Declaration for understanding

I, Nikola Georgiev (student's name, and ID) ID# 20019133?

the requirements, conditions, and consequences of the above regulations in developing the COS or INF Senior Project

in Fall Semester of 2025-26 Academic Year.

Student Signature: _____ Supervisor Signature: _____

Name: Nikola Georgiev Name: ZACHARY HUTCHINSON

Date: 05.12.2025 Date: 05.12.2025

Bulgarian Diploma Thesis

DDoS Detection System

Nikola Georgiev, ID: 200191333

Student: Nikola Georgiev , **Date:** 05.12.2025

signature

Supervisor: , **Date:** 05.12.2025

signature

Department of Computer Science, AUBG

Blagoevgrad, 2025

0. Table of Contents

1. Introduction:

Distributed Denial of Service (DDoS) attacks are one of the most severe and persistent threats to the stability, reliability, and availability of online systems. These attacks overwhelm network infrastructure by flooding targeted servers or services with excessive traffic, resulting in significant service degradation or complete unavailability. As a System Engineer at A1 Telekom Bulgaria, I witnessed firsthand how these attacks can degrade services and make large-scale network infrastructures unusable. This project was conceived as a practical and technical response to that problem - using data analysis and classification techniques to identify and mitigate DDoS attacks in real time.

The project focuses on the development of a C++ library designed for DDoS detection and classification. The library is then used with a two-service FastAPI micro service architecture, consisting of two primary components: one that is responsible for data collection and second that is for network analysis.

The motivation behind this project stems from real-world challenges encountered in telecommunications infrastructure. DDoS attacks are not only increasing in frequency but also evolving in sophistication. Attackers often use botnets of compromised IoT devices or amplification techniques to launch multi-vector attacks that are difficult to detect through traditional signature-based systems.

Also as a person who has a networking knowledge and software development knowledge I can propose a solution to the community that is easy to integrate with their current system in order to detect ongoing malicious activities

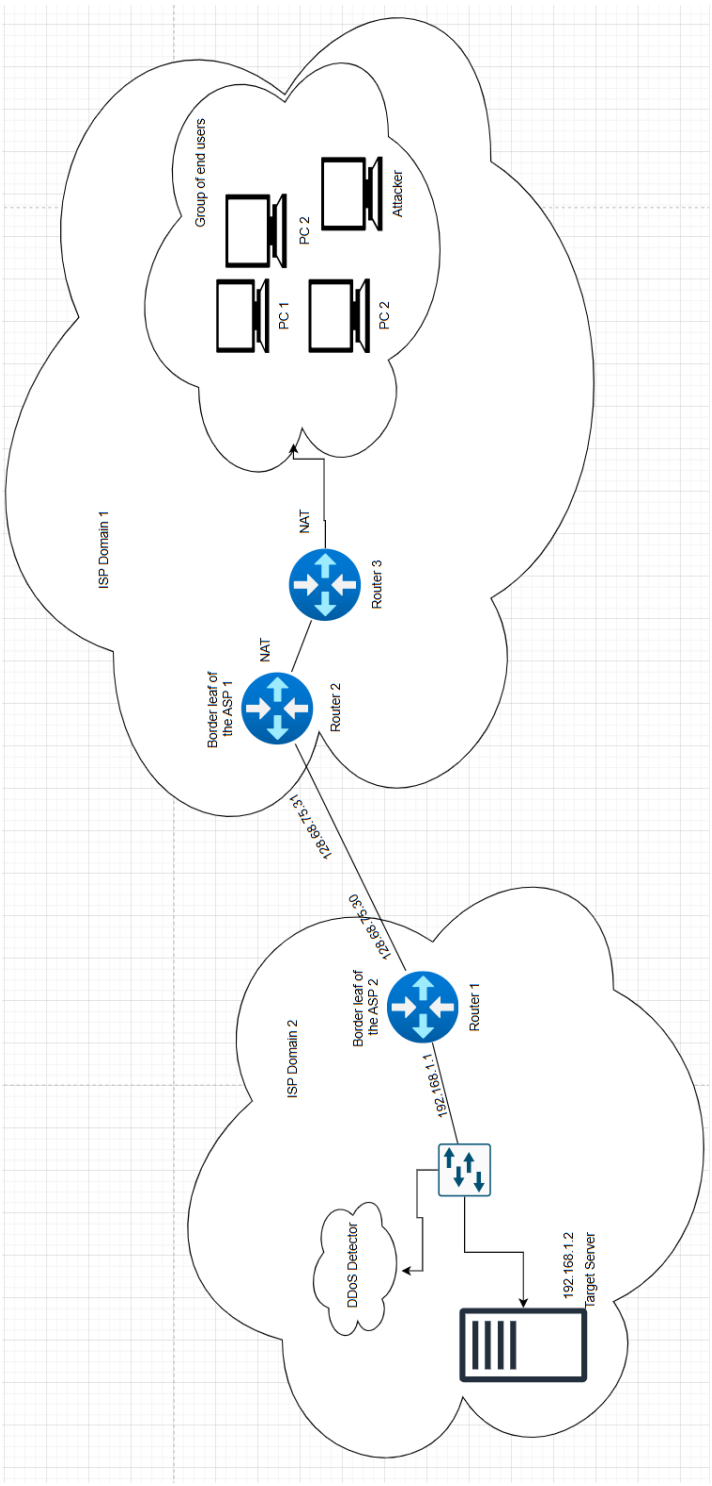


Figure 1: Network architecture overview

Figure 1 is illustration of how 2 Internet Service Providers (ISP) communicate with each other we can see a cluster of users in the domain of IPS 1 they are behind Network Address Translation (NAT) interface of Router 3. This means that all the requests we

capture in our ISP 2 domain will be with source IP of 128.68.75.31, no matter from which user the request is coming. The proposed network setup follows a server with top-of-rack switch (ToR) architecture, commonly deployed in enterprise data centers. The target server (IP: 192.168.1.2) is directly connected to the ToR switch server this approach will enable us to connect our DDoS classifier instance directly to the switch and by tunneling traffic to its interface in order to capture all the request. Tunneling is a setting on the switch which enables certain MAC address to listen to all the traffic on the switch no matter if it is the destination mac or not.

2.Specification of the Software Requirements and their Analysis

As mentioned because the requests are coming from behind A NAT interface we will not be able to distinguish different IP's which means that we will have to gather second layer data. We will need to listen on ports: 80, 443 via TCP in order to get traffic data.

2.1 Requirements and their Analysis

2.1.1 Start/Stop Capture

The *Start/Stop Capture* functionality represents the first and operational feature of the DDoS detection system. It allows the system administrator or security engineer to control packet capture operations through RESTful endpoints exposed by the FastAPI-based Network Sniffer micro service. The endpoint /open_socket initiates the packet capture process by specifying the network interface to listen on only TCP packets targeting ports 80 and 443 (HTTPS, HTTPS)

The packet capture process is done at the kernel level to collect real time network traffic data. Each packet is timestamped and forwarded to the feature computation module for further aggregation. The /close_socket endpoint stops the capture process gracefully, ensuring that any packets stored in the buffer of the machine are flushed and that system resources such as sockets and file descriptors are properly released. This mechanism provides flexible management of data acquisition while preventing resource exhaustion and ensuring accurate data gathering

2.1.2 Feature Computation

Feature computation is at the core of the data processing pipeline. The captured packets are aggregated into 5 second time intervals. For each active IP address, a set of numerical and statistical features is extracted to represent the behavioral characteristics of the traffic during that time frame. Typical features include packet rate, byte rate, ratio of TCP flags (SYN, ACK). The system maintains a rolling window which is allowing temporal context for the machine learning models to evaluate ongoing trends. This design balances responsiveness and stability - short enough to detect attacks quickly, but long enough to minimize false positives from short spikes in traffic.

Feature computation runs as a background task, automatically triggered at each bucket interval. It ensures synchronization between the capture timestamp and the analyzer's feature timeline. Any missing packets or dropped intervals are logged for diagnostic purposes.

2.1.3 Feature Delivery API

The Feature Delivery API is the interface between the data collection micro service and external analytical tools. The endpoint /get_data/{ip} enables the retrieval of the latest N feature vectors associated with a particular destination IP address. This API serves as the bridge between the raw packet data and the machine learning inference engine.

It ensures that the data which returned is normalized and timestamp aligned. The response is formatted in JSON, containing each feature name and value, time range, and version metadata for traceability. Engineers and downstream systems can use this endpoint to visualize traffic behavior trends or feed external monitoring dashboards.

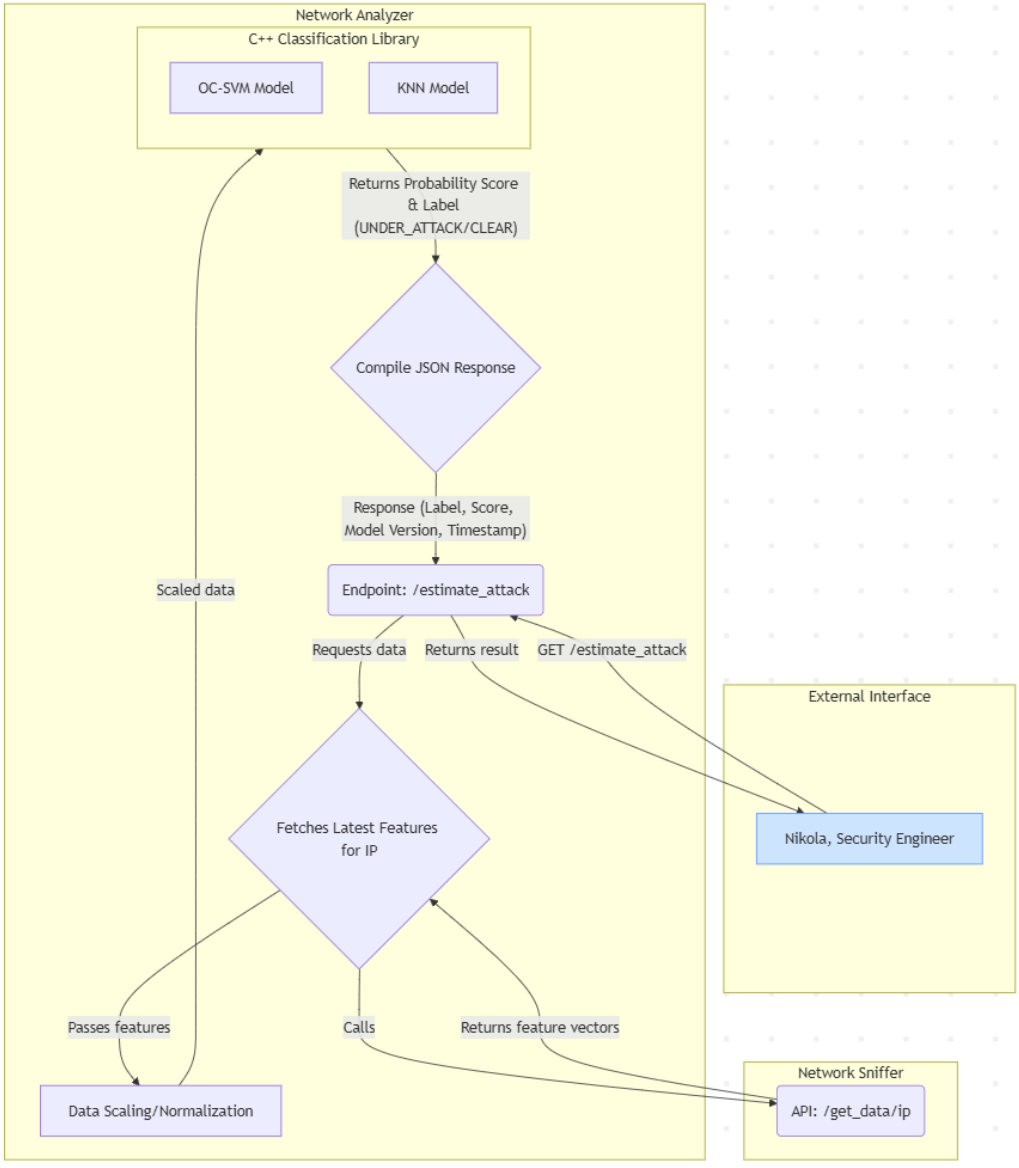


Figure 2 Feature Delivery Process

2.1.4 Analyze on Demand

The Analyze on Demand functionality allows an engineer or automated monitoring system to query the analyzer micro service for an immediate DDoS status evaluation. By calling /estimate_attack?ip=…, the analyzer fetches the latest set of feature vectors for the target IP and passes them to the C++ classification library through the pybind11 interface.

The library executes the One-Class Support Vector Machine (OC-SVM) and K-Nearest Neighbors (KNN) models, returning a predicted label. These models have been trained using previous attack-labeled data, the output includes the decision label (UNDER_ATTACK or CLEAR).

The service compiles these results into a structured JSON response which, including model version and timestamp, enabling traceability and auditability of the decision-making process.

2.1.5 Continuous monitoring

Continuous monitoring ensures that DDoS detection occurs automatically and in real-time. Rather than requiring manual requests, the analyzer micro service executes periodic classification cycles synchronized with the feature computation intervals (every 5 seconds). Each monitored IP is analyzed automatically as new feature data becomes available.

The system maintains a state for each IP, storing its current classification (e.g., CLEAR, UNDER_ATTACK). When a state change is detected, potentially alert or some different system that is connected can be triggered. This allows for immediate response and correlation across network management systems. Continuous monitoring transforms the DDoS detector into an autonomous early warning system, capable of detecting emerging attacks with minimal human intervention.

2.1.6 Get Current State

This functionality provides an on-demand summary of the current security status of any monitored IP. The endpoint /ongoing_attack/{ip} returns a structured response containing the latest classification label, the detection confidence score, timestamp of the last analysis,

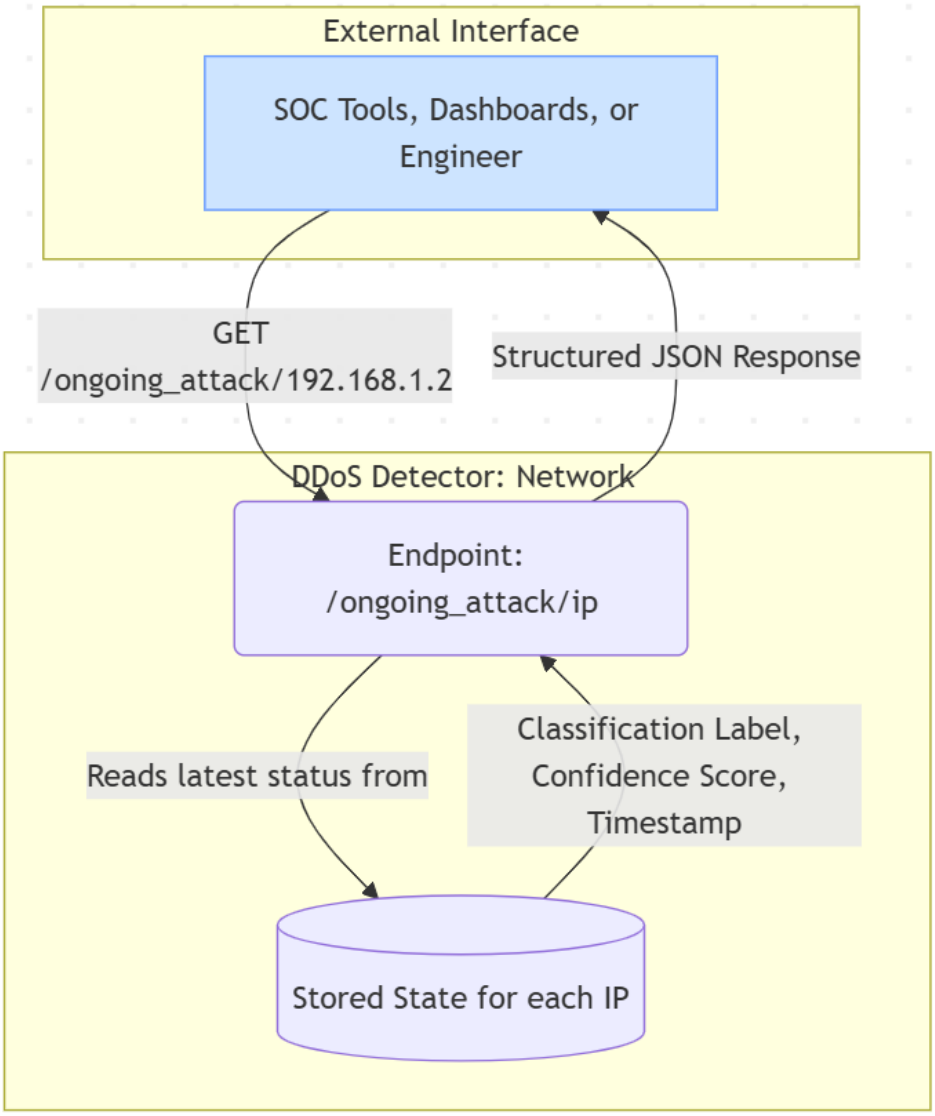


Figure 3 Get Current state Overview

This feature is particularly useful for integrating with external network monitoring dashboards, automated mitigation systems, or SOC tools. It provides a clear and machine-readable status snapshot, enabling operators to quickly verify whether an IP is currently under attack or behaving normally.

2.1.7 Model Management

The Model Management functionality guarantees that the machine learning models used for DDoS detection (OC-SVM and KNN) are always current and consistent. Upon startup, the analyzer micro service loads both models from disk through the in the C++ library. Such option helps to load pre-trained models start up time of the service and update the model's settings without the need of restarting. This dynamic model management enables rapid iteration and retraining without redeploying the entire system—an essential capability in evolving threat environments.

2.1.8 Data Scaling

Before feeding network traffic features into the OC-SVM and KNN models, it is essential that all input variables are transformed into a normalized scale to prevent feature magnitude bias. This requirement specifies that each computed feature - such as packet rate, byte rate, SYN/ACK ratios - must undergo standardization using parameters (mean and standard deviation) derived from the training dataset. During real - time analysis, the system applies the same scaling coefficients to ensure consistency between training and inference distributions.

The scaling ensures that no individual metric dominates the model's decision due to numerical range differences. These normalization parameters are stored in a configuration file (e.g.config.txt) within the Network Analyzer micro service and are loaded automatically by the analyzer at startup.

2.2 Constraints

2.2.1 Micro Service Containerization

Due to compatibility limitations between VMware Workstation and containerization technologies such as Docker, WSL2, and Hyper-V, the system must be built and executed natively on Windows 10 without relying on any form of containerization. Docker and similar virtualization layers must not be used or required for deployment, as VMware Workstation is essential for running the GNS3 network simulator that is used in the testing and validation environment.

3. Design of the software solution

3.1 Project Architecture Overview

The design of the DDoS detection application is shaped by several key contextual and operational constraints. Since all traffic originates from behind a NAT, it is impossible to distinguish unique public IP sources. Therefore, the system captures and analyzes packet-level attributes like TCP flags, bytes per second and connection diversity metrics (SYN count,). The entire system is implemented to run natively on Windows 10 without the use of Docker, WSL2, or other containerization tools, since compatibility with VMware Workstation is required for integration with the GNS3 network simulator.

Services such as packet capture, feature computation, and attack analysis are deployed as independent micro service services. Network traffic is captured through the Npcap library, for low-level packet access. To support near real-time detection, all components process data in 5-second intervals. The separation of data capture from analysis is ensuring that packet acquisition continues even if the analytical subsystem restarts or fails This way a persistent store can be added.

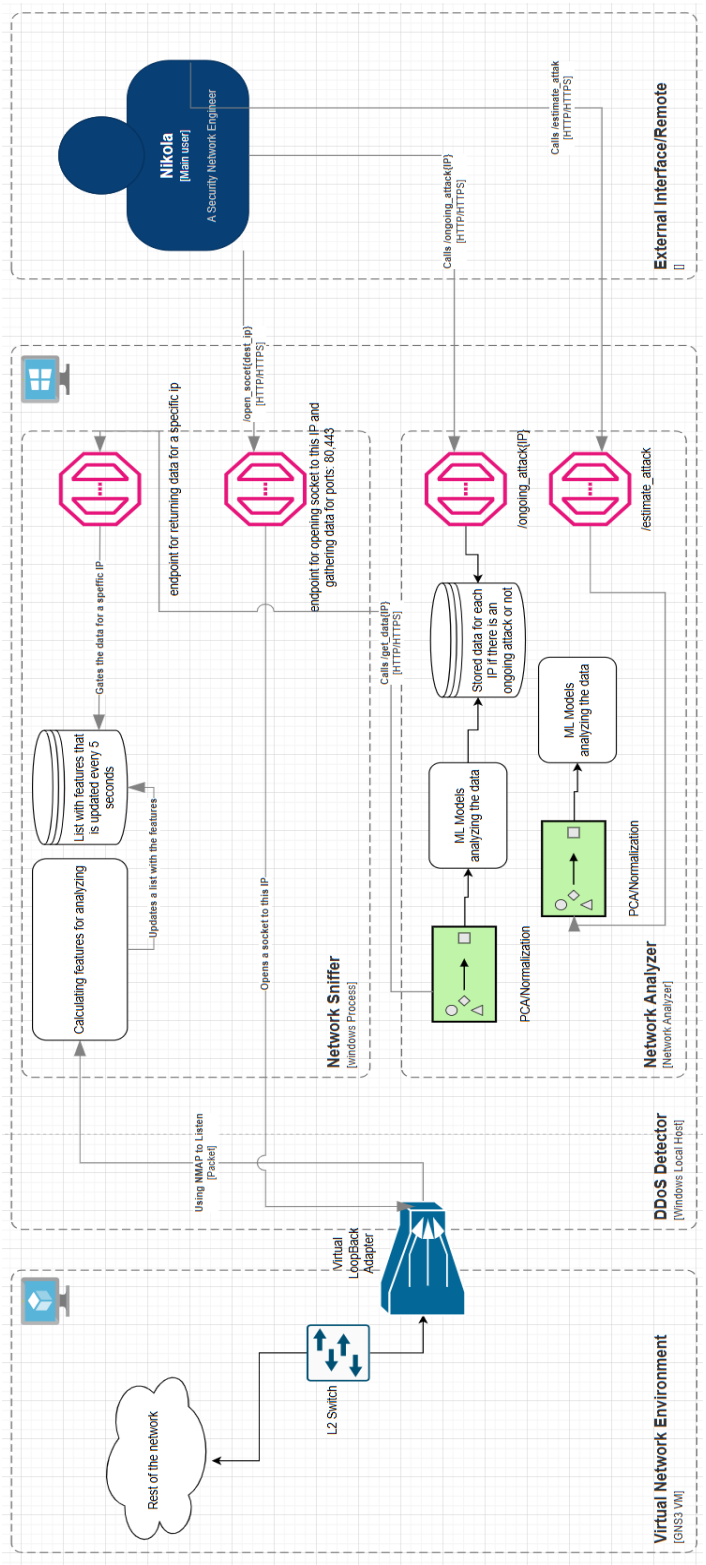


Figure 4 Micro services Architecture Overview

z

The microservice architecture shown in the diagrams is designed to meet the functional requirements of a DDoS detection system by integrating modular components that handle data collection, processing, and analysis in real time. The environment is divided into three main layers: the Virtual Network Environment that is going to be used for testing a real DDoS attack, the Network Sniffer micro service, and the DDoS Analyzer micro service, all orchestrated through RESTful APIs.

In the Virtual Network Environment, traffic generated from the simulated GNS3 setup is directed toward a target server (192.168.1.2). This setup replicates a real-world Internet topology where an attacker and legitimate users reside in different ISP domains. A SPAN or mirrored port forwards copies of network packets to the Network Sniffer microservice for inspection. This forms the entry point for traffic monitoring and DDoS detection.

The Network Sniffer micro service serves as the system's data acquisition and preprocessing layer. Built using FastAPI, it provides two critical operational endpoints—/open_socket and /close_socket—that allow administrators to start and stop packet capture dynamically. The capture process operates at the kernel level, filtering only TCP packets destined for ports 80 and 443 (HTTP and HTTPS), ensuring efficient monitoring of web traffic. When started, packets are timestamped and sent to the feature computation module, and when stopped, any buffered data is flushed to prevent resource leakage.

Feature computation forms the analytical core of the Network Sniffer service. Captured packets are aggregated into five-second intervals, during which numerical and statistical features such as packet rate, byte rate, and TCP flag ratios (SYN/ACK) are calculated per active IP address. This rolling window is allowing the system to retain

temporal context - short enough to respond quickly to attacks, yet stable enough to avoid false positives caused by transient spikes or momentum packet jiters. Synchronization between capture timestamps and feature intervals ensures data accuracy.

The Feature Delivery API bridges the Network Sniffer and external systems such as the DDoS Analyzer. Through the /get_data/{ip} endpoint, the API provides normalized, timestamp-aligned feature vectors in JSON format, complete with feature names, time ranges, and version metadata. This enables interoperability with visualization dashboards, monitoring tools, and machine learning modules that rely on consistent data for analysis and decision-making.

The DDoS Analyzer microservice is responsible for detecting and classifying potential attacks. It can perform both on-demand and continuous analysis. When the /estimate_attack?ip=… endpoint is called, the analyzer fetches recent features for the target IP, scales them using stored normalization parameters, and forwards them to a C++ classification library via a pybind11 interface. The library executes One-Class Support Vector Machine (OC-SVM) and K-Nearest Neighbors (KNN) algorithms, which have been trained on labeled historical attack data. The output includes a classification label (either UNDER_ATTACK or CLEAR), along with timestamps and model version identifiers for auditability.

For automated protection, the analyzer includes a continuous monitoring mode, running every five seconds in sync with feature generation. Each monitored IP (Target Host) is automatically analyzed, as new data becomes available, maintaining a continuous state that tracks whether an address is currently under attack or behaving normally. When a state change occurs, alerts or external triggers can be sent to integrated

systems such as security operation centers or mitigation services, enabling rapid incident response.

The Get Current State endpoint (/ongoing_attack/{ip}) provides a quick snapshot of any monitored IP's latest classification, including the detection confidence score and timestamp of the last analysis. This supports easy integration with SOC dashboards or automated mitigation platforms, offering clear visibility into the network's current security posture.

Model management is another vital aspect of the system. Upon startup, the analyzer loads its machine learning models (OC-SVM and KNN) directly from disk through the C++ library. This allows for dynamic model updates without restarting the service, enabling frequent retraining or tuning as new attack data becomes available. Alongside the models, a configuration file (e.g., config.txt) stores the normalized coefficients used for feature scaling, ensuring the same behavior between training and real-time inference.

Finally, the deployment constraint requires the system to run natively on Windows 10 without containerization. This design choice arises from compatibility issues between VMware Workstation - used to host the GNS3 simulation - and container technologies such as Docker, WSL2, or Hyper-V. As a result, all components, including FastAPI services and the C++ inference engine, are deployed directly on Windows. The sniffer interacts with the network interface through native Windows drivers, while the analyzer runs its Python and C++ processes within the same environment.

Overall, the micro service architecture effectively meets the DDoS detection system's functional requirements by combination of modular approach, scalability, and real-time analytics. The Network Sniffer ensures accurate and efficient packet collection and feature, the Analyzer provides both manual and automated attack detection using advanced models, and the REST APIs offer interoperability and traceability. Together, they form a reliable, self-contained detection framework that operates seamlessly in a virtualized Windows-based testing environment.

3.2 Class Architecture

The system is organized around a hierarchical structure that separates general algorithmic responsibilities from the specific logic required by individual learning methods. At the foundation of this structure is an abstract component that defines the essential capabilities expected of any model within the framework. It maintains information about the input features, the target variable, and the size of the training data. It also manages memory on the GPU for storing training inputs and targets and provides optional preprocessing mechanisms such as feature scaling and dimensionality reduction. This foundational component establishes a uniform interface for training and prediction that all specialized models must follow, ensuring consistency across the framework.

Two concrete learning components extend this foundation, each implementing a distinct method while relying on the shared services provided at the base. One represents a nearest-neighbour approach. It stores a parameter that determines how many neighbours are considered when making a prediction and uses the common data preparation mechanisms inherited from the base. Its training phase relies primarily on setting up the data properly, while its prediction phase involves identifying the closest stored examples to each new sample. The actual computations involved in this process

are delegated to a dedicated helper that runs on the GPU, enabling efficient distance calculation and neighbour selection.

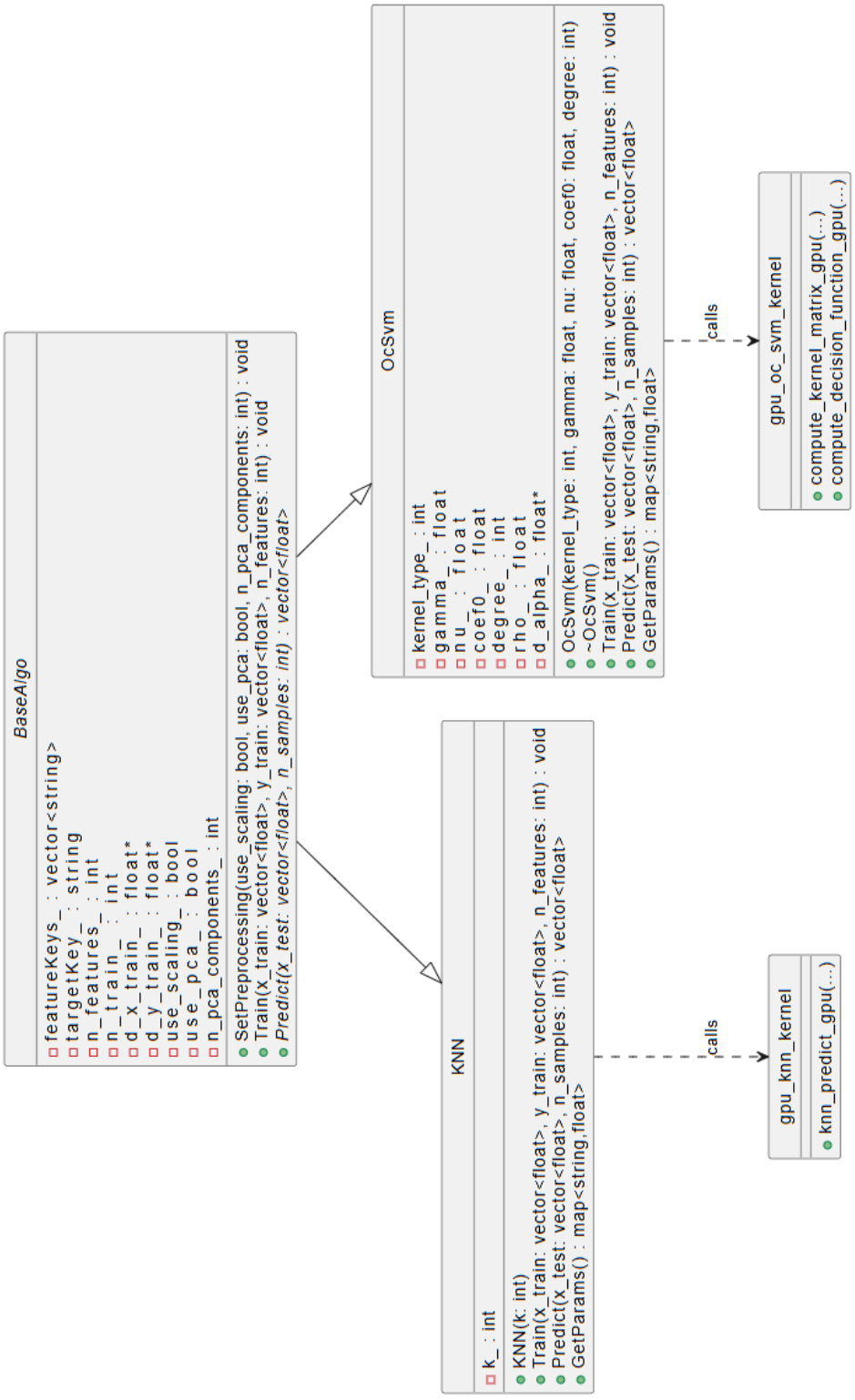


Figure 5 UML Class Diagram

The second specialized component implements a one-class classification method aimed at distinguishing normal samples from anomalous ones. It introduces a set of additional parameters that define the behaviour of the underlying decision function, such as kernel type, kernel coefficients, and learned values associated with the resulting boundary. Like the nearest-neighbour component, it uses the shared capabilities provided by the foundational layer to handle input data and apply any configured preprocessing. During training, it constructs the decision boundary using a helper that performs kernel computation and optimization on the GPU. During prediction, it again relies on this helper to evaluate new samples against the learned boundary.

Supporting these two learning methods are two GPU-based helper modules. Although they are represented independently, they do not participate in the class hierarchy. Instead, they serve as computational engines that the learning components invoke during training and prediction. One provides procedures for computing distances and neighbour relationships for the nearest-neighbour method, while the other computes kernel matrices and decision function values required by the one-class classifier. Their presence allows the main learning components to offload intensive numerical operations to the GPU while maintaining a clear separation between high-level algorithmic structure and low-level computational routines.

3.3 Implemented Algorithms

Anomaly detection plays a central role in modern cybersecurity, especially when it comes to identifying Distributed Denial of Service (DDoS) attacks. These attacks work by overwhelming a target—typically a server or network—with an enormous volume of traffic originating from many different sources. What makes DDoS attacks particularly challenging is that they often begin subtly, blending in with regular network activity before ramping up to damaging levels. Because of this, the ability to detect even small deviations from normal network behavior can make the difference between early mitigation and a full-blown service outage.

To tackle this challenge, machine learning algorithms that model normal network behavior—without relying on large datasets of labeled attacks—are extremely valuable. This is where k-Nearest Neighbors (k-NN) and One-Class Support Vector Machines (OC-SVM) come into play. Both are well suited for anomaly detection, but they approach the problem from fundamentally different angles, making them excellent candidates for benchmarking against each other.

The k-NN algorithm offers a very intuitive approach based on local density. It looks at how close a given network observation is to its nearest neighbors in historical traffic. Under normal conditions, traffic tends to form clusters: similar packet rates, flow sizes, connection durations, and so on. When a point appears far from these clusters—meaning its k closest neighbors are unusually distant—it is flagged as suspicious. This makes k-NN especially effective at detecting irregular traffic spikes or unusual connection patterns that may hint at a DDoS attack. Importantly, k-NN does not assume anything about the

data distribution, which is ideal in networks where traffic patterns can vary widely and unpredictably.

In contrast, the OC-SVM takes a more global and mathematically formal approach. Instead of focusing on local relationships, it tries to learn the overall shape of what “normal” traffic looks like. By using kernel functions, it can model complex patterns and create a boundary that tightly encloses the normal data. Anything falling outside this learned boundary is considered an anomaly. This makes OC-SVM particularly powerful when labeled data is limited—a common situation in cybersecurity—because it can generalize well from only normal traffic samples. It also tends to be more robust to small fluctuations and noise, which helps reduce false alarms in busy or volatile network environments.

Benchmarking k-NN and OC-SVM side by side provides valuable insight into the trade-offs between local sensitivity and global generalization. k-NN is better at spotting sudden, localized deviations, such as an abrupt spike in packet size or a surge of connections from a single host—typical signs of early-stage DDoS activity. OC-SVM, on the other hand, excels at understanding the broader structure of network traffic, making it more resilient to noise and better at detecting slow-burning or distributed attacks that subtly shift overall traffic characteristics.

By comparing these two methods, we gain a deeper understanding of how different modeling assumptions impact detection performance under various network conditions. This ensures a more complete assessment of anomaly detection strategies and helps identify which approach—or combination of approaches—is best suited for real-world DDoS detection.

3.3.1 One class Support Vector Machines (Oc-SVM)

The One-Class Support Vector Machine (OC-SVM) is a powerful method for anomaly detection, especially in situations where I only have examples of what “normal” looks like. Instead of relying on labeled anomalies—which are often rare, expensive to obtain, or simply unknown—the OC-SVM tries to learn the shape of the normal data distribution and then determines whether new observations fit inside that shape or fall outside of it.

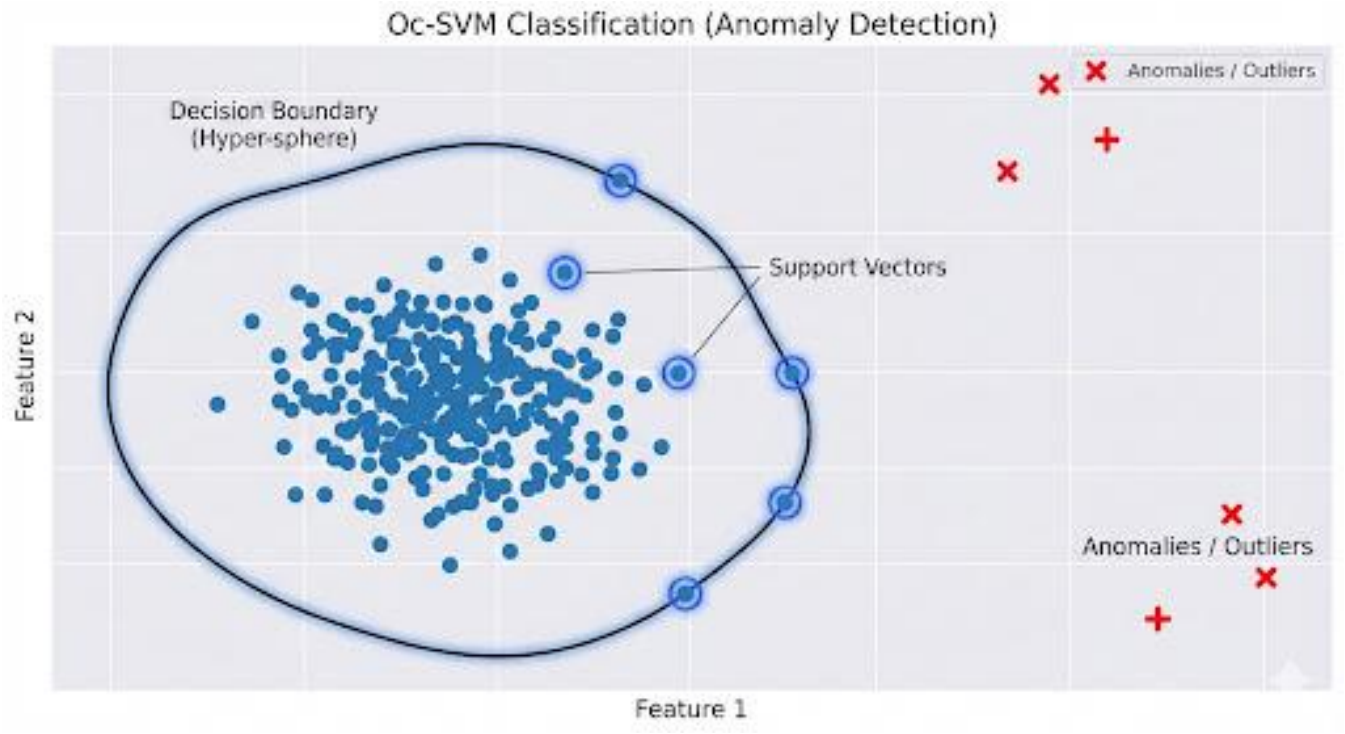


Figure 6 Oc-SVM Classification

The idea behind OC-SVM is quite elegant. First, it uses a kernel function to map the data into a high-dimensional feature space. This is done so the algorithm can separate the normal data from the origin with as simple a boundary as possible. In this transformed

space, the OC-SVM tries to draw a surface that encloses most of the training points. When the algorithm is done, it has essentially learned a function that outputs positive scores for points that fall inside the learned region (i.e., points that look normal) and negative scores for points that lie outside (potential anomalies).

Training the model involves solving a quadratic optimization problem, where the goal is to push the boundary outward while still keeping it tight enough around the data. A key part of this process is the hyperparameter v (nu). This parameter acts as a knob that determines how strict or flexible the boundary should be. A smaller v means the model assumes very few anomalies exist in the training data and tries to tightly wrap the boundary around them. A larger v allows more points to lie outside the boundary, making the model more tolerant of noise or variation in the training data

3.3.2 Kernel Functions Used for Oc-SVM

3.3.2.1 Linear Kernel

The linear kernel is the most straightforward kernel function used in a One-Class SVM. It is simply the dot product of two vectors, meaning it measures similarity based on how aligned two points are in the original feature space. Because no nonlinear transformation is applied, the One-Class SVM with a linear kernel attempts to separate the normal data from the origin using a single flat hyperplane. In other words, the algorithm assumes that normal data lies roughly on one side of a straight boundary while anomalies fall outside that region. This simplicity makes the linear kernel computationally efficient and easy to interpret.

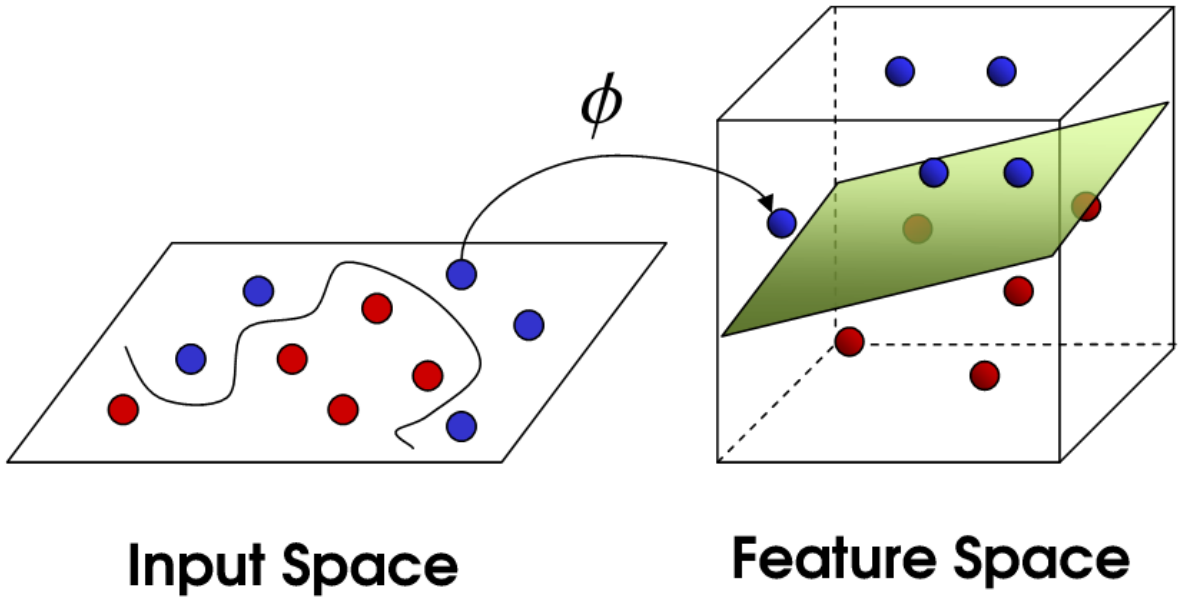


Figure 7 Linear Kernel Function

The linear kernel tends to work best when the distribution of normal data is approximately convex and does not require curved or complex boundaries to enclose it. If the normal region can be reasonably captured by a flat separating surface, then a linear kernel is not only adequate but often ideal. This is especially true in high-dimensional, sparse datasets, such as text representations (TF-IDF vectors, bag-of-words models) or certain types of document and log data. In such cases, the high dimensionality often makes the data more naturally linearly separable, and adding nonlinearity may not provide any benefit.

However, the linear kernel's simplicity is also its most significant limitation. It has no ability to model curved or irregular shapes in the data distribution. If the boundary that separates normal from abnormal data is inherently nonlinear—such as data forming clusters, rings, spirals, or other complex manifolds—the linear kernel can easily underfit. It will impose a flat hyperplane where a flexible boundary is needed, causing it to misclassify many valid normal points as anomalies or fail to detect outliers that lie in low-density regions. In such cases, more expressive kernels like the RBF or polynomial kernel are usually preferred because they can capture nonlinear patterns and adapt to the true geometry of the dataset.

3.3.2.2 Poly Kernel

The polynomial kernel adds a layer of expressive power by taking into account not only the original features but also their interactions. Instead of relying solely on the linear relationships in the data, it implicitly expands the feature space to include combinations such as squared features, cross-products between different features, and higher-order terms depending on the degree of the polynomial. This expansion allows the One-Class SVM to draw curved, flexible boundaries around the normal data, making it possible to model more intricate patterns of normality that a linear kernel would completely miss.

In the context of One-Class SVM, the polynomial kernel is especially helpful when the normal behavior in the dataset is shaped by feature interactions. For example, if the relationship that defines what is “normal” depends on both feature A and feature B changing together in a specific way—say, one increasing as the other increases—the polynomial kernel can capture this interaction naturally. Quadratic or cubic boundaries, which arise from second- or third-degree polynomials, can outline regions of the data that are curved, elliptical, or shaped by complex dependencies between features. This makes the polynomial kernel a solid choice when the structure of normality is not well approximated by a simple straight line or flat hyperplane.

However, with this flexibility comes a significant trade-off. As the degree of the polynomial grows, the decision boundary becomes increasingly complex. A high-degree polynomial can fit extremely detailed shapes in the data, but it can also become overly responsive to small fluctuations or noise. This may cause the model to overfit, creating a boundary that wraps tightly around the training points but fails to generalize to new data.

On the other hand, choosing a degree that is too low might not provide enough capacity to capture the true structure of the normal region, leaving the model underfitting and unable to recognize certain anomalies.

Finding the right degree for the polynomial kernel therefore requires careful consideration. It depends on the underlying nature of the dataset: If normal behavior results from meaningful interactions among features, a moderate-degree polynomial can be powerful and effective. But if the feature relationships are simple or the data is noisy, a polynomial kernel can quickly become too rigid or too sensitive. Balancing these factors is key to using the polynomial kernel effectively in One-Class SVM anomaly detection.

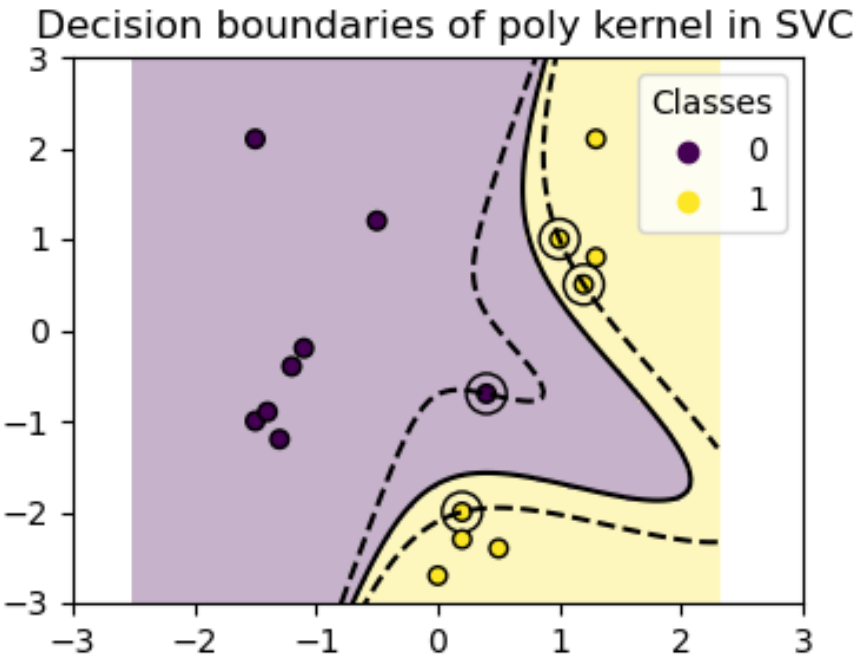


Figure 8 Poly Kernel

3.3.2.3 Radial Basis Function Kernel

The Radial Basis Function (RBF) kernel is one of the most powerful and commonly used kernels in One-Class SVM because it focuses on local similarity. Instead of relying on global linear relationships, it measures how close two points are in the input space—points near each other are considered highly similar, while similarity drops off quickly with distance. This allows the OC-SVM to form smooth, curved boundaries that naturally wrap around the true shape of the data, even when that shape is irregular or highly complex.

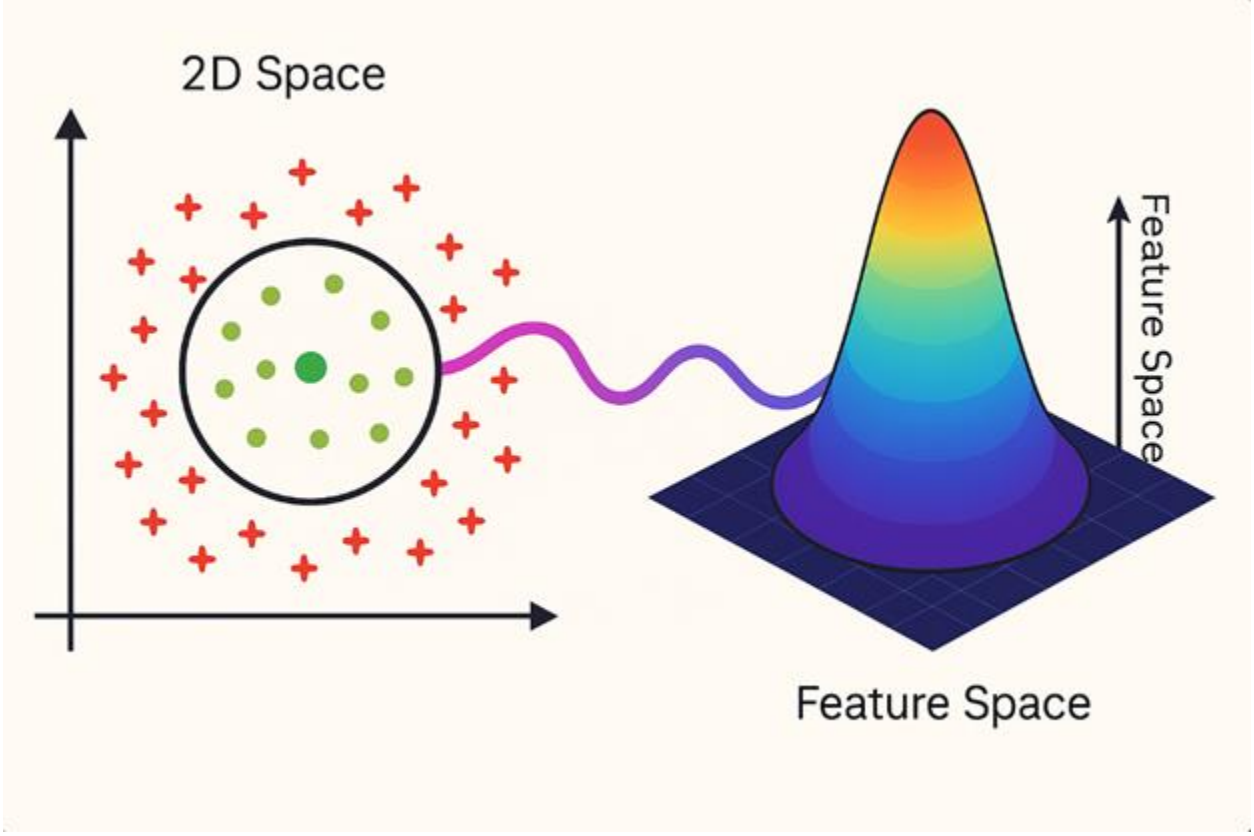


Figure 9 Radial Basis Function: 2D -> Hyper Space

Conceptually, the RBF kernel lifts the data into an infinite-dimensional feature space, but the kernel trick makes this practical by computing similarities directly without constructing that space. This ability to model nonlinear patterns is especially valuable in anomaly detection, where normal behavior rarely follows simple geometric shapes. The RBF kernel can capture clusters, curved structures, and subtle variations that linear or polynomial boundaries would miss.

A key parameter of the RBF kernel is γ (gamma). A small gamma value leads to broad, smooth boundaries that risk overlooking fine details, while a large gamma makes the model very sensitive to small fluctuations and can cause overfitting. Finding the right balance is essential for good performance.

The main drawback of the RBF kernel is its computational cost, since it requires computing pairwise similarities between points. On large datasets, this can be expensive. However, using GPU acceleration—as in your system—significantly reduces this cost by performing these operations in parallel

3.3.2.4 Sigmoid Kernel

The Sigmoid kernel—often called the hyperbolic tangent kernel—serves as a special case within Support Vector Machines (SVMs) that behaves differently from distance-based kernels. Unlike the Radial Basis Function (RBF) kernel, which measures similarity using Euclidean distance and focuses on local relationships, the Sigmoid kernel models similarity through activation-like responses. This means it maps inputs into a feature space where decisions depend on applying a tanh function to the dot product of the input vectors. In effect, the SVM separates classes by adjusting activation thresholds rather than relying purely on geometric proximity.

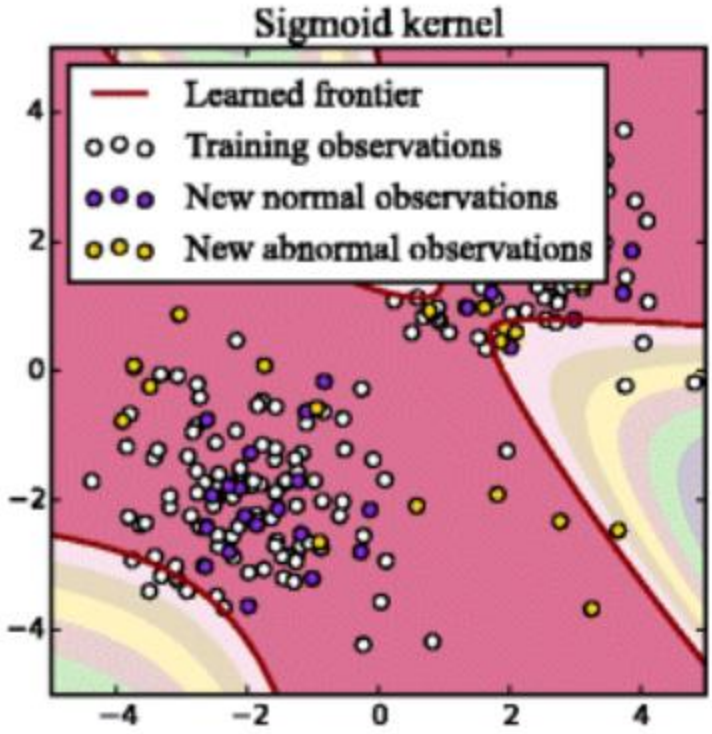


Figure 10 Sigmoid kernel classification

From a geometric standpoint, the Sigmoid kernel creates decision boundaries with global influence. While distance-based kernels typically form closed, localized hyperspheres, the Sigmoid kernel generates open, hyperbolic surfaces. As a result, even points located far from the dense regions of the feature space can meaningfully affect how the boundary curves. In DDoS detection, this property enables a One-Class SVM to identify anomalies by detecting changes in activation patterns, offering a complementary capability to traditional clustering approaches.

The effectiveness of this approach in anomaly detection depends heavily on tuning two key hyperparameters: **ν (nu)** and **γ (gamma)**. The ν parameter acts as a

regularization control, setting an upper limit on training errors and a lower limit on the proportion of support vectors. When the model is trained on benign network traffic, v essentially represents the expected contamination rate, so it's typically kept low (around 0.01–0.05) to keep false positives down and avoid misclassifying normal variations as threats.

The γ parameter, on the other hand, determines the kernel coefficient and influences the complexity of the decision boundary. In the Sigmoid function, γ controls how sharply the activation curve changes. If γ is set too high, the model becomes overly sensitive and overfits to noise in the data; if too low, the decision boundary becomes nearly linear and fails to capture the non-linear signatures of more advanced attacks. Finding the right balance between flexibility and generalization requires careful hyperparameter tuning.

3.3.3 K-Nearest Neighbor (KNN)

The k-Nearest Neighbors (k-NN) algorithm, when applied to anomaly detection, takes a very direct and intuitive approach. Unlike model-based methods such as One-Class SVM, k-NN does not build an abstract model or learn a decision boundary. Instead, it is a brute-force, instance-based method. This means that during prediction, the algorithm simply compares each new data point to all points in the training set and bases its decision on these distance comparisons. There is practically no “training” phase; the training data is simply stored and later used as a reference.

The brute-force nature of k-NN becomes especially clear in anomaly detection. For every test point, the algorithm computes its distance to all training samples, identifies the k closest ones, and examines how “far” or “close” they are. The fundamental idea is that normal points live in dense regions, surrounded by many similar observations, while anomalies tend to be isolated, sitting far from their neighbors. If the average or maximum distance to the k nearest neighbors is unusually large, the point is flagged as an anomaly.

This makes k-NN particularly well suited for anomaly detection because it does not assume any specific data distribution or global shape. Many real-world datasets—especially those involving human behavior, network traffic, sensor data, or fraud patterns—are highly irregular and do not follow neat, Gaussian-like structures. k-NN naturally adapts to complex, nonlinear, non-convex patterns because its decisions depend purely on local density rather than on a global optimization function. In settings

where anomalies are sparse and far from the main data cloud, k-NN can be extremely effective.

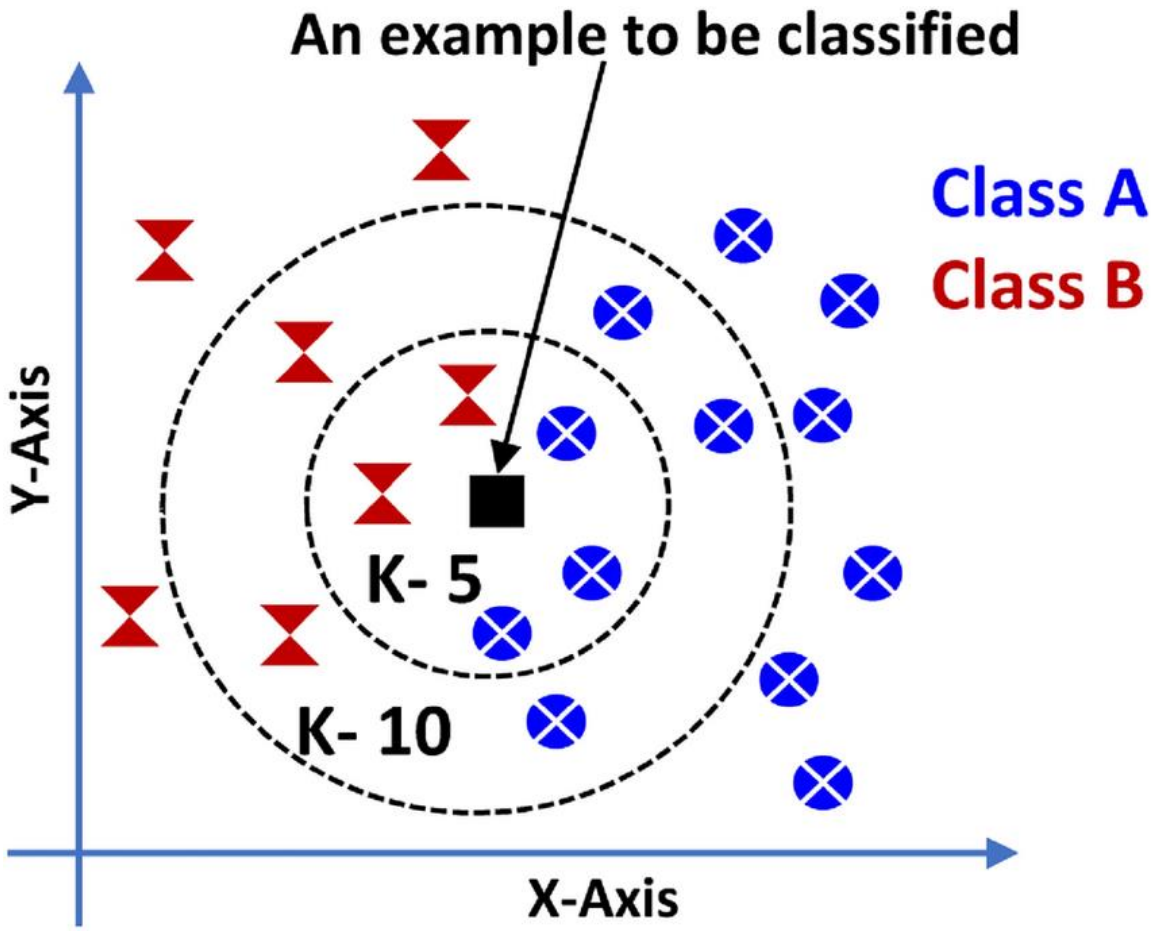


Figure 11 KNN Classification

However, this simplicity and flexibility come with several downsides. First, the brute-force comparison against all training samples makes k-NN computationally expensive, especially for large datasets. Every time we classify a new point, we must compute potentially thousands or millions of distances. Without GPU acceleration or smart indexing structures, this can quickly become impractical.

Another major challenge is that k-NN is very sensitive to feature scaling. Because its decisions rely on distance metrics, features with larger numeric ranges can

dominate the outcome unless the data is properly normalized. This is why scaling and sometimes dimensionality reduction (like PCA) are essential when using k-NN.

The choice of k also has a significant impact. A very small k may be overly sensitive to noise and produce unstable decisions, while a very large k may smooth out local structures and miss subtle anomalies. Picking the right k requires experimentation and depends on the dataset's characteristics.

Finally, because k-NN relies entirely on the stored training data, it is vulnerable to issues such as memory usage, slow prediction times, and sensitivity to irrelevant or redundant features. High-dimensional data can make distance comparisons less meaningful, a phenomenon known as the "curse of dimensionality," which can degrade the algorithm's ability to distinguish between normal and abnormal points.

Despite these limitations, k-NN remains a widely used and intuitive method for anomaly detection, especially when combined with GPU acceleration, proper preprocessing, and thoughtful parameter tuning. It offers a straightforward way to detect outliers by focusing directly on the local structure of the data, without imposing any restrictive assumptions or requiring complicated model training.

3.3.4 Principal Component Analysis

Principal Component Analysis (PCA) is a technique I use to simplify high-dimensional data while still keeping the most important information. In many datasets, a lot of the features are either correlated with each other or contain noise that doesn't really help the model learn anything useful. PCA helps me cut through that complexity by transforming the original features into a new set of uncorrelated components—these are the "principal components." They are ordered so that the first one captures the most variation in the data, the second captures the next most, and so on. By doing this, the method gives me a much cleaner and more compact representation of the dataset.

To perform PCA, the data first needs to be centered (and optionally scaled), so that each feature has roughly the same influence and everything is measured relative to its mean. After this, I compute the covariance matrix, which shows how the features vary together. If two features have a high covariance, it means they change in similar ways and likely contain overlapping information. Once I have the covariance matrix, I use an eigen-decomposition algorithm—specifically a Jacobi eigenvalue method in my implementation—to extract the eigenvalues and eigenvectors. The eigenvectors point in the directions of maximum variance, and the corresponding eigenvalues tell me how much variance lies along each direction.

After computing all the eigenpairs, I sort them so the directions with the most variance come first. Then I select only the top portion of them, depending on how much dimensionality reduction I want to achieve. Finally, I project the data onto these top

eigenvectors. The result is a reduced-dimensionality version of the dataset that still preserves the structure that matters most while filtering out noise and redundancy.

PCA is particularly useful in the context of my system, where I combine GPU-accelerated algorithms such as k-Nearest Neighbours (k-NN) and One-Class SVM (OC-SVM). k-NN, for instance, is very sensitive to high-dimensional data and becomes less effective as the number of features grows. Reducing the dimensionality helps make distances more meaningful and also speeds up computation significantly on the GPU. For OC-SVM, PCA reduces both the size of the kernel matrix and the computational cost associated with kernel evaluations. It also improves numerical stability by removing correlated features that could otherwise complicate the optimization process. Overall, PCA helps reduce overfitting, lowers memory requirements, and often improves the quality of the anomaly-detection results.

Within my architecture, PCA fits naturally into the preprocessing component. The Base Algorithm class stores all the PCA-related parameters—like the mean values and the eigenvectors—and ensures they are applied consistently during training and prediction. This design choice means that models downstream don’t need to worry about how PCA works; they simply receive the transformed data. It keeps preprocessing centralized, avoids duplication, and ensures that the models operate on clean, efficient, and stable representations of the input data.

4. Implementation

4.1 Technologies and implementation approach

The proposed DDoS detection system adopts a microservice-based architecture, separating the concerns of packet acquisition and machine-learning-driven analysis into two distinct, cooperating components. This design improves modularity, facilitates independent development and testing, and enables future deployment in distributed or containerized environments.

The first component, referred to as the network_sniffer microservice, is implemented entirely in Python. Its primary responsibility is the continuous capture of network traffic within the experimental environment. This microservice relies on *Scapy*, a widely used Python library for low-level packet manipulation and inspection. Scapy provides direct access to link-layer frames, making it suitable for extracting protocol-level features required by intrusion and anomaly detection systems. The microservice exposes a set of REST endpoints using the FastAPI framework, allowing external control of the capture process, including starting and stopping continuous monitoring and retrieving recently collected packets. All captured traffic is parsed and transformed into structured feature vectors, which are then transmitted via HTTP to the analysis microservice for classification.

The second component, the network_analyzer microservice, combines Python orchestration with a high-performance backend implemented in C++ and CUDA. This service also uses FastAPI to expose endpoints for model training and real-time prediction using two machine-learning algorithms: *k-Nearest Neighbors (KNN)* and *One-Class*

Support Vector Machines (OC-SVM). The computational core of these algorithms is encapsulated in a dedicated library written in modern C++ and accelerated with NVIDIA CUDA to leverage GPU parallelism. The library is made accessible to Python through a pybind11 extension module named algos, allowing the Python-based microservice to invoke high-performance routines with minimal overhead.

The algos module includes a GPU-accelerated implementation of KNN, which performs large-scale distance computations using CUDA kernels optimized for parallel execution. Similarly, the OC-SVM implementation supports multiple kernel functions (e.g., RBF, polynomial, and sigmoid) and relies on CUDA-based routines to compute kernel matrices and decision scores efficiently. Both algorithms inherit shared preprocessing operations—such as scaling and optional PCA transformation—through a dedicated BaseAlgo class, ensuring consistent data transformations between training and prediction stages. The module additionally provides utility functions for PCA-based feature scaling and evaluation metrics such as ROC-AUC and recall, thereby supporting the full lifecycle of model development and validation.

The current testing and development environment is based on Windows 10, selected due to the use of GNS3 as the primary network virtualization and traffic-generation platform. GNS3 provides a high-fidelity emulation environment that allows the system to be tested against complex network topologies and synthetic DDoS scenarios. However, the GNS3 virtual machine relies on specific virtualization configurations that are incompatible with the Hyper-V and WSL2 technologies required to operate Docker Desktop on Windows. As a result, although the microservice architecture is inherently suitable for containerization, Docker-based deployment is not currently feasible in this environment. Both microservices therefore run natively on the Windows host during the experimental phase.

Despite these constraints, the implementation is designed to be platform-agnostic. In future deployments, where the Hyper-V/Docker conflict does not apply, the system can be migrated to a Linux-based server equipped with an NVIDIA GPU, where each microservice can be containerized using Docker. Such an environment would provide improved scalability, better resource isolation, and more efficient GPU utilization for real-time DDoS detection.

4.2 Programming Languages, Libraries, and Frameworks

The implementation of the system relies on a combination of programming languages and external libraries, each selected to fulfil specific functional and performance requirements. The choice of technologies was dictatied by considerations of development efficiency, computational performance, interoperability, and suitability for processing high-volume network traffic.

Python serves as the primary connection language for both microservices. Its extensive ecosystem, rapid prototyping capabilities, and robust support for networking tools make it a natural choice for implementing the application-level logic of the system. The network_sniffer microservice benefits from Python's mature packet-processing libraries, most notably Scapy, which facilitates low-level traffic capture and protocol parsing within the GNS3 test environment. Additionally, the use of FastAPI enables the rapid development of high-performance REST interfaces, allowing seamless communication between the system's components. Python's compatibility with native extensions further permits the integration of high-performance modules without sacrificing ease of development.

C++ is utilised for the implementation of the core machine learning algorithms that underpin the DDoS detection process. Its strong memory-management capabilities and deterministic performance characteristics make it well-suited for computationally demanding tasks such as distance calculations in k-Nearest Neighbours (KNN) and kernel evaluations in One-Class SVM (OC-SVM). Writing the algorithms in C++ ensures fine-grained control over data structures, execution flow, and resource allocation, all of which are critical for achieving low latency and high throughput in real-time traffic analysis.

Pybind11 is a lightweight, modern C++ library designed to create seamless bindings between C++ and Python. Unlike older binding frameworks that require verbose configuration files or complex wrapper code, pybind11 uses C++11 metaprogramming techniques to expose C++ classes, functions, and data structures directly to Python with minimal boilerplate. This design significantly reduces the effort required to integrate performance-critical C++ modules into Python applications. Pybind11 is header-only, meaning it does not require additional libraries or build dependencies, and integrates cleanly into existing C++ projects. Its syntax closely mirrors Python's own object model, which makes it intuitive to use and allows developers to map C++ constructs to Python types in a natural, expressive manner.

Moreover, pybind11 contributes to a clean and maintainable architecture. Instead of implementing separate APIs or communication protocols to interact with the C++ backend, the Python-based microservice interacts with the GPU-accelerated algorithms through a single shared module (algos). This modularity simplifies the codebase, reduces potential sources of error, and ensures that the performance benefits of CUDA acceleration are preserved without complicating the Python layer. As a result, pybind11 enables the system to retain Python's ease of development while leveraging the full computational capabilities of modern GPUs, making it a crucial component of the overall design.

To further increase computational efficiency, particularly given the high dimensionality and volume of traffic data typical of DDoS detection scenarios, the system employs CUDA for GPU acceleration. CUDA enables the offloading of highly parallelizable computations—such as large-scale matrix operations, kernel evaluations, and nearest-neighbour searches—to the GPU. This accelerates both training and inference phases of the detection algorithms, resulting in performance improvements unattainable through CPU-only implementations.

By transferring all major computational tasks to the GPU, the system was able to overcome the performance limitations imposed by the local network simulation simulation environment. The GPU executes the training and prediction procedures while the CPU remains dedicated to packet capture, API orchestration, and managing the microservices. This separation of responsibilities ensures that the machine learning workload does not interfere with essential system operations during periods of high traffic intensity, such as those experienced during a simulated DDoS attack. Moreover, the use of GPU parallelism significantly improves scalability. Operations such as distance evaluations for k-Nearest Neighbours and kernel calculations for the One-Class SVM typically scale poorly on CPUs because their computational cost increases rapidly with the size of the dataset. On the GPU, however, these operations can be distributed across thousands of threads, enabling the system to handle larger datasets and higher traffic volumes without a corresponding deterioration in performance.

The relocation of the intensive computations to the GPU also contributes to greater stability and responsiveness. Because the CPU is no longer burdened with heavy numerical operations, it is able to maintain steady performance for tasks that are inherently sequential or I/O-bound, including the handling of incoming packets, communication between microservices, and the management of FastAPI endpoints. This

balance is essential in a real-time system, where delays in packet capture or API response times could compromise the accuracy or timeliness of the detection mechanism. Most critically, GPU acceleration made the entire experimental setup feasible. The simultaneous simulation of a live network environment in GNS3 and the execution of machine learning tasks led to severe CPU saturation during initial trials. The host system simply lacked the computational headroom to emulate a DDoS attack and train detection models concurrently. Translating these computations to the GPU alleviated this bottleneck, enabling the system to function reliably under the demanding conditions imposed by local traffic simulation and allowing the models to adapt dynamically to the ongoing attack scenario

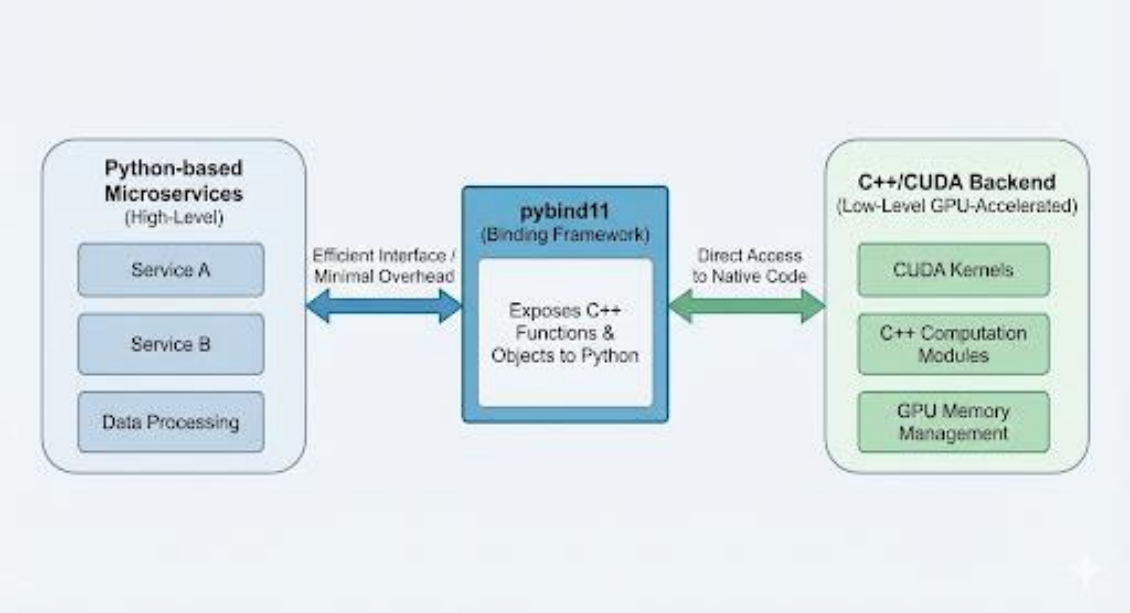


Figure 12 Connection between C++ algorithms Library and python micro services

As shown on the figure, integration between the Python-based microservices and the C++/CUDA backend is achieved through pybind11, a modern, lightweight binding framework. Pybind11 removes much of the complexity traditionally associated with creating Python bindings for C++ code and allows native C++ objects and functions to

be exposed directly to Python with minimal overhead. This facilitates a clean and efficient interface between the high-level Python services and the low-level GPU-accelerated computation modules, while maintaining a modular and maintainable architecture.

4.3 Installation Guidelines

The installation of the DDoS Detection System requires careful consideration of both the underlying hardware environment and the network infrastructure in which the system will operate. In a production-oriented topology, the system is intended to run on a dedicated virtual machine (VM) or physical host positioned within a top-of-rack (ToR) switch architecture. The ToR switch must be configured to mirror ingress and egress traffic from selected interfaces or VLANs and forward that mirrored traffic through a tunneled port to the VM running the detection framework. This mechanism ensures that the monitoring system receives a complete and faithful copy of the traffic flowing within the rack, enabling accurate and timely anomaly detection without interfering with the operational workload of the servers attached to the switch.

For the VM to function as an effective monitoring node, it must possess the necessary rights and capabilities to capture and process raw network traffic. Specifically, the operating system must allow the VM to operate in promiscuous mode, granting it the ability to receive all packets arriving through the mirrored interface, not only those addressed to its own MAC or IP. In virtualized environments, this often requires explicit configuration at both the hypervisor and virtual switch levels—such as enabling promiscuous mode, forged transmits, or allowing MAC address changes—depending on the underlying virtualization platform. Without these permissions, the network_sniffer microservice would be unable to access low-level packet data, thereby compromising the

integrity and completeness of the DDoS detection process. The VM must therefore be deployed with full packet capture privileges to ensure continuous, lossless ingestion of mirrored network traffic.

In addition to the network-level permissions, the system requires a software stack capable of supporting both high-throughput packet capture and GPU-accelerated machine learning. The host must provide an NVIDIA GPU compatible with the CUDA runtime, accompanied by the appropriate CUDA Toolkit and driver installation. A C++17-compliant compiler is necessary to build the C++/CUDA backend and generate the pybind11 extension module used by the Python microservices. The Python environment itself must include FastAPI for service orchestration, Scapy for packet capture, pybind11 for Python-C++ integration, and an HTTP client library for inter-service communication. When deployed in a ToR-based architecture, these components collectively enable the detection system to receive mirrored traffic, process it in real time, and execute GPU-accelerated inference on the incoming packet stream.

5. Testing

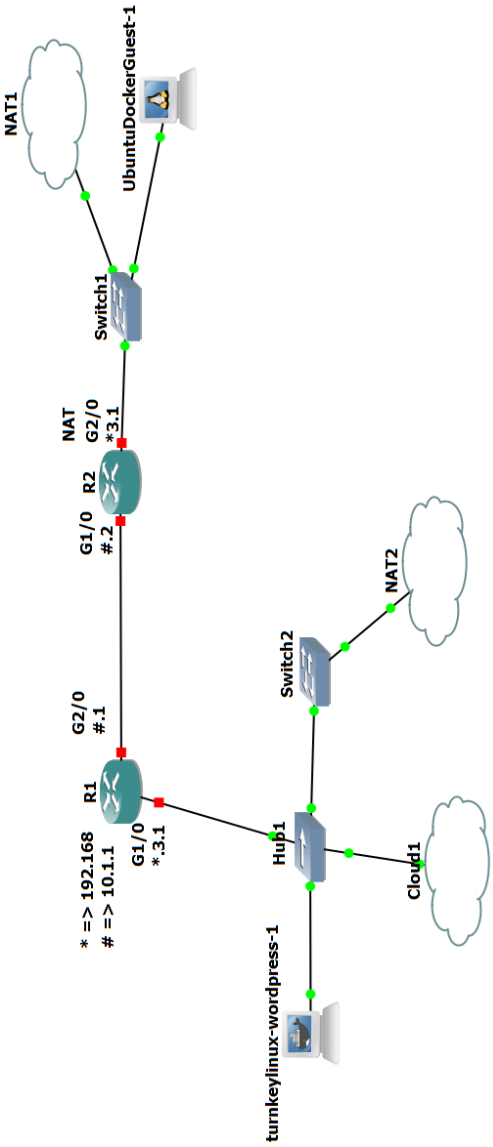


Figure 13: GNS3 simulation of a enterprise network

GNS3 (Graphical Network Simulator 3) provides a flexible and realistic environment for prototyping and evaluating network-security mechanisms, making it well suited for studies that require controlled manipulation of routing, switching, and traffic-translation behavior. In this project, GNS3 is used to construct a multi-segment topology incorporating routers, switches, virtual machines, and multiple layers of Network Address Translation (NAT). This setup closely mirrors operational networks in which end-to-end visibility is limited, and security systems must operate without access to stable or unique source IP addresses.

The use of NAT within the simulated environment is particularly relevant for DDoS research. Because NAT masks internal host identities and aggregates flows behind shared public addresses, traditional IP-based detection or filtering becomes ineffective. Instead, the detection system must rely on statistical and behavioral indicators embedded in the traffic itself—such as SYN rates, the ratio of SYN to SYN-ACK packets, packet bursts, half-open connection counts, and anomalies in aggregate connection patterns. GNS3 enables fine-grained observation of these behaviors by allowing packet capture at intermediate routers even when the original source IPs are no longer visible after translation. This makes the simulator an ideal platform for developing and testing machine-learning–based detection models that infer the presence of an attack despite incomplete or NAT-obscured metadata.

By integrating virtual hosts capable of generating real attack traffic, such as SYN floods and HTTP floods, GNS3 provides a controlled yet realistic environment for evaluating the robustness of the proposed DDoS detection system. The simulator's ability to emulate NAT, route redistribution, interface-level congestion, and multi-hop propagation ensures that the trained model is tested against conditions that closely resemble those encountered in production networks. Consequently, GNS3 serves not only as a test harness for generating repeatable experiments but also as a critical tool for

validating that detection can occur effectively even when traditional packet identifiers are unavailable.

5.2 Types of DDoS Attacks

5.2.1 SYN Flodd Attacks

A SYN flood is a classical transport-layer Denial-of-Service attack that exploits the design of the TCP three-way handshake. Under normal conditions, a TCP connection is established through a sequence of SYN, SYN-ACK, and ACK packets. In a SYN flood, the attacker transmits an unusually high volume of SYN packets but deliberately omits the final ACK response. Each incomplete handshake forces the server to allocate memory and maintain state for a “half-open” connection while it waits for the nonexistent acknowledgement. When repeated at scale, the backlog of pending connections quickly fills, preventing legitimate clients from establishing new sessions. Because the attacker often spoofs source addresses, the server cannot easily filter malicious traffic, and the attack can propagate through intermediate routers and firewalls before mitigation measures can take effect. This combination of resource exhaustion and traffic obfuscation makes SYN floods one of the most disruptive network-layer DDoS techniques.

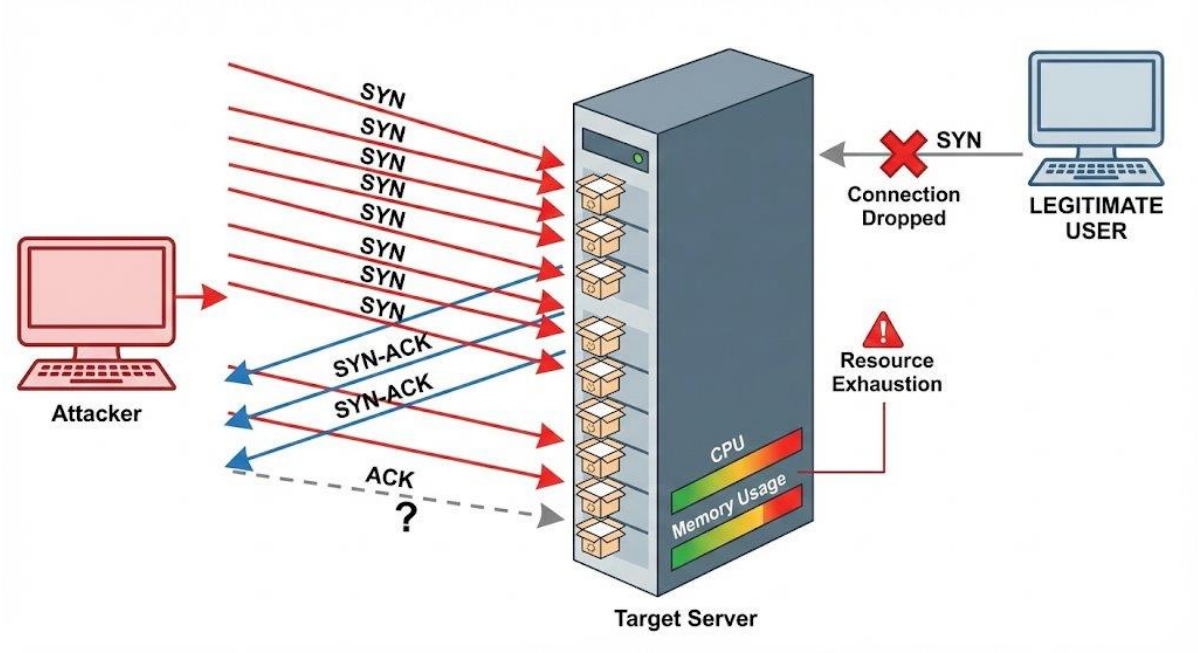


Figure 14: SYN Flood

5.2.1 HTTP Flodd Attacks

An HTTP flood is a higher-layer DDoS attack that targets the web application stack rather than the underlying transport mechanisms. In contrast to packet-level floods, HTTP floods involve sending large volumes of syntactically valid HTTP requests—typically using many concurrent clients or distributed bots—to overwhelm a server's processing capacity. Because each request may trigger backend operations such as database queries, dynamic content generation, or session handling, even modest increases in request rate can impose substantial computational overhead on the server. As a result, HTTP floods degrade service availability not by saturating bandwidth but by exhausting application-level resources, including worker threads, CPU cycles, and memory allocations. The difficulty in distinguishing malicious requests from legitimate user activity further complicates detection, particularly during attacks that mimic normal traffic patterns or target resource-intensive endpoints. These

characteristics make HTTP floods a significant threat to modern web services and a crucial component of comprehensive DDoS evaluation.

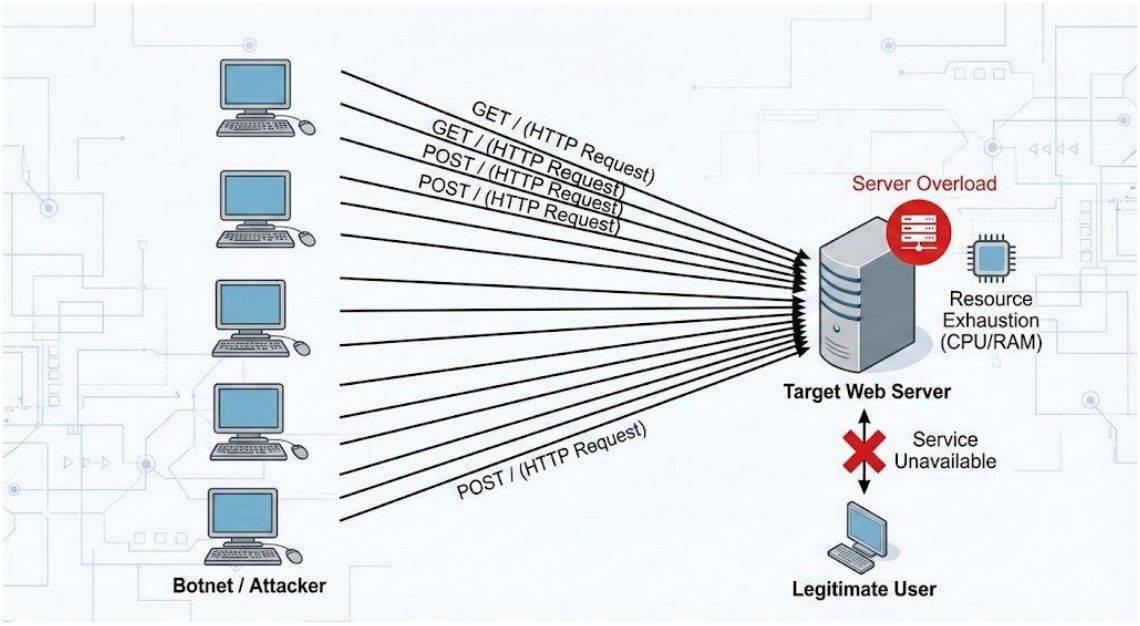


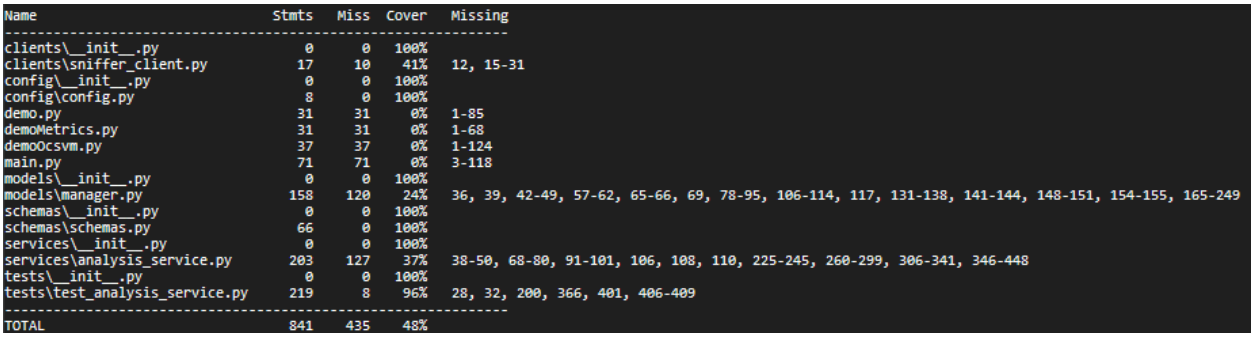
Figure 15 HTTP Flood

5.3 Unit Test

The coverage analysis shown in figure reveals that several modules within the project exhibit limited or no unit-test coverage. This distribution of coverage is intentional and follows from the architectural role of each module within the system. The primary testing effort focuses on the analytical pipeline and model-evaluation logic, while auxiliary components—particularly those concerned with demonstrations, configuration, and external I/O—are excluded from systematic unit testing.

A significant proportion of uncovered statements arises from the demonstration scripts located in demo.py, demoMetrics.py, and demoocsvm.py. These files function

solely as standalone examples illustrating how the underlying C++-backed algorithms (exposed through the algos module) can be invoked for demonstration purposes. They are not integrated into the operational DDoS-detection workflow and contain top-level execution code, ad-hoc data generation, and manual print statements that do not meaningfully contribute to system correctness. Consequently, including them in the test suite would not enhance the reliability or validity of the experimental results. Their exclusion is consistent with standard software-engineering practice, where demonstration and exploratory scripts fall outside the scope of unit-level verification.



Name	Stmts	Miss	Cover	Missing
clients__init__.py	0	0	100%	
clients\sniffer_client.py	17	10	41%	12, 15-31
config__init__.py	0	0	100%	
config\config.py	8	0	100%	
demo.py	31	31	0%	1-85
demoMetrics.py	31	31	0%	1-68
demoOcsvm.py	37	37	0%	1-124
main.py	71	71	0%	3-118
models__init__.py	0	0	100%	
models\manager.py	158	120	24%	36, 39, 42-49, 57-62, 65-66, 69, 78-95, 106-114, 117, 131-138, 141-144, 148-151, 154-155, 165-249
schemas__init__.py	0	0	100%	
schemas\schemas.py	66	0	100%	
services__init__.py	0	0	100%	
services\analysis_service.py	203	127	37%	38-50, 68-80, 91-101, 106, 108, 110, 225-245, 260-299, 306-341, 346-448
tests__init__.py	0	0	100%	
tests\test_analysis_service.py	219	8	96%	28, 32, 200, 366, 401, 406-409
TOTAL	841	435	48%	

Figure 16: Unit Test

The configuration layer, represented primarily by config/config.py, remains entirely uncovered for similar reasons. This module contains static configuration parameters, environment defaults, and constant definitions used by other components. Because the file has no computational logic and its values are automatically exercised when higher-level modules are invoked, explicit unit tests offer little additional value. The role of this module is infrastructural rather than algorithmic, and its absence from coverage does not affect the integrity of the machine-learning evaluation.

The clients/sniffer_client.py module, which also exhibits partial coverage, is intentionally left largely untested because it interfaces directly with the operating system's packet-capture mechanisms and network stacks. Its functionality depends on

live traffic, socket availability, and asynchronous event handling. Testing such components at the unit level would require substantial mocking of system APIs or the construction of a full integration environment, neither of which aligns with the methodological focus of this work. The research questions addressed in this project concern the detection performance derived from processed traffic features, not the behaviour of the network-capture client itself; thus, this module is validated implicitly through live experiments rather than through isolated unit tests.

Partial coverage in the core service logic (services/analysis_service.py) and model-management module (models/manager.py) is attributed to defensive branches, error-handling pathways, and FastAPI integration code that are not executed during normal operation. The test suite targets the central execution paths—such as model training, prediction, evaluation routines, and data preprocessing—while peripheral branches, including those triggered by malformed inputs, unused parameter combinations, or exceptional runtime states, are not exhaustively tested. These lines do not influence the machine-learning behaviour under typical conditions and therefore do not impact the conclusions of the DDoS-detection evaluation.

5.4 Evaluation Metrics

Evaluating the performance of a DDoS-detection system requires metrics that meaningfully reflect its ability to distinguish attack traffic from benign activity under a wide range of network conditions. In this context, ROC–AUC (Receiver Operating Characteristic – Area Under the Curve) and recall are particularly valuable because they capture complementary aspects of classifier behaviour that are directly relevant to intrusion-detection scenarios. ROC–AUC provides a threshold-independent measure of separability between normal and malicious traffic, quantifying how reliably the model assigns higher anomaly scores to attack windows than to legitimate ones. This is especially important in network environments affected by NAT, routing asymmetries, and heterogeneous traffic profiles, where absolute thresholds may be difficult to tune and where decision boundaries vary across time and load conditions. A high ROC–AUC therefore indicates that the underlying scoring function is robust and that the model can discriminate effectively even before a fixed threshold is applied.

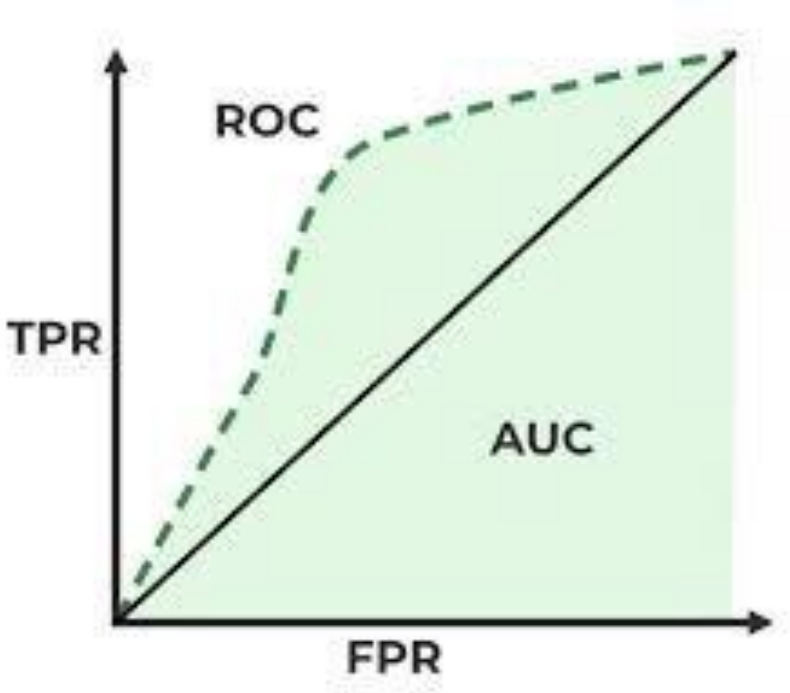


Figure 17 ROC-AUC

Recall, shown on figure serves as a threshold-dependent metric that captures the system's sensitivity to ongoing attacks. In a DDoS context, the cost of false negatives—failing to detect malicious bursts—is significantly higher than the cost of false positives. Missing an attack window can allow the adversary to consume resources, degrade service availability, and destabilize network segments before mitigation mechanisms can intervene. Because recall measures the proportion of actual attack instances that the detector successfully identifies, it directly reflects the model's ability to maintain situational awareness during periods of congestion or protocol abuse. High recall indicates that the detection pipeline consistently flags attack conditions even when the traffic mix is noisy, bursty, or partially obscured by NAT translation.

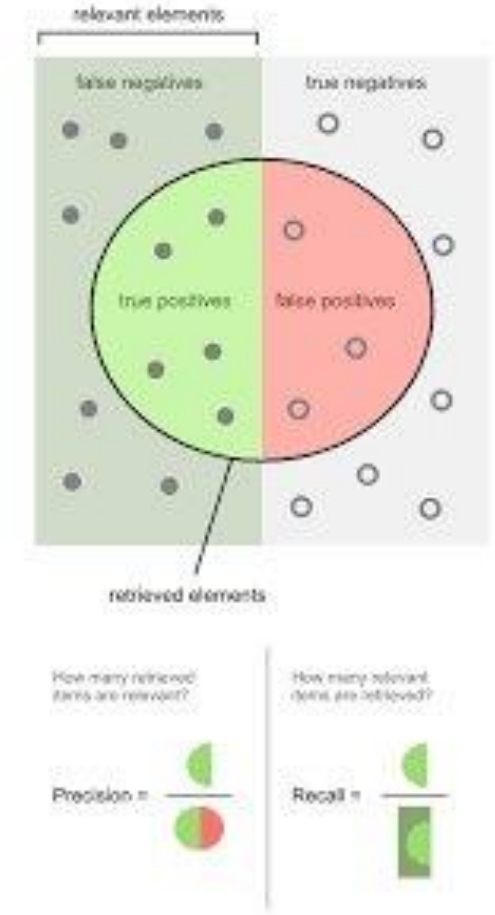


Figure 18 Recall

Taken together, ROC–AUC and recall offer a balanced and meaningful assessment of the detection system. ROC–AUC evaluates the intrinsic discriminative capacity of the model independent of operational thresholds, while recall quantifies practical detection performance at the decision boundary used in deployment. Their combined use ensures that the system is both theoretically capable of separating malicious from benign traffic and empirically effective in identifying attacks under real, NAT-affected network conditions. This dual perspective is essential for validating the reliability of a DDoS-detection approach intended for complex, dynamic, and partially observable network environments.

6. Results and Conclusion

Model name	ROC-AUC	Recall
knn_5	0.96	0.92
knn_5_pca	0.96	0.92
ocsvm_poly	0.95	0.90
ocsvm_linear	0.95	0.92
ocsvm_rbf	0.98	0.97
ocsvm_poly_bad_data	0.5	1
ocsvm_linear_bad_data	0	1
ocsvm_rbf_bad_data	0	1

Table 1 Results from testing loaded models

The performance results reveal a clear distinction between models trained on clean, well-curated datasets and those trained on data that contains contamination or mislabeled traffic. The KNN models (knn_5 and knn_5_pca) achieve strong and consistent results, each obtaining a ROC–AUC of 0.96 and a recall of 0.92, indicating that both models reliably distinguish benign traffic from attack traffic across threshold variations and successfully detect the vast majority of malicious events. The PCA variant performs identically, suggesting that dimensionality reduction neither improves nor degrades discriminative capability in this specific feature space.

The OC-SVM models trained on high-quality data (ocsvm_poly, ocsvm_linear, and ocsvm_rbf) likewise exhibit strong performance, with ROC–AUC values ranging from 0.95 to 0.98 and recall values between 0.90 and 0.97. These results demonstrate that, when supplied with clean training data, OC-SVM can learn stable boundaries that effectively

capture the statistical characteristics of normal traffic. Notably, the RBF kernel achieves the highest scores, reflecting its suitability for modeling nonlinear decision regions typical in complex network-behavior patterns.

However, the OC-SVM models trained on “bad data” (ocsvm_poly_bad_data, ocsvm_linear_bad_data, and ocsvm_rbf_bad_data) illustrate the fundamental vulnerability of anomaly detection techniques to data contamination. Although these corrupted models technically report Recall = 1, this is misleading: a classifier that simply labels *all* samples as hostile will trivially achieve perfect recall, yet be entirely useless in practice. Their ROC–AUC values—0.5, 0, and 0, respectively—show that the learned scoring function has collapsed. An ROC–AUC around 0.5 indicates random guessing, while an AUC of 0 indicates complete inversion, with the model assigning higher anomaly scores to benign traffic than to attacks. These outcomes are symptomatic of poisoned or unrepresentative training data, which causes the OC-SVM boundary to shift in the wrong direction.

This project set out to design and evaluate a machine-learning–based system capable of detecting DDoS attacks in a realistic, NAT-obscured network environment. By constructing a controlled topology in GNS3 and generating live SYN-flood and HTTP-flood attacks, the study reproduced conditions similar to those encountered in operational networks, where packet-level visibility is limited and traditional IP-based detection strategies are often ineffective. Within this setting, a dedicated feature-extraction pipeline was developed to capture behavioral indicators of hostile activity, such as SYN rates, half-open connections, and traffic burstiness. These feature vectors formed the basis for training and testing two types of models: K-Nearest Neighbors classifiers and One-Class Support Vector Machines with several kernel configurations.

The results demonstrate that both model families can perform very well when provided with clean, attack-free training data. KNN models with and without PCA

achieved consistently high ROC–AUC and recall values, indicating reliable discrimination between benign and malicious traffic. Similarly, OC-SVM models trained on high-quality data showed strong detection performance, with the RBF kernel offering the most expressive decision boundary. However, the experiments also revealed a critical vulnerability: when trained on contaminated or unverified data, the OC-SVM models collapsed entirely, producing degenerate decision functions with ROC–AUC scores near zero while still reporting perfect recall by labeling all traffic as hostile. This behavior illustrates how strongly the performance of anomaly-detection methods depends on the purity of the training dataset.

This dependence poses a significant challenge for real-world deployment. In practical network environments, especially those involving NAT and complex multi-hop routing, it is rarely possible to guarantee that traffic is free from ongoing attacks at the moment of model initialization. Any stealthy or low-volume attack present during baseline collection can distort the learned model and undermine the system's reliability. While the current work demonstrates that machine-learning techniques can successfully detect DDoS activity under idealized conditions, it also highlights the need for mechanisms that can verify or enforce the integrity of training data.

Therefore, a promising direction for future research is the development of models and training procedures that are robust to data contamination or that can autonomously determine whether the observed traffic represents a legitimate baseline. Approaches that integrate unsupervised drift detection, statistical robustness, adversarial contamination modeling, or self-diagnosing anomaly filters could significantly improve the dependability of such systems in operational settings. Advancing these techniques would bring machine-learning–based DDoS detection closer to practical, autonomous deployment in modern networks, where uncertainty about traffic purity is the norm rather than the exception.

7. References

Abiramasundari, S., and V. Ramaswamy. “Distributed denial-of-service (DDOS) attack detection using supervised machine learning algorithms.” *Scientific Reports*, vol. 15, no. 1, 16 Apr. 2025, https://doi.org/10.1038/s41598-024-84879-y.

Alduailij, Mona, et al. “Machine-learning-based DDoS attack detection using mutual information and random forest feature importance method.” *Symmetry*, vol. 14, no. 6, 27 May 2022, p. 1095, https://doi.org/10.3390/sym14061095.

Almaraz-Rivera, Josue Genaro, et al. “Transport and application layer ddos attacks detection to IOT devices by using machine learning and Deep Learning Models.” *Sensors*, vol. 22, no. 9, 28 Apr. 2022, p. 3367, https://doi.org/10.3390/s22093367.

Arango-López, Jeferson, et al. “Cloud-based Deep learning architecture for ddos cyber attack prediction.” *Expert Systems*, vol. 42, no. 1, 23 Jan. 2024, https://doi.org/10.1111/exsy.13552.

Bahashwan, Abdullah Ahmed, et al. “A deep learning-based mechanism for detecting variable-rate ddos attacks in software-defined networks.” *Mobile Networks and Applications*, vol. 30, no. 1–2, Apr. 2025, pp. 12–41, https://doi.org/10.1007/s11036-025-02458-5.

Leka, Elva, et al. “Web application firewall for detecting and mitigation of based ddos attacks using machine learning and Blockchain.” *TEM Journal*, 27 Nov. 2024, pp. 2802–2811, https://doi.org/10.18421/tem134-17.

Li, Yi, et al. “A transformer-based framework for ddos attack detection via temporal dependency and behavioral pattern modeling.” *Algorithms*, vol. 18, no. 10, 4 Oct. 2025, p. 628, https://doi.org/10.3390/a18100628.

Loaiza, Francisco L., et al. Institute for Defense Analyses, 2019, *Utility of Artificial Intelligence and Machine Learning in Cybersecurity*, http://www.jstor.org/stable/resrep22692. Accessed 5 Dec. 2025.

Satpathy, Suneeta, et al. “Cloud-based ddos detection using hybrid feature selection with Deep Reinforcement Learning (DRL).” *Scientific Reports*, vol. 15, no. 1, 21 Oct. 2025, https://doi.org/10.1038/s41598-025-18857-3.

Wahab, Sheikh Abdul, et al. “A multi-class intrusion detection system for ddos attacks in IOT networks using Deep Learning and transformers.” *Sensors*, vol. 25, no. 15, 6 Aug. 2025, p. 4845, https://doi.org/10.3390/s25154845.

Table of Figures