semantically similar logs are retrieved from OpenSearch. However, instead of relying purely on similarity labels, an LLM (e.g., LLaMA 3 8B Instruct) interprets the log in natural language and decides whether it is malicious or benign.
- **Scenario 3: LLM + RAG-Based Detection**
  The most advanced scenario combines semantic search with Retrieval-Augmented Generation (RAG). Here, retrieved similar logs are included as context for the LLM, enabling it to reason over both the current log and historical incidents. This setup provides natural language justifications and deeper contextual awareness.

## 4.2 Implementation

This section details the full implementation of the proposed system, starting from log collection and preprocessing to the deployment of vector indexing, KNN classification, and LLM-based evaluation, explaining how components like OpenSearch, Transformers, and LLM are integrated.

### 4.2.1 CSV Creation and Preprocessing

Due to the memory comsumption and difficulties of extracting, partioning and indexing the logs directly from the dataset directories, create a csv with the needed data show up as the best idea.

The CSV generation process for the intrusion detection dataset is implemented using a custom Python script that traverses a structured directory of log files and corresponding label annotations. The source directory, named `gather`, is recursively explored to identify files with the `.log` extension. For each valid log file, the source host or service

is inferred from the directory path, allowing the CSV to include a `source` column in addition to the `log_message` and `label` columns. To ensure scalability and uniformity, a maximum of 1.000.000 lines are ingested per log file. This design enables efficient streaming of log data and prevents memory overflows during parsing.

Each log message is labeled using a JSON file from a mirrored `labels` directory, which stores ground-truth annotations based on line numbers. A helper function parses these JSON entries, extracting structured labels for attack steps such as `attacker_change_user` or `exfiltration`. If no label is found for a line, the log is marked as `normal_log`, indicating benign system behavior. This line-based labeling strategy is simple but effective and align logs with simulated attack time. Furthermore, the approach avoids mislabeling caused by preprocessing artifacts like shuffling or sorting.

Minimal cleaning is applied before writing to the CSV. Log lines are stripped of whitespace on the start and end, empty entries, and non-alphanumeric lines are discarded. Although timestamp parsing was implemented in a helper function, it was excluded from the final CSV to focus on the log content and semantic embedding rather than temporal modeling because timestamp extraction and normalization is very difficult due to the variety of timestamp formats. Additionally, only logs from valid services (e.g., `mail`, `vpn`, `apache`, etc.) are preserved, ensuring that irrelevant or misaligned data do not skew the model. Before splitting the data, dealing with large and inherently imbalanced log datasets, downsampling the 30% of normal logs is recommended. Primarily, this action significantly reduces the overall dataset size, from 11 million logs, more than 10 million are normal logs,so , the final downsampled dataset is around 8 million rows. Thereby conserving computational resources required for subsequent embedding, indexing, and retrieval operations. More importantly, this reduction concentrates the LLM's exposure on the rarer, yet more critical, attack log entries. This focused exposure enhances the relevance and precision of retrieved contexts during anomaly detection, preventing the system from being disproportionately influenced or overwhelmed by voluminous normal data.

The final CSV is split into training and evaluation subsets using a stratified 80-20 ratio. This split maintains the distribution of log types and attack classes, providing a reliable basis for training and benchmarking the intrusion detection system. Both subsets are then persisted as separate CSV files. The application of stratification during this splitting process ensures that the proportional representation of all log labels, including all attack types and the remaining normal logs, is meticulously preserved across both the training and evaluation subsets. This meticulous preservation mitigates the risk of data imbalance distorting evaluation metrics, such as scenarios where rare attack patterns might be entirely absent from one set, leading to biased model training or unrepresentative performance assessments. Moreover, a more severe downsampling has been taken out and only 40.000 normal logs have been kept with all the attacks

logs that represent around 12.000 logs, therefore, the evaluation dataset has 52.000 samples.

### 4.2.2 OpenSearch Vector Database and Indexing

To enable scalable and efficient semantic search over log data, OpenSearch version 2.19 was employed as the core vector database engine. Its k-NN plugin allows dense vector fields to be indexed and queried using Approximate Nearest Neighbor (ANN) techniques. Each log entry is paired with an embedding vector derived from the `all-MiniLM-L6-v2` transformer-based model, the source and the label of the log. These vectors are stored in the `embedding` field, which is defined with the `knn_vector` type and marked with `"index.knn": true` in the index settings.

To support high-volume search workloads with minimal memory overhead, the index uses `mode: on_disk`, a feature that stores vectors on disk instead of keeping them in RAM. This is complemented by a high `compression_level` of 32x, significantly reducing storage cost while maintaining reasonable recall performance. Compression is applied through scalar quantization via the Faiss engine, a high-performance similarity search library.

The ANN method chosen is Hierarchical Navigable Small World, configured with `m: 35` and `ef_construction: 64`, which builds a multi-layered proximity graph that enables fast and accurate vector retrieval. Cosine similarity (`space_type: cosinesimil`) was selected as the distance metric due to its alignment with normalized embeddings produced by sentence transformers. Overall, this configuration ensures low-latency, semantically rich log retrieval suitable for LLM-based threat detection systems.

#### Pipeline

In order to create the embedding in the most optimal way, an **OpenSearch's pipeline** is needed to create the embedding in execution time inside OpenSearch. To address this, the all-MiniLM-L6-v2 transformer-based embedding model is deployed in OpenSearch. The embedding pipeline is installed and deployed using the OpenSearch REST API, executed through its Dev Tools CLI. It begins by registering a model group and registering the MiniLM model using its Hugging Face identifier. Once the model is properly configured, a custom ingest pipeline (log-filter-pipeline) is defined. This pipeline makes use of the text embedding processor in order to project the model into the log field of incoming documents and write the resulting vectors into a new embedding field. Each of these operations, all done via HTTP POST and PUT requests to OpenSearch's _plugins/_ml endpoints, serves to make semantic enrichment of logs automatic in advance of future k-NN vector search.

### 4.2.3 ML based Scenario

The Machine Learning based scenario is based on a Transformer to create embedding from logs and a knn search to retrieve the most similar logs and with a certain policy, label the log. This scenario needs training, index the log in the vector database to subsequently do the vector search and label a new log. Furthermore, this vector database is the same that we are going to use for the RAG, so, only one train is needed.

#### Training

The training flow, as depicted in Figure 4.2, illustrates the ingestion and semantic embedding process that enables OpenSearch to build the vector-based index for intrusion detection. The process begins with a Python client that is used to load the csv as a Dataframe and interact with the OpenSearch API, passing individual log entries for indexing, using a function that is going to be explained below.

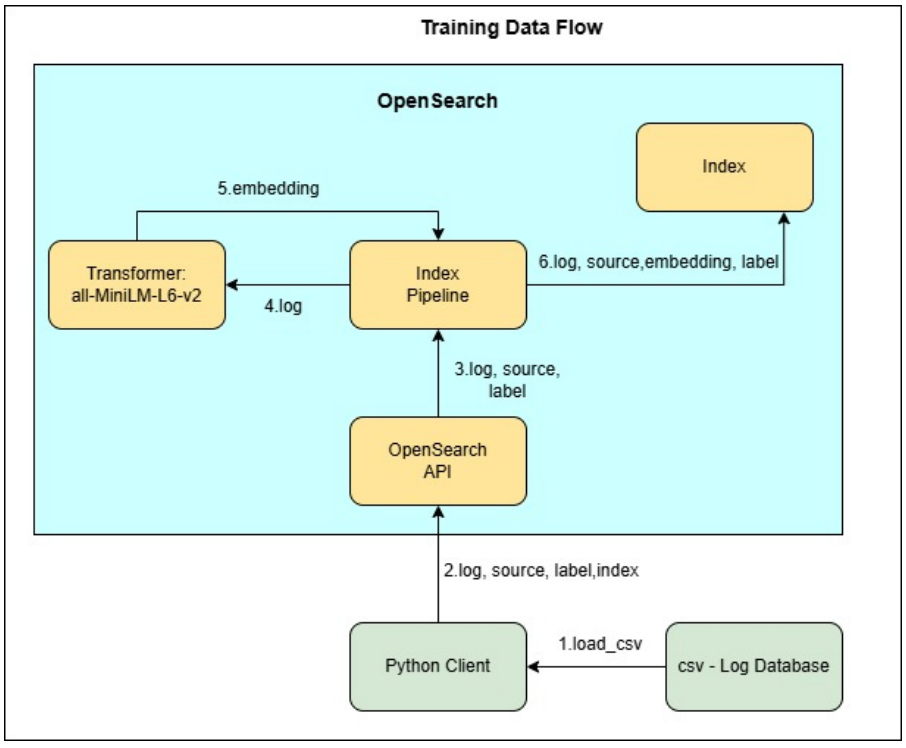


**Figure 4.2:** Training phase.

Within OpenSearch, the ingestion pipeline plays a critical role in transforming raw log messages into dense vector embeddings. This is accomplished via the `text_embedding` processor, which uses the MiniLM-L6-v2 transformer model. As each log passes through

the pipeline, the transformer generates a 384-dimensional embedding that semantically represents the log's content. This embedding, along with the original log, source, and label, is then stored in a designated OpenSearch index.

The architecture shown here ensures modularity and scalability in the training pipeline. By separating log ingestion from vector embedding and indexing, the system remains flexible and easy to extend with alternative transformer models or indexing strategies. Furthermore, using the OpenSearch ML plugin to deploy the transformer natively allows embedding to be executed directly in the ingestion flow, avoiding the need for external preprocessing services. The generated index becomes a central semantic knowledge base for subsequent intrusion detection models, whether based on traditional KNN or LLM-driven analysis. This design balances performance, explainability, and real-time requirements for effective IDS deployment in high-volume environments.

The ingestion of logs into OpenSearch is managed by a Python client that processes the dataset in batches to optimize throughput and minimize network overhead. The core of this operation is the `ingest_batches_from_csv_pipeline` function 4.1, which takes a DataFrame, OpenSearch client instance, and an optional batch size parameter. The function reads the dataset row by row, extracting the log message, its source, and corresponding label. These are temporarily stored in buffers until the batch size is reached or the end of the dataset is encountered. Once a complete batch is assembled, a list of index actions is created, which formats the documents appropriately. The batch is then sent to OpenSearch in bulk using a custom `send_actions` function, targeting the configured index. After each submission, the buffers are cleared, and the process repeats until all logs are processed.

Listing 4.1: Python ingestion function used to batch logs and send them to OpenSearch

```
def ingest_batches_from_csv_pipeline(data, client, batch_size=32):
    actions = []
    batch_logs = []
    batch_labels = []
    batch_sources = []
    for idx, row in enumerate(data.itertuples(index=False)):
        batch_logs.append(row.log_message)
        batch_sources.append(row.source)
        batch_labels.append(row.label)

        # When batch is full or at the end of the dataset
        if len(batch_logs) == batch_size or idx == len(data) - 1:
            # Build actions
            for log_message, source, label in zip(batch_logs, batch_sources, batch_labels):
                actions.append(create_action(log_message, source, label))

            # Bulk insert
            if actions:
                send_actions(client, actions)
                print(
```

```
                f"Ingested {len(actions)} documents into '{INDEX_NAME}' index.")

        # Reset batches
        actions = []
        batch_logs = []
        batch_labels = []
```

## Evaluation

To evaluate the detection capabilities of the k-NN-based scenario, a semantic similarity search is conducted on previously indexed log embeddings. Each test log is embedded and compared to existing indexed logs using the cosine similarity metric. This evaluation allows measurement of the system's ability to classify unseen logs based on their semantic proximity to known threats.

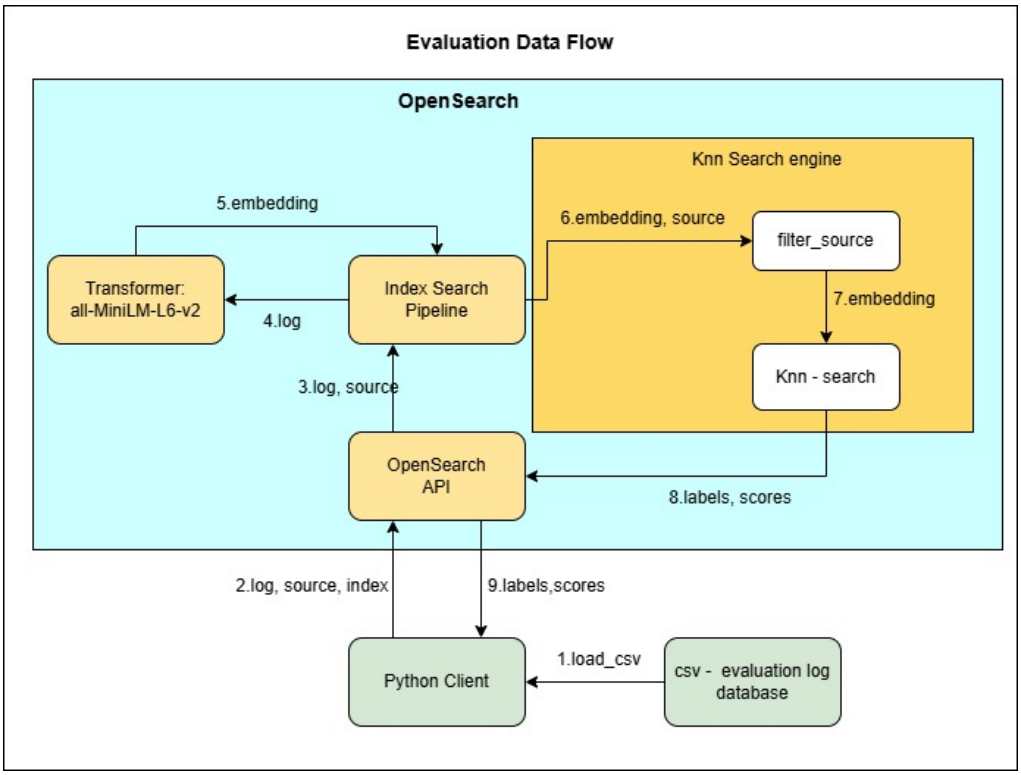


**Figure 4.3:** Evaluation Data Flow for k-NN-Based Intrusion Detection

Figure 4.3 illustrates the evaluation phase architecture, where each log from the evaluation CSV dataset is processed through a Python client. Logs are first sent to OpenSearch, where they are embedded using the same MiniLM-L6-v2 transformer through the `Index Search Pipeline`. Once embedded, the k-NN engine retrieves the most similar historical log embeddings by comparing vector distances—typically using cosine similarity. A filtering step ensures that logs are compared only against their

sources to preserve contextual integrity. The k-NN module returns the K labels of the nearest neighbors with their similarity scores, which are sent back to the Python client to apply decision policy and return only one label.

The **decision policy** is implemented as a weighted voting policy to infer the label of a new log based on the results retrieved from a k-NN similarity search. Each retrieved log contributes a score associated with its label, and these scores are summed per label category. To assess the overall distribution, the scores are normalized into percentages. Although optional alerts can be triggered to indicate whether the majority of neighbors are anomalous, not in my test but recommended in deployment, the final decision is made by selecting the label with the highest accumulated score. This method allows for decision making based on proximity confidence rather than a strict majority vote.

### 4.2.4 LLM Scenario

The LLM scenario is based on the inference power of the Llama-3.1-8B-Instruct, deployed in another machine and accessible via REST API. As we can see in the figure 4.4, the Python client sends the prompt with the log and source to be classified and the LLM returns the label and a short explanation of the decision. Therefore if the log is not normal, the client can send a message with the label and explanation to the system administrator for instance.

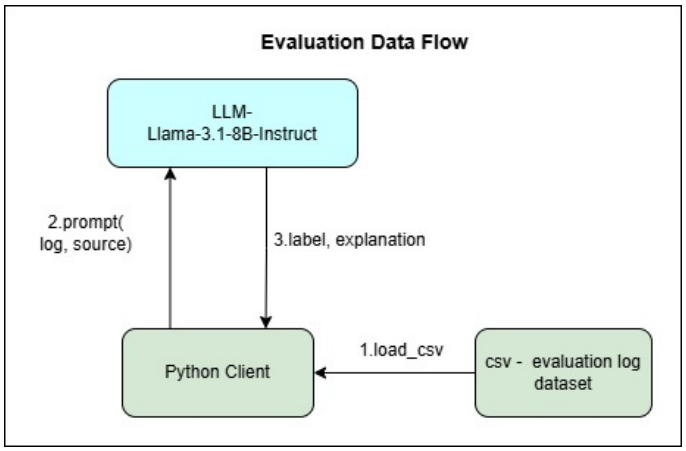


**Figure 4.4:** Evaluation Data Flow for LLM-Based Intrusion Detection

In order to reduce the evaluation time, asynchronous HTTP requests are implemented to query the LLM in an asynchronous way. Apart from that, the LLM request comprises two essential components: the system prompt and the user input. The system prompt explicitly defines the context and role of the LLM as a log-based Intrusion Detection System, specifying the types of services available within the monitored network, including web servers, cloud file shares, mail servers, VPN gateways, and

more. It further instructs the LLM to classify log entries strictly into the predefined labels, such as `normal_log`, `dnsteal`, or `attacker_http`, among others. The prompt mandates that the output adheres precisely to a JSON schema, incorporating both a classification label and a concise explanatory reasoning limited to 50 words. Complementing the system prompt, the user input includes dynamic content composed of the log message itself and its source, enhancing contextual accuracy for classification.

On the other hand, variables such as `temperature`, `max_tokens`, `stream` and `guided_json` significantly influence the behavior of the LLM: A low temperature (0.2) ensures deterministic and consistent output suitable for precise and reliable classification. The `max_tokens` parameter caps the length of responses to 150 tokens, enforcing conciseness and computational efficiency. Also, stream is set to False because it is not needed for the research and finally, the `guided_json` is set to the Class Response json schema with label and explanation. Consequently, these carefully calibrated variables and detailed instructions ensure systematic, interpretable, and reliable LLM outputs, crucial for accurate and efficient real-time log evaluation.

Finally, the `main_async` function orchestrates the evaluation of an entire dataset of logs using concurrent requests to the LLM API. For each log entry in the evaluation DataFrame, it creates an asynchronous task via `process_single_alert` and collects them into a task list. A semaphore limits the number of concurrent requests to avoid overwhelming the server. Once all tasks are defined, `asyncio.gather` executes them in parallel. The results are then processed: successful responses are parsed to extract predicted labels and real labels, which are stored in `y_predict` and `y_true` lists respectively. These serve as inputs to the `analize_results` function, which evaluates model performance. During processing, misclassified or anomalous logs are logged to a file with the explanation of the LLM for later inspection. This function enables scalable, label-aware evaluation of LLMs under realistic conditions and supports detailed analysis of their behavior.

### 4.2.5 LLM with RAG scenario

The last scenario is based on a `Llama-3.1-8B-Instruct` with a RAG, which is also based on a Knn search of the most similar logs. Initially, as we can observe on the diagram, the Python client send the new log to OpenSearch, where a transformer model (`all-MiniLM-L6-v2`) encodes the messages into embeddings and performs a KNN search to retrieve similar past entries, filtered by source, returning relevant labels and confidence scores to the client. The KNN search results, which form the RAG context, are then combined with the new log and its source into a structured prompt. This enriched prompt is sent from the Python client through a REST API to the LLM, by incorporating both semantic retrieval (labels and scores from similar logs) and new log content. Then, the LLM returns the label and explanation for the new log to the

client. Finally, this labeled log is stored in OpenSearch, enriching the index for further retrievals.

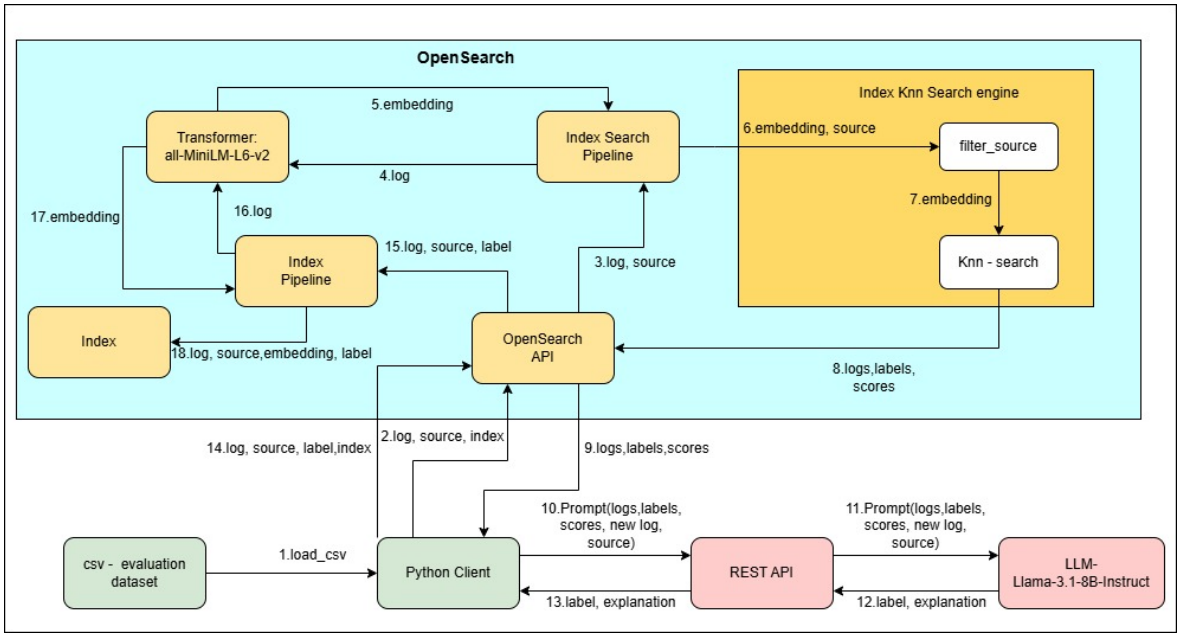


**Figure 4.5:** Evaluation Data Flow for LLM+RAG-Based Intrusion Detection

In this scenario the prompt sent to the LLM changes a little from the normal LLM prompt mentioned above. Although most of the variables remains the same, the system and user inputs should be lightly converted. A few indications are added on the system message, first, now in the user message you have: A list of similar past logs with known labels and the similarity score, and a new log message with the source to classify. Second, a clear message to use the RAG guiding the decision: "Use the context logs to inform your prediction. If the new log is similar to known attack patterns, label it accordingly". Third, in the explanation, add if your decision is based on the provided list of log or not. Finally, the RAG joint with the source and log to classify are passed through the user message. Everything is shown on the code below, which is the process_single_alert function, responsible for processing a single LLM query, although some of the code has been cut.

Listing 4.2: Python function to process a single LLM with RAG query

```
SYSTEM_PROMPT = (
    "You are a log-based Intrusion Detection System (IDS). "
    "The network includes this services: a web server, cloud file share, mail servers, VPN gateway
        ↪ , DNS, internal intranet, a firewall, four internal employees, three remote employees,
        ↪  and three external users. "
    f"Assign a label from the label list: {list(LabelList.__args__)} based on the content "
    "Your response must strictly follow the JSON format defined in the schema provided. "
    "Only return valid labels allowed by the schema."
    "You will be provided with:\n"
```

```
    "1. A list of similar past logs with known labels and the similarity score.\n"
    "2. A new log message with the source to classify.\n"
    "Use the context logs to inform your prediction. If the new log is similar to known attack patterns, label it accordingly. "
    "In addition to the label, provide a short explanation (in plain English, max 70 words) describing the reasoning behind your decision. Let me know if your decision is based on the provided list of logs or not."
)

async def process_single_alert(
    session: aiohttp.ClientSession,
    semaphore: asyncio.Semaphore,
    log_index: int,
    log_message: str,
    source:str,
    open_search_client,label:str,
    k
) -> Dict:
    """Processes a single alert by sending it to the LLM asynchronously."""

    async with semaphore:
        rag = get_rag(log_message,source, open_search_client,k)
        user_input = f"""Context logs:
            {rag}\n
            Classify:
            source: {source} ,log message: \"{log_message}\"
            """
        request_start_time = time.monotonic() # Time for this specific request
        try:
            payload = {
                "model": MODEL_NAME,
                "messages": [
                    {"role": "system",
                    "content": SYSTEM_PROMPT},
                    {"role": "user",
                    "content": f"{user_input}"}
                ],
                "temperature": 0.3, # Adjust for creativity vs. determinism
                "max_tokens": 150, # Adjust based on expected response length
                "stream": False,
                "guided_json": JSON_SCHEMA
            }

            async with session.post(LLM_URL, json=payload, timeout=1200) as response:
                if response.ok:
                    result = await response.json()
                    try:
                        return {
                            "index": log_index,
                            "log_message": log_message,
                            "source": source,
                            "llm_response": Response.model_validate_json(result["choices"][0]["message"]["content"]),
                            "true_label" : label
                        }
                    except Exception as e:
                        print("An Error occurs validating the JSON:")
```

### 4.2.6 Testbed Hardware Details

To support the experimentation and evaluation of the proposed system, two distinct hardware setups were employed, each aligned with a specific architectural component. Table 4.1 shows the main characteristics of both.

First, the KNN approach, which is also used for log retrieval in the LLM+RAG pipeline, was executed on a personal laptop running OpenSearch v2.19 with Docker. Also, the Sentence Transformer was deployed here enabling CUDA to be executed on the GPU.

| Component | OpenSearch Laptop | LLM Server |
|---|---|---|
| **CPU** | Intel Core i7-10750H | AMD EPYC 9554 64-Core |
| **GPU** | NVIDIA GTX 1650 Ti (4GB) | 2× NVIDIA H100 NVL (94GB each) |
| **RAM** | 16 GB | 1 TB |
| **Operating System** | Windows 10 | Debian 12 |
| **Python Version** | 3.11 | 3.10 |
| **CUDA Version** | 11.2 | 12.1 |

**Table 4.1:** Comparison of Hardware and Software Environments

On the other hand, the Large Language Model was deployed on a high-performance server designed for deep learning workloads. The LLM was served using vLLM v0.8.5.post1 and FlashInfer v0.2.2.post1, enabling optimized inference. Notably, all evaluations were run on a single GPU, and no model or tensor parallelism was applied, despite the availability of multiple GPUs.

# 5 Evaluation results

In this section, based on the proposed methodology and implementation, a detailed description of our evaluation results is provided. For this purpose, the following metrics are considered:

- Accuracy: $\frac{TP+TN}{TP+FP+FN+TN}$
- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1-score: $2 * \frac{Recall*Precision}{Recall+Precision}$

where TP: true positives, TN: true negatives, FP: false positives, and FN: false negatives.

Given that a multi-class classification is done, some metrics such as precision, recall and F1-score can be calculated per label and also using micro, macro or weighted averaging. According to the sklearn official documentation for this part [20], the micro averaging gives each pair sample class an equal contribution or importance to the overall metric, rather than summing the metric per class. Macro averaging calculates the mean of binary metrics, giving equal weight to each class. By last, weighted average accounts for class imbalance, by computing the average of binary metrics in which each class score is weighted by its presence in the true data sample. Since our data partitions are generally imbalanced, in particular in the basic scenario, we will configure metrics to use weighted averaging.

Moreover, the classification_report calculates the precision, recall and f1-score per class and also the support that is the number of samples of each one that have been classified. In addition, the classes with less than 10 samples have been removed from the plots because they represent a tiny part of the dataset and the metrics are normally either 0 or 1, but all the classes are used to calculate the total accuracy. A confussion matrix has been calculated per model in order to display the results better and to have the opportunity to make the calculations of new metrics such as False Positive Rate.

On the other hand, the only training we need is to ingest the training logs to the OpenSearch index to retrieve them later (Section 4.2.3). Furthermore, the evaluation dataset which is used for the models evaluation is explained in 4.2.1.

## 5.1 ML based Scenario

In this section, the $k$ value is the first thing to discuss and later the evaluation and explanation of the classification results.

The selection of the optimal $k$ value in KNN-based retrieval hinges on a balance between classification performance and computational efficiency. These times and accuracies have been calculated with a 1000 samples dataset in order to do a quick research of the most suitable value. As illustrated in Figure 5.1, $k = 3$ achieves the highest classification accuracy of 96.19%, outperforming $k = 5$ and $k = 7$, which score 95.23% and 95.21% respectively. This suggests that using a smaller neighborhood yields more precise label predictions, likely due to the tighter semantic similarity among closer neighbors. Larger values of $k$ introduce additional, possibly less relevant, log entries, diluting the influence of the most pertinent ones and slightly decreasing classification effectiveness.

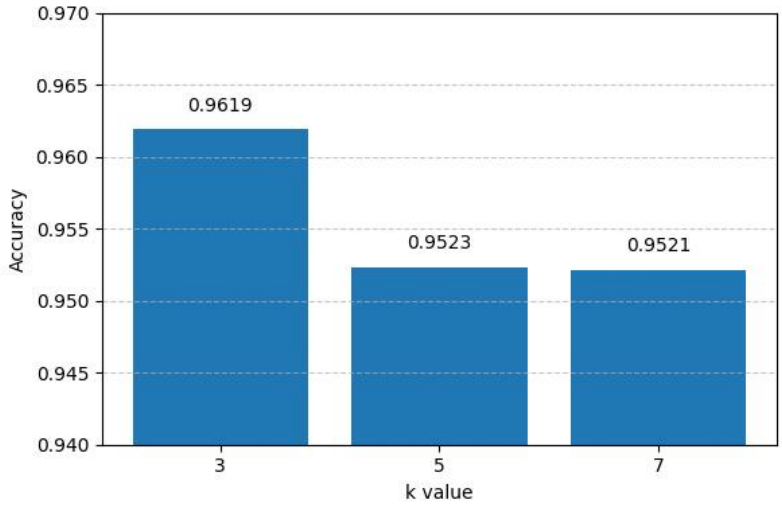


**Figure 5.1:** KNN classification accuracy across different $k$ values.

Moreover, as shown in Figure 5.2, $k = 3$ not only provides superior accuracy but also the fastest query time at approximately 0.133 seconds. This is significantly faster than the 0.141 seconds recorded for $k = 5$, representing an improvement of about 5.8% and to $k = 7$, which takes 0.214 seconds, $k = 3$ is roughly 60% faster. What is more, a lower k value uses less memory,so , reduces the computational cost. In the context of a real-time intrusion detection system, low latency is critical. Faster query execution enables the system to handle larger volumes of incoming logs efficiently. Given the dual advantage of higher accuracy and lower computational cost, $k = 3$ represents the optimal choice for KNN-based log retrieval and classification within the current

architecture.

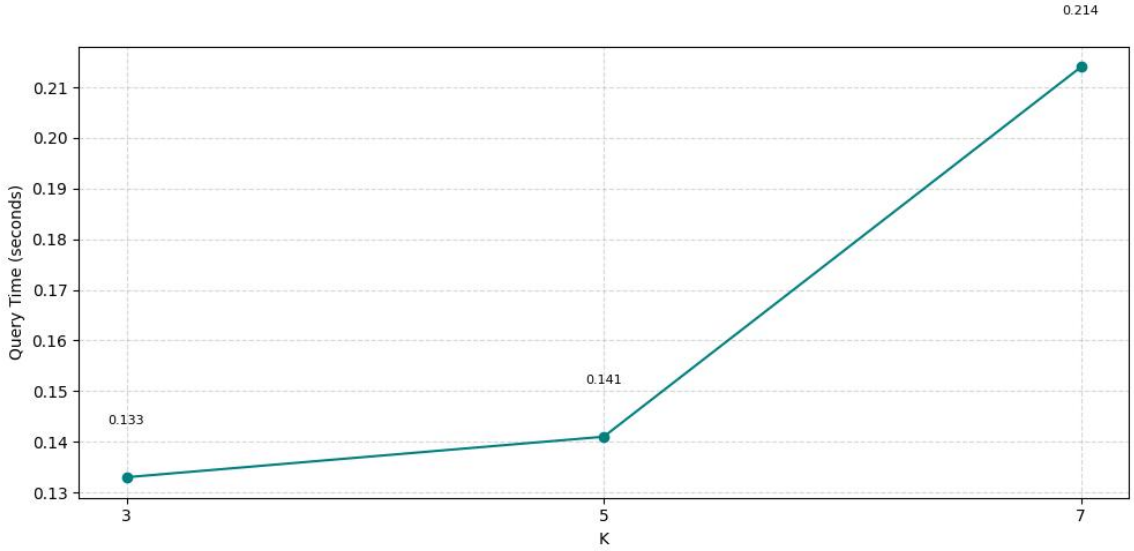


**Figure 5.2:** Query time (in seconds) for different $k$ values.

Changing to the evaluation results, the overall accuracy is 96.2%, meaning that out of all 50,307 evaluated instances, 48.396 were correctly classified. Building on this, additional performance metrics are presented as weighted averages, which take into account the relative frequency (support) of each class to ensure a balanced evaluation. The weighted precision is 0.963, indicating that when the model predicts a class, it is correct approximately 96.3% of the time. Similarly, the weighted recall is 0.962, showing that the model successfully identifies 96.2% of the true labels across all categories. These two measures are harmonized in the weighted F1-score, which stands at 0.960 and offers a single metric that balances false positives and false negatives. Taking everything into account, the high accuracy and tightly aligned weighted scores suggest that the KNN model maintains consistent and robust performance, even in the presence of class imbalance or variation in category frequency.

Figure 5.3 illustrates the precision, recall, and F1-score for each class. The classes `attacker_http` and `dnsteal` exhibit perfect scores (1.00) across all metrics, indicating flawless prediction performance. `normal_log` and `service_scan` also perform strongly, with scores above 0.90 in all categories, demonstrating the model's reliability in both benign and common attack detections. Meanwhile, `dns_scan` shows high precision but relatively lower recall (0.58), which suggests under-detection of true instances and room for improvement in sensitivity. This pattern is also reflected in a lower F1-score (0.73) for the class.

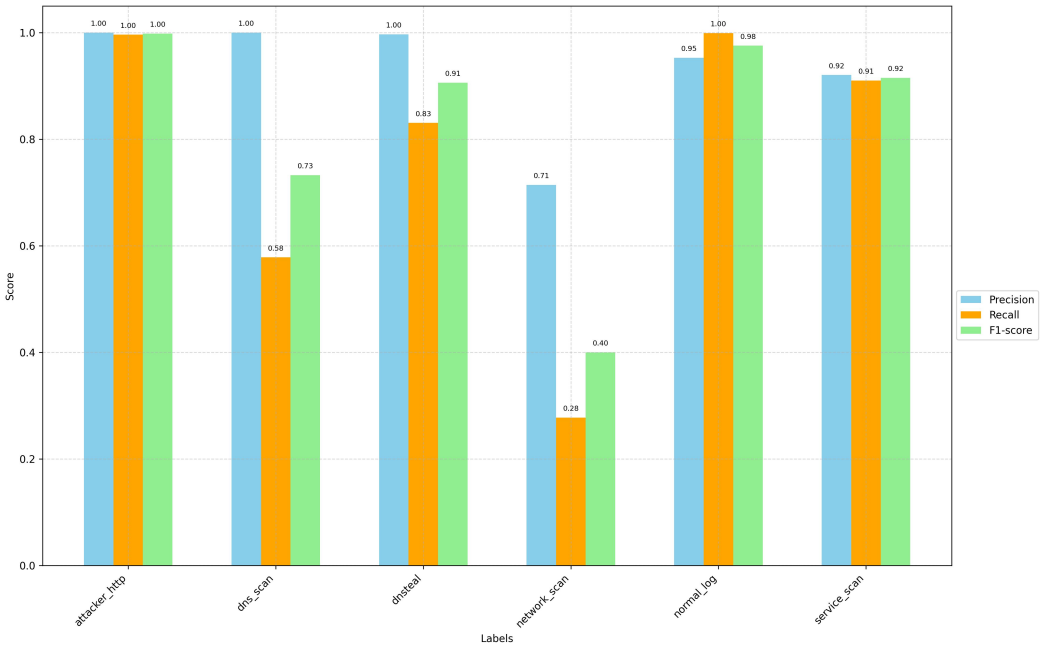


**Figure 5.3:** Precision, Recall, and F1-score per label for the KNN-based model.

The most concerning performance is observed in the `network_scan` class, which has a recall of just 0.28 and an F1-score of 0.40, despite a moderate precision of 0.71. Therefore, the model struggles significantly to identify true instances of this class, likely due to overlapping features with other types or data scarcity. The disparity between high precision and low recall in some classes reflects a conservative bias in the model, it predicts certain labels only when very confident. This could be because most of the logs on the dataset are normal_log and it has a little overfitting with this label. In addition, the low precision and recall in certain labels might be due to the low number of samples of the labels, for instance, network_scan are only 18 in the evaluation dataset whereas there are 1545 attacker_http samples. Overall, the visualization highlights strong model performance on dominant classes, while revealing opportunities to enhance recall and generalization in less frequent categories.

## 5.2 LLM Scenario

The LLM scenario involves classifying logs based solely on the pre-trained knowledge of the Llama-3.1-8B-Instruct language model, without any external context or retrieval. This setup tests the model's intrinsic ability to generalize and detect threats from raw log inputs.

First, the LLM model achieves a high overall accuracy of 88.0%, indicating that a substantial portion of the predictions align with the ground truth. However, this value is heavily influenced by the dataset's imbalance, where approximately 75% of the instances belong to the `normal_log` class. This imbalance inflates the weighted average metrics, such as a precision of 0.96, recall of 0.88, and F1-score of 0.90, since these metrics give more weight to frequent classes. The high precision suggests the model makes few false positive predictions overall, but the recall implies that some attack instances may still be missed. The F1-score balances both factors, yet remains skewed toward the majority class performance. While the model performs reliably on common categories, like benign logs, these scores may not reflect its effectiveness on minority attack classes. If we take the macro average metrics, which treats all classes equally, reveals much lower values: a precision of 0.21, recall of 0.30, and F1-score of just 0.15. This suggests that the model performs poorly across many minority classes, likely predicting the majority class more frequently. Thus, while the metrics appear strong, they highlight the model's limited ability to generalize across diverse threat types without additional contextual information.

Second, the precision, recall, and F1-scores per class are presented in Figure 5.4. As seen, the model achieves strong results for `normal_log`, with a precision of 0.98, recall of 0.96, and F1-score of 0.97, indicating reliable performance for distinguishing benign activity. The `dnsteal` class also shows acceptable results, a F1-score of 0.82. While `network_scan` completely fails to be identified, zero across all metrics, this highlights a limitation in the LLM's internal knowledge when not supplemented with similar historical examples.

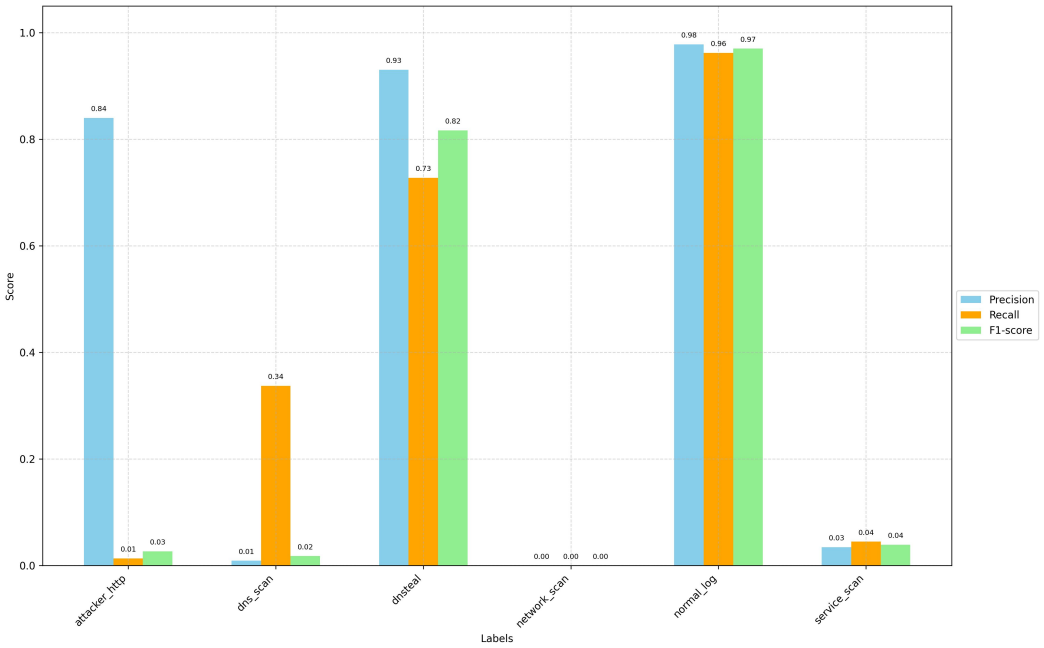


**Figure 5.4:** Precision, Recall, and F1-score per class for the LLM model without RAG.

Moreover, the precision and recall for `attacker_http`, `dns_scan`, and `service_scan` are drastically low, despite `attacker_http` showing high precision (0.84), its recall is just 0.01, resulting in a negligible F1-score of 0.03, meaning it predicts this label accurately but almost never detects it. This suggests over conservative, only labeling when extremely confident. Whereas, for `dns_scan`, the recall is higher (0.34), but precision drops to 0.01, indicating frequent misclassification and poor distinction from similar classes. In the same way, `service_scan` exhibits severe class imbalance or semantic confusion, with all the metrics behind 0.05.

In the end, when a log is not labeled as `normal_log`, a message is created to be sent to the system administrator, here you can see an example with the log, label and the explanation of the LLM:
**Log:** Jan 23 20:42:47 dnsmasq[3468]:reply3x6-.903-
.rTnBmCaADL5hmeSJnHPj9nMmNYSZ3ouMP7-.Xbbx8kPjXpYHH3S27VvpTO
NZZHn*ganSNp-.KgonJrI9NevIh0vnj4XRztgvRVoPb4TRcG-
.oTDf5iqfFpU2yVjCpQdUhJ54Fap5IYsbAd-.payroll_2017.xlsx.email-19.kennedy-
mendoza.info is 195.128.194.168
**Source:** `inet-firewall`
**Label:** `dnsteal`
**Explanation:** The log indicates a DNS reply for a suspicious domain, suggesting an attempt to steal DNS information, likely for malicious purposes such as phishing or malware distribution.

## 5.3 LLM with RAG scenario

As described in 4.2.5, the LLM with RAG scenario combines the LLM's reasoning with contextual information retrieved from similar historical logs via KNN search. This setup is evaluated with the same dataset as the other two scenarios. First, we need to decide the batch_size, as figure 5.5 shows, the query time drops sharply from 0.753 seconds at batch size 1 to just 0.087 seconds at size 25, then stabilizes with minimal gains before rising again beyond size 200. The optimal batch size is 25 because it offers a strong trade-off and fast response without risking memory overload or latency spikes. Now we can start explaining the findings.

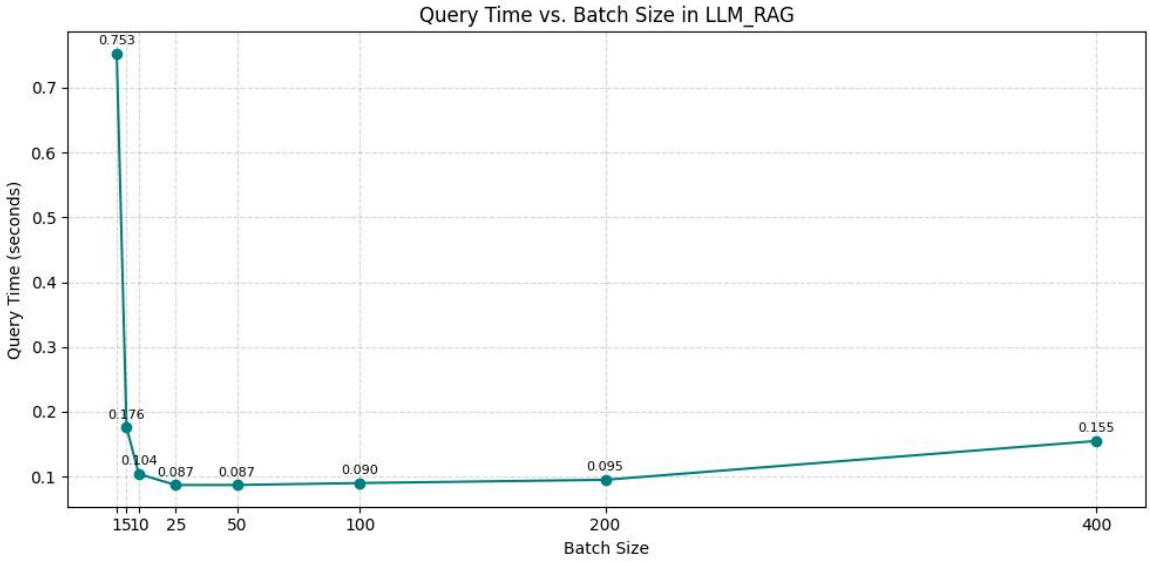


**Figure 5.5:** Query time (s) vs Batch Size

On one hand, the overall metrics for the LLM with RAG model highlight exceptionally strong and consistent performance. First, the accuracy of 98.1% reflects the proportion of correct predictions across all instances. Weighted averages further support this, with a precision of 0.9811 indicating that nearly all predicted labels are correct, and a recall of 0.9812 showing that the model gets almost every true instance right. Furthermore, 0.9807 on the F1 score confirms a strong balance between these two aspects. The closeness of all three metrics suggests the model is not biased toward precision or recall, and instead achieves uniform, high quality performance across the dataset.

On the other hand, precision, recall and F1-Score per class is shown in figure 5.6, indicating consistently strong performance across all labels. `attacker_http`, `dnsteal`,

and `normal_log` all achieve precision and recall values above 0.94, with F1-scores reaching or exceeding 0.95. These high scores suggest the model is highly accurate and complete in identifying these labels, showing strong generalization when context is provided. These three are the most common classes, representing more than 92% of the dataset. Moreover, for `service_scan`, the model performs reliably with precision of 0.87, recall of 0.92, and an F1-score of 0.89, indicating effective recognition of this class despite moderate complexity.

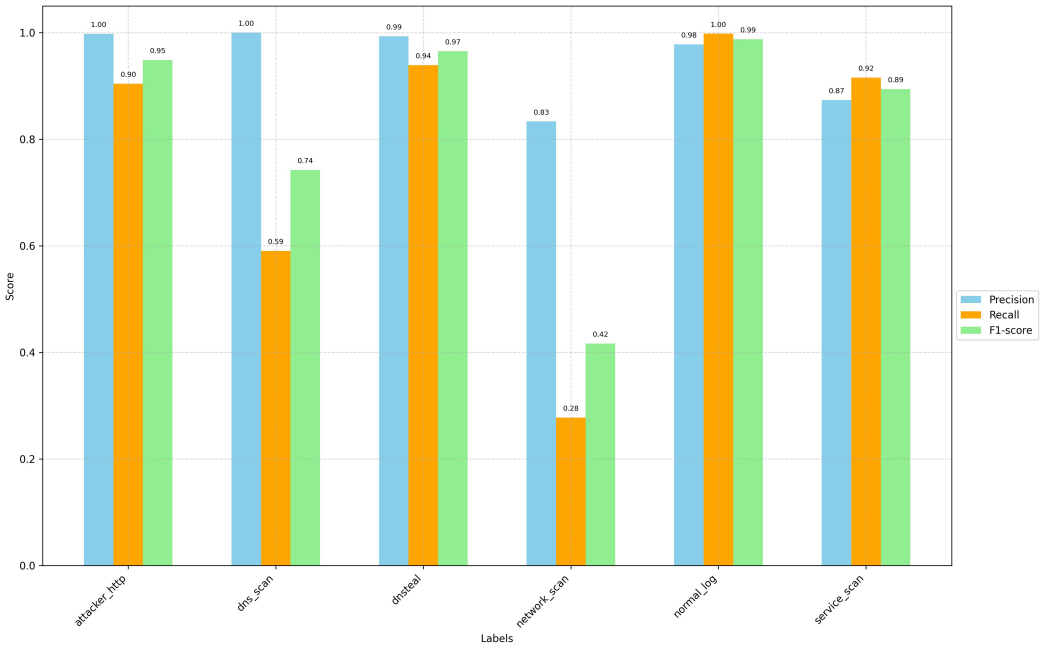


**Figure 5.6:** Precision, Recall, and F1-score per class for the LLM with RAG.

Additionally, the `dns_scan` and `network_scan` classes, which previously showed poor performance in the LLM-only setup, benefit significantly from RAG. The `dns_scan` class achieves perfect precision (1.00), meaning that all predicted instances are correct, but its recall is 0.59, suggesting that many true cases remain undetected. This could result from class imbalance or overfitting to the most common classes. Apart from this, `network_scan` reaches 0.83 precision, indicating accurate predictions, yet recall is low (0.28), leading to an F1-score of 0.42. Therefore, these patterns reveal that while predictions are precise, the model is still conservative in triggering these labels.

As on the LLM scenario, an alert is sent to the system adminitrator if an anomalous log is detected, the alert is the log, source, label or attack type and the explanation created by the LLM. In this case, the LLM explain shortly the attack and the reasons behind the decision, which is based on the RAG. Sometimes, it gives also a similartity

score between the RAG logs and the new classified log.

**Log:** Jan 23 20:42:47 dnsmasq[3468]:reply3x6-.903-.rTnBmCaADL5hmeSJnHPj9nMmNYSZ3ouMP7-.Xbbx8kPjXpYHH3S27VvpTONZZHn*ganSNp-.KgonJrI9NevIh0vnj4XRztgvRVoPb4TRcG-.oTDf5iqfFpU2yVjCpQdUhJ54Fap5IYsbAd-.payroll_2017.xlsx.email-19.kennedy-mendoza.info is 195.128.194.168
**Source:** `inet-firewall`
**Label:** `dnsteal`
**Explanation:** This log is similar to the known 'dnsteal' attack pattern, where an attacker is attempting to steal sensitive information from a DNS server. The log message contains a similar format and IP address, indicating a potential DNS amplification attack. This decision is based on the provided list of logs.

## 5.4 Comparasion between scenarios

Table 5.1 presents a global comparison of KNN, LLM, and LLM with RAG across key evaluation metrics. The LLM model shows the weakest performance, particularly in recall and accuracy (both 0.8802), reflecting limited generalization without external context. KNN performs better with balanced scores around 0.96, but the highest results are achieved by the LLM + RAG model, reaching up to 0.9812 in accuracy and recall. These findings highlight the effectiveness of augmenting language models with useful context. To better understand where each model succeeds or fails, a more detailed analysis will be presented.

| Metric | KNN | LLM | LLM + RAG |
|---|---|---|---|
| Accuracy | 0.9619 | 0.8802 | 0.9812 |
| Precision | 0.9630 | 0.9597 | 0.9811 |
| Recall | 0.9619 | 0.8802 | 0.9812 |
| F1-score | 0.9603 | 0.9047 | 0.9807 |

**Table 5.1:** Comparison of overall metrics across KNN, LLM, and LLM + RAG models.

Firstly, figure 5.7 compares precision per label across models, showing that the LLM with RAG consistently outperforms or matches the others. Notably, RAG boosts precision over normal LLM for all cases and matches KNN in `attacker_http`, `dns_scan` and `dnsteal`, achieving a near perfect precision. Furthermore, It also improves significantly over both models in `network_scan`, while KNN leads slightly in `service_scan`, RAG offers the most consistent high precision, especially for rare or ambiguous categories, confirming its value in enhancing LLM decision confidence. Apart from this, precision on `normal_log` is better than in the KNN model, meaning that the LLM

+ RAG model makes fewer false positive predictions for benign activity, enhancing reliability in real-world deployment by reducing unnecessary alerts.

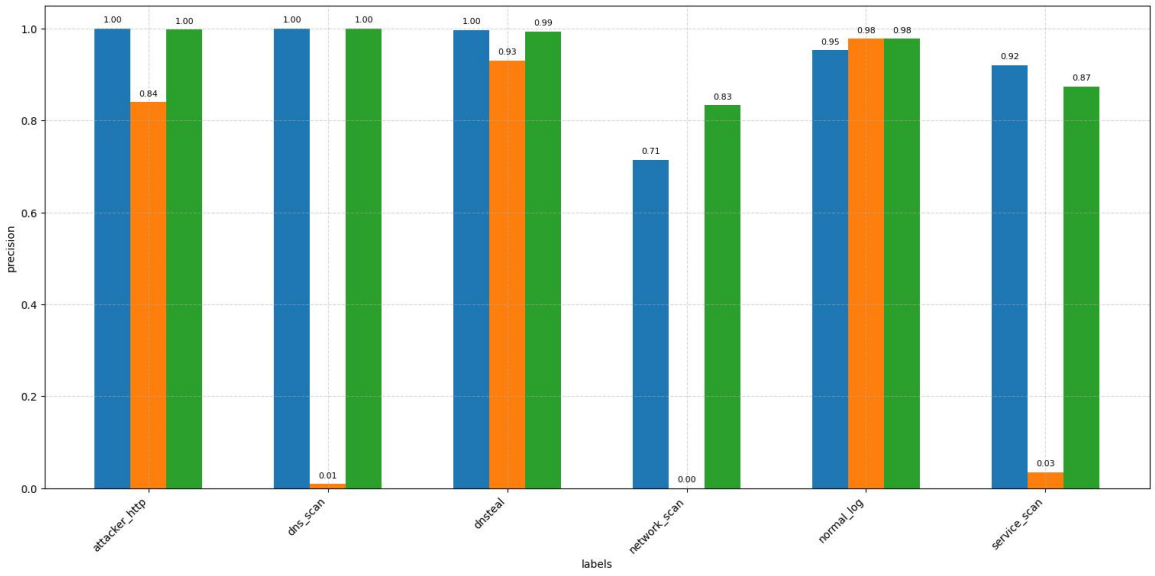


**Figure 5.7:** Precision comparison per class across KNN, LLM, and LLM + RAG models.

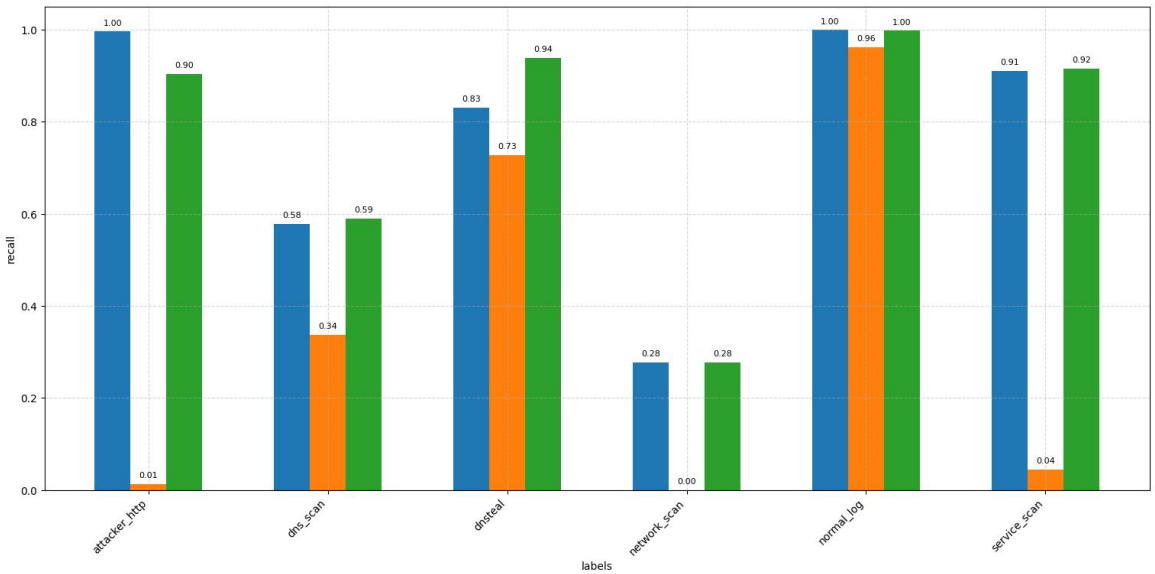


**Figure 5.8:** Recall comparison per class across KNN, LLM, and LLM + RAG models.

Secondly, barplot 5.8 shows the recall per label across models. The LLM + RAG model outperforms LLM significantly in `attacker_http`, `dns_scan` and `dnsteal`,

also matches top performance in `normal_log` and `service_scan`. Althought the recall still low in `network_scan`, it equals KNN on 0.28 and LLM fails. However, in `attacker_http`, KNN still leads with perfect recall. These results confirm that RAG boosts LLM sensitivity for several challenging classes, but KNN alone can outperform it on certain high-confidence labels.

Lastly, F1-scores per label are displayed in figure 5.9. Both the KNN and LLM + RAG models consistently outperform the standalone LLM across all classes. Moreover, KNN and LLM with RAG exhibit nearly identical F1-scores in most labels, reflecting comparable performance. However, KNN slightly surpasses RAG in `attacker_http`, achieving a perfect score. In contrast, LLM + RAG outperforms KNN in `dnsteal` and `normal_log`, which together account for over 90% of the dataset. Therefore, while KNN shows strength in one specific attack detection, the superior performance of RAG in the most frequent classes highlights its greater practical relevance and overall classification robustness.

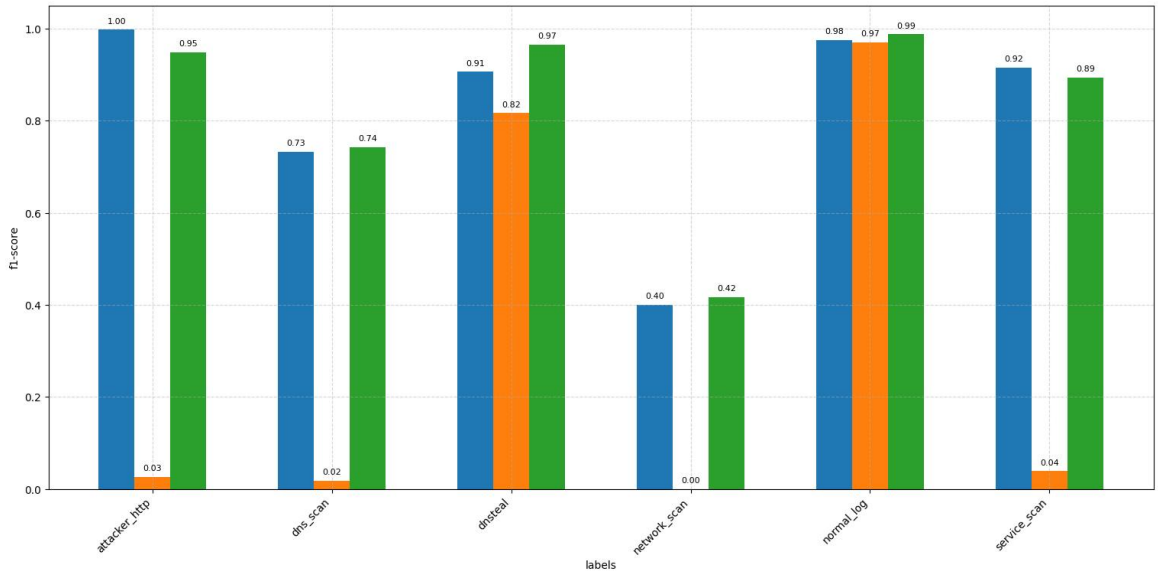


**Figure 5.9:** F1-score comparison per class across KNN, LLM, and LLM + RAG models.

Finally, the table 5.2 shows that while the LLM achieves the lowest query time, both KNN and LLM + RAG require significantly more time, that is mainly because we are using a light LLM and the LLM server is way more powerfull than the machine used to retrieve the logs using KNN as it is described in 4.2.6. However, this additional latency is justified, as it brings substantial performance gains. Therefore, the RAG approach balances speed with enhanced classification quality.

| Model | Query Time (seconds) |
|---|---|
| KNN | 0.105 |
| LLM | 0.013 |
| LLM + RAG | 0.108 |

**Table 5.2:** Query time comparison for KNN, LLM, and LLM + RAG models.

# 6 Conclusions & research direction

## 6.1 Conclusion

This work set out to explore and evaluate the integration of Large Language Models with retrieval-augmented generation techniques for log-based intrusion detection. The main goal was to enhance classification performance by enriching model inputs with similar historical logs, retrieved via a K-Nearest Neighbors search on semantic embeddings. To achieve this, a complete architecture was implemented and tested, including asynchronous evaluation scripts, a deployed LLM instance, and OpenSearch for efficient retrieval and storage. All technical components were integrated successfully, and a detailed experimental evaluation was conducted across three models: KNN, LLM, and LLM with RAG. This setup allowed us to assess their capabilities in handling real log data and detecting a diverse set of network threats.

Based on the evaluation results detailed in Chapter 5, a clear conclusion can be drawn: the LLM + RAG model consistently outperforms both the LLM and KNN baselines. While the LLM alone demonstrates fast inference times, its performance is limited in terms of accuracy, precision and recall, especially for rare or obfuscated attack patterns. KNN delivers strong baseline results using similarity-based classification, but the hybrid LLM + RAG approach offers the most balanced and accurate outcomes, achieving the highest overall accuracy, precision, recall and F1-score, while maintaining practical query times. Furthermore, it surpasses both other models in critical labels such as `dnsteal` and `normal_log`, which represent the majority of the dataset, validating its generalization capabilities and real-world applicability. Additionally, the LLM with RAG is the most accurate predicting extremely rare labels, fewer than 10 samples, making it the most suitable for predicting new attacks.

All things considered, this research provides valuable insight into the potential of hybrid LLM + RAG architectures in the cybersecurity domain. Despite facing challenges such as log imbalance and interpretability of embeddings, the integration of semantic retrieval with LLM reasoning proves highly effective. The findings emphasize the benefit of enriching LLM inputs with contextually similar examples to improve prediction quality. Nevertheless, this line of work is still evolving and there is much room for future development. These open paths are outlined in the next section and form a promising direction for further research. In light of these promising findings,

the preparation of a research article will be considered to report the results presented in this investigation.

## 6.2 Research directions

Based on the results and insights obtained from the evaluation of the proposed architecture, several research directions have emerged that may guide future work in this area. While the integration of retrieval mechanisms with language models has shown considerable improvements in classification performance, important challenges remain. These are some of them:

1. **Improving retrieval strategies in RAG architectures.** While the RAG-enhanced LLM demonstrated strong results, its retrieval mechanism relies heavily on KNN similarity in the embedding space. This approach, although effective, may overlook more nuanced semantic relationships or prioritize irrelevant but similar patterns. Future research could explore the use of hybrid retrieval models that combine dense and sparse representations, or dynamically adjust KNN search parameters based on context. Furthermore, learning-to-rank techniques could be introduced to improve the quality of retrieved logs, ensuring that the most informative examples are passed to the LLM, thus further enhancing classification accuracy and consistency.

2. **Addressing class imbalance in log datasets.** The evaluation revealed that dominant classes like `normal_log` and `dnsteal` significantly skew model performance. This imbalance affects the recall and F1-score of minority classes, which are often the most critical from a security perspective. A valuable research direction is the development of synthetic data generation strategies tailored to underrepresented log classes. Techniques like SMOTE for text, adversarial log generation using generative models, or focused data augmentation during retrieval could help mitigate imbalance.

3. **Real-time deployment and batch optimization.** Although the current system achieves competitive response times, real-time deployment in operational environments introduces new challenges. Batch size, for example, directly impacts system latency and throughput. While batch size 25 was found to be optimal during evaluation, further investigation into dynamic batching strategies based on system load or alert frequency could yield performance gains. Research could also explore model quantization or low-latency deployment frameworks to accelerate LLM inference in production, such as TensorRT or ONNX.

4. **Enhancing explainability of LLM decisions.** Despite generating concise justifications for each classification, the explication of the LLM outputs remains

opaque to system administrators. For operational trust and forensic analysis, enhancing the interpretability of model decisions is essential. Future research could investigate methods for structured reasoning explanations, chain-of-thought prompting, or visualization tools that highlight which tokens or log parts contributed most to the decision. Finally, aligning LLM outputs with known attack patterns can improve both the quality of explanations and the confidence of the analyst.

# Bibliography


[1] Usama Ahmed, Mohammad Nazir, Amna Sarwar, Tariq Ali, El-Hadi M Aggoune, Tariq Shahzad, and Muhammad Adnan Khan. Signature-based intrusion detection using machine learning and deep learning approaches empowered with fuzzy clustering. *Scientific Reports*, 15(1):1726, 2025.

[2] Richard Kimanzi, Peter Kimanga, Dedan Cherori, and Patrick K Gikunda. Deep learning algorithms used in intrusion detection systems–a review. *arXiv preprint arXiv:2402.17020*, 2024.

[3] Mohammed Hassanin and Nour Moustafa. A comprehensive overview of large language models (llms) for cyber defences: Opportunities and directions. *arXiv peprint arXiv:2405.14487*, 2024.

[4] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[5] Paul RB Houssel, Priyanka Singh, Siamak Layeghy, and Marius Portmann. Towards explainable network intrusion detection using large language models. *arXiv preprint arXiv:2408.04342*, 2024.

[6] Liam Daly Manocchio, Siamak Layeghy, Wai Weng Lo, Gayan K Kulatilleke, Mohanad Sarhan, and Marius Portmann. Flowtransformer: A transformer framework for flow-based network intrusion detection systems. *Expert Systems with Applications*, 241:122564, 2024.

[7] Marc Norton. Optimizing pattern matching for intrusion detection. *Sourcefire, Inc., Columbia, MD*, 2004.

[8] Abebe Abeshu Diro and Naveen Chilamkurti. Distributed attack detection scheme using deep learning approach for internet of things. *Future Generation Computer Systems*, 82:761–768, 2018.

[9] Hamid Bostani and Mansour Sheikhan. Hybrid of anomaly-based and specification-based ids for internet of things using unsupervised opf based on mapreduce approach. *Computer Communications*, 98:52–71, 2017.

[10] Yakub Kayode Saheed, Aremu Idris Abiodun, Sanjay Misra, Monica Kristiansen Holone, and Ricardo Colomo-Palacios. A machine learning-based intrusion detection for detecting internet of things network attacks. *Alexandria Engineering Journal*, 61(12):9395–9409, 2022.

[11] Zarrin Tasnim Sworna, Zahra Mousavi, and Muhammad Ali Babar. Nlp methods in host-based intrusion detection systems: A systematic review and future directions. *Journal of Network and Computer Applications*, 220:103761, 2023.

[12] Oscar G. Lira, Alberto Marroquin, and Marco Antonio To. Harnessing the advanced capabilities of llm for adaptive intrusion detection systems. In *International Conference on Advanced Information Networking and Applications*, pages 453–464. Springer, 2024.

[13] Michael Guastalla, Yiyi Li, Arvin Hekmati, and Bhaskar Krishnamachari. Application of large language models to ddos attack detection. In *International Conference on Security and Privacy in Cyber-Physical Systems and Smart Vehicles*, pages 83–99. Springer, 2023.

[14] Seshu Bhavani Mallampati and Hari Seetha. Enhancing intrusion detection with explainable ai: A transparent approach to network security. *Cybernetics and Information Technologies*, 24(1):98–117, 2024.

[15] Yanjie Li, Zhen Xiang, Nathaniel D Bastian, Dawn Song, and Bo Li. Ids-agent: An llm agent for explainable intrusion detection in iot networks. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

[16] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[17] Max Landauer, Florian Skopik, Maximilian Frank, Wolfgang Hotwagner, Markus Wurzenberger, and Andreas Rauber. Maintainable log datasets for evaluation of intrusion detection systems. *IEEE Transactions on Dependable and Secure Computing*, 20(4):3466–3482, 2022.

[18] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. Have it your way: Generating customized log datasets with a model-driven simulation testbed. *IEEE Transactions on Reliability*, 70(1):402–415, 2020.

[19] Xingfang Wu, Heng Li, and Foutse Khomh. What information contributes to log-based anomaly detection? insights from a configurable transformer-based approach. *arXiv preprint arXiv:2409.20503*, 2024.

[20] Metrics and scoring: quantifying the quality of predictions. URL `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html`.

[21] VVRPV Jyothsna, Rama Prasad, and K Munivara Prasad. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35, 2011.

[22] Yihua Liao and V Rao Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & security*, 21(5):439–448, 2002.

[23] Faheem Ullah, Matthew Edwards, Rajiv Ramdhany, Ruzanna Chitchyan, M Ali Babar, and Awais Rashid. Data exfiltration: A review of external attack vectors and countermeasures. *Journal of Network and Computer Applications*, 101:18–54, 2018.

[24] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009.

[25] Md Liakat Ali, Kutub Thakur, Suzanna Schmeelk, Joan Debello, and Denise Dragos. Deep learning vs. machine learning for intrusion detection in computer networks: A comparative study. *Applied Sciences*, 15(4):1903, 2025.

[26] Michael Guastalla, Yiyi Li, Arvin Hekmati, and Bhaskar Krishnamachari. Application of large language models to ddos attack detection. In *International Conference on Security and Privacy in Cyber-Physical Systems and Smart Vehicles*, pages 83–99. Springer, 2023.

[27] Albara Awajan. A novel deep learning-based intrusion detection system for iot networks. *Computers*, 12(2):34, 2023.

[28] Tarek Ali and Panos Kostakos. Huntgpt: Integrating machine learning-based anomaly detection and explainable ai with large language models (llms), 2023.

[29] Amit Pandey and Achin Jain. Comparative analysis of knn algorithm using various normalization techniques. *International Journal of Computer Network and Information Security*, 10(11):36, 2017.

[30] VB Prasath, HAA Alfeilat, O Lasassmeh, and ABA Hassanat. Distance and similarity measures effect on the performance of k-nearest neighbor classifier–a. *arXiv preprint arXiv:1708.04321*, 2017.

[31] Anna Huang et al. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, volume 4, pages 9–56, 2008.