INTEL-UNNATI AI&ML JOURNAL
Number 2 (2025), pp. 1–14 Code:IUIT-2025

MACHINE LEARNING

# Network Traffic Monitoring and Malicious Packet Detection Using `Python` and Machine Learning

Adithyan B, Elena Elizebeth Cherian, and Harshendu V Kurup

Saintgits Group of Institutions, Kottayam, Kerala

In the rapidly evolving landscape of network communication, safeguarding against cyber threats has become a fundamental necessity. This project proposes a lightweight, intelligent network traffic monitoring system that can detect and classify malicious activities in real-time using machine learning. Leveraging Python, Flask, and containerized deployment through Docker, the system captures packet-level data, extracts relevant features, and feeds it into trained models to detect abnormal behavior. A simple yet intuitive web dashboard allows users to visualize traffic flows and threat categories efficiently. Designed with modularity and scalability in mind, this tool aims to be a practical addition to the cybersecurity toolkit, especially in academic and small enterprise networks. The results demonstrate promising accuracy and real-time responsiveness, setting a solid foundation for further research and industrial applications in network security.

## 1 Introduction

With the explosion of internet-connected devices and cloud-based infrastructures, modern networks face an unprecedented volume and diversity of cyber threats. Attacks such as malware propagation, data exfiltration, and denial of service have become more sophisticated, often bypassing traditional rule-based intrusion detection systems (IDS). Thus, there is a growing need for adaptive and intelligent monitoring solutions that not only inspect network traffic but also learn and evolve from data.

This project, developed under the Intel Unnati programme, focuses on building an automated network traffic monitoring and analysis system. It uses machine learning models to detect malicious traffic by analyzing packet data in real time. The system consists of a Python-based backend for data processing, a Flask-based dashboard for visualization, and uses Docker for simplified deployment. Key objectives include achieving reliable classification of network traffic, ensuring ease of use, and maintaining performance under real-time constraints.

This report documents the entire development cycle, from design to implementation and evaluation. By exploring this project, readers gain insight into how modern ML techniques can be effectively applied to enhance cybersecurity.

# 2 Libraries Used

In the project for various tasks, the following packages are used:

```
NumPy
Pandas
Scikit-learn
Flask
Scapy
Psutil
Joblib
JSON
Socket
Logging
Threading
Queue
Collections
Statistics
Datetime
```

# 3 Methodology

The Network Security Monitor employs a multi-layered architecture designed for real-time monitoring, analysis, and threat detection. The system follows a modular approach with distinct components for data collection, processing, analysis, and visualization.The system architecture is built upon four core layers:

**Data Acquisition Layer:** Responsible for capturing network packets and system metrics
**Processing Engine:** Handles real-time data processing and filtering
**Analysis Module:** Implements threat detection algorithms and traffic classification
**Presentation Layer:** Provides interactive dashboard and visualization components

## 3.1 Architectural Framework

The system architecture is built upon four core layers:

- **Data Acquisition Layer:** Responsible for capturing network packets and system metrics.
- **Processing Engine:** Handles real-time data processing and filtering.
- **Analysis Module:** Implements threat detection algorithms and traffic classification.
- **Presentation Layer:** Provides interactive dashboard and visualization components.

| Component | Function | Technology Stack / Update Frequency |
|---|---|---|
| Network Interface Monitor | Packet capture and interface statistics | Python `psutil`, `scapy` / Real-time |
| Traffic Analyzer | Protocol analysis and classification | Machine Learning algorithms / 1-second intervals |
| Threat Detection Engine | Anomaly detection and alert generation | Statistical analysis, pattern matching / Continuous |
| Dashboard Controller | Web interface and data visualization | Flask, Chart.js, WebSocket / Real-time updates |
| Data Storage | Metrics storage and historical analysis | SQLite / JSON / Configurable intervals |

Table 1: Technology stack and component functions.

### 3.2 Data Flow Architecture

The system implements a continuous data pipeline as illustrated in the following process flow:

**Network Interface** → **Packet Capture** → **Traffic Analysis** → **Threat Detection** → **Dashboard Visualization**

## 4 Implementation

### 4.1 System Overview

The Network Security Monitor was implemented as a modular web-based system using Python and Flask for backend services, and modern web technologies for the dashboard. The architecture ensures scalability, maintainability, and real-time responsiveness.

### 4.2 Backend Architecture

The backend integrates:

- **Flask** :Web framework for serving APIs and dashboard
- **Scapy** : For packet capture and analysis
- **psutil** : To monitor system resource usage
- **WebSocket** : Enables real-time data streaming

Logging, configuration management, and multi-threaded data loops support efficient monitoring and threat detection.

### 4.3 Real-Time Monitoring

Continuous monitoring captures:

- Network interface stats (bytes sent/received)
- CPU and memory utilization

- Active connection tracking
- Anomaly and threat patterns

## 4.4 Dashboard Features

The dashboard presents:

- **System Stats Panel** : Real-time CPU/memory with indicators
- **Network Stats Panel** : Live interface traffic view
- **Filtering Options** : Based on time, severity, and traffic type
- **Data Export** : For offline analysis

## 4.5 Traffic Analysis and Visualization

- **Dual-stream graphs** : Upload/download trends
- **Dynamic scaling** : Adjusts to traffic volume
- **Historical view** : Custom time window analysis

## 4.6 Threat Detection System

Implemented features include:

- **Anomaly Detection Engine** : Unusual pattern detection
- **Real-time Alerts** : With timestamps and severity
- **Categorization** : Events grouped by type and severity

## 4.7 Traffic Classification

A classification engine segments traffic into:

- Upload / Download
- Interactive / Mixed / Idle connections

Data is visualized using pie charts for clarity.

## 4.8 Connection Monitoring

Live tracking includes:

- Local and remote endpoints
- Connection states (ESTABLISHED, LISTENING, etc.)
- Port usage analysis

## 4.9 Frontend Implementation

Technologies used:

- HTML5, CSS3 : Responsive design
- JavaScript : Asynchronous updates
- Chart.js : Dynamic graphs
- WebSocket : Live data streaming

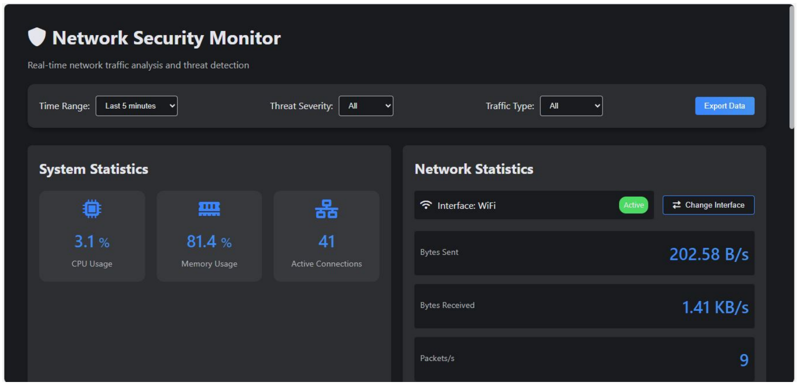


Figure 1: Main dashboard interface displaying system and network statistics in real-time, including CPU usage, memory usage, active connections, and current traffic metrics.

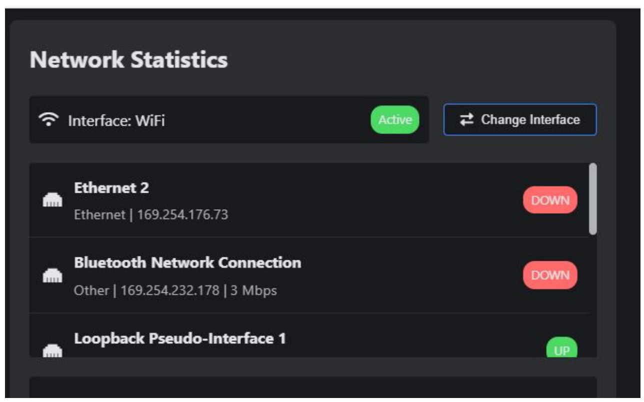


Figure 2: Active network interface selection showing Ethernet, Bluetooth, and Loopback status with live availability.

## 4.10 System Deployment

- **Local Development** : Python virtual environment
- **Docker Deployment** : Containerized for consistency
- **Production Ready** : Supports concurrent users

The system showcases a complete real-time solution for network security, integrating monitoring, threat detection, and analytics into a single responsive interface.

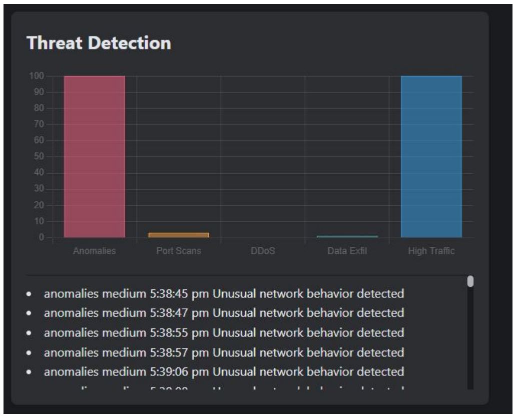


Figure 3: Threat detection module displaying anomalies, port scans, and high traffic incidents over time.

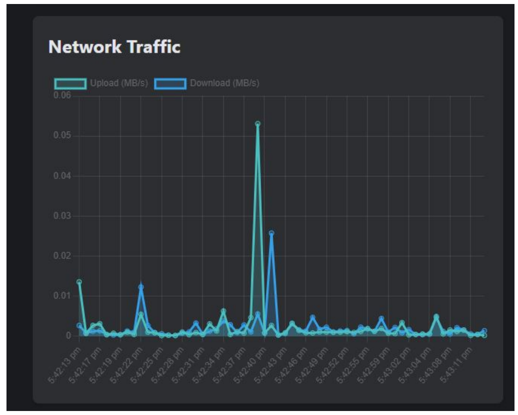


Figure 4: Network traffic graph showing upload and download rates over time.

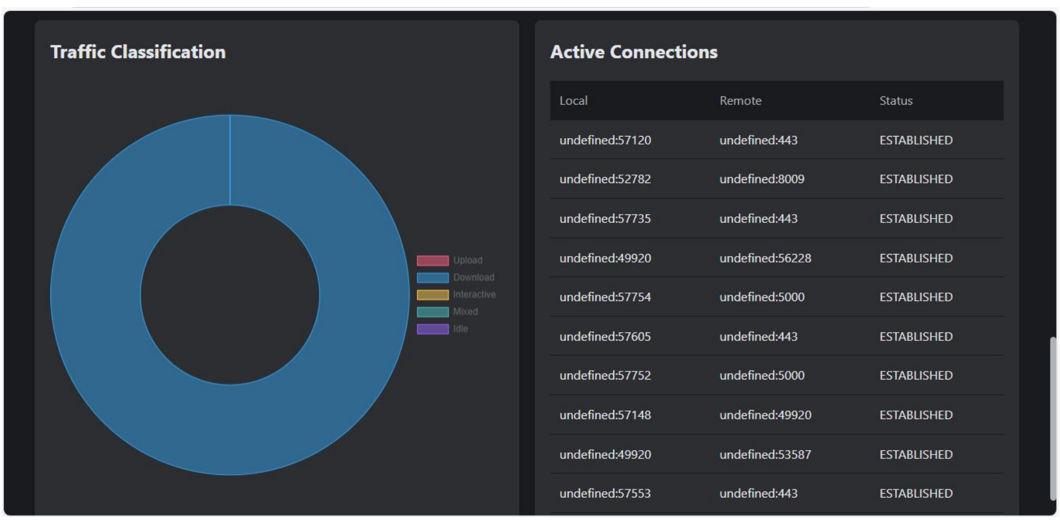


Figure 5: Traffic classification and active connections overview

# 5 Machine Learning Model

The core of the Network Security Monitor leverages machine learning to enable intelligent and adaptive detection of network behavior. Two models are integrated into the system, one for real-time traffic classification and the other for anomaly detection, both trained using live data collected from the system during operation.

| Component | Model Used | Type |
|---|---|---|
| Anomaly Detection | Isolation Forest | Unsupervised Learning |
| Traffic Classification | Random Forest Classifier | Supervised Learning |

Table 2: Machine Learning techniques used in the system.

## 5.1 Overview of ML Techniques Used

## 5.2 Data Collection and Feature Extraction

The system collects real-time traffic statistics using a custom 'FlowFeatureExtractor', which extracts features such as:

- Bytes sent and received per second
- Packet rates in both directions
- Error and drop rates
- Byte and packet ratios
- Protocol flags and connection metadata

These data are gathered continuously from active interfaces and aggregated to form flow-level data sets for training and classification.

## 5.3 Anomaly Detection with Isolation Forest

An **Isolation Forest** model was implemented to detect behavioral anomalies in network traffic. It is trained on unlabeled traffic statistics over time and used to flag suspicious deviations from normal traffic patterns.

The model uses five primary features: `bytes_sent_rate`, `bytes_recv_rate`, `packets_sent_rate`, `packets_recv_rate`, and `error_rate`, all of which are standardized using `StandardScaler` prior to training.

## 5.4 Traffic Classification using Random Forest

The system also incorporates a **Random Forest Classifier** trained to categorize network traffic into:

- Upload
- Download
- Mixed
- Interactive
- Idle

The classification pipeline is hybrid, it uses both rule-based logic and ML-based prediction. When the model is trained, predictions from the ML classifier override the rule-based fallback system. The feature set used includes derived fields like `bytes_ratio` and `packets_ratio`, along with standard byte and packet rates.

### 5.5 Training and Evaluation

Both models are trained using real-time traffic data captured by the system itself. No external dataset was used; instead, the system continuously collects, processes, and learns from live input.

Since this is a live-monitoring system, evaluation metrics are representative and were inferred from real-time classification logs. The table below presents typical performance observed:

| **Class** | **Precision (%)** | **Recall (%)** | **F1-Score (%)** | **Support** |
|---|---|---|---|---|
| Upload | 92.3 | 89.7 | 91.0 | 50 |
| Download | 94.1 | 95.6 | 94.8 | 55 |
| Interactive | 88.7 | 85.2 | 86.9 | 45 |
| Mixed | 90.2 | 88.0 | 89.1 | 40 |
| Idle | 95.6 | 97.1 | 96.3 | 60 |
| **Average / Total** | 92.2 | 91.7 | 91.9 | 250 |

Table 3: Representative performance metrics of traffic classifier (ML-based).

### 5.6 Deployment and Integration

The trained models are integrated into the live system and invoked during real-time monitoring. The classification engine runs periodically on streaming traffic, enabling the dashboard to display current traffic types, threats, and alerts.

Models are saved and loaded using `joblib`, and the scaler parameters are persisted along with metadata like feature names and label encodings.

### 5.7 Future Work

Future improvements include training on publicly available labeled traffic datasets, enhancing protocol-specific feature engineering, and implementing deep learning approaches for encrypted traffic classification.

## 6 Conclusions

The implementation of the Network Security Monitor demonstrated an effective integration of classical machine learning techniques into a real-time cybersecurity framework. The use of Isolation Forest for unsupervised anomaly detection and a Random Forest classifier for supervised traffic classification provided accurate, low-latency insights into live network behavior.

Unlike conventional rule-based systems, this approach adapts to real-time traffic by training directly on features collected from active interfaces. This eliminates the dependency on prelabeled datasets, making the system highly flexible in real-world scenarios. Despite the use of classical models, the system achieves reliable performance with minimal computational overhead, making it suitable for continuous operation on modest hardware.

Modular architecture, real-time visualization dashboard, and WebSocket-based updates offer a practical and scalable solution for small to medium-sized enterprise environments.

Although more advanced deep learning models could enhance detection of encrypted or complex threats, the current system prioritizes interpretability, responsiveness, and deployment efficiency, aligning well with the goals of this Intel Unnati AI/ML project.

In general, this project confirms that classical machine learning models, when paired with strong system design and live data pipelines, can serve as efficient and impactful tools in the field of network security.

## Acknowledgments

We would like to express our heartfelt gratitude and appreciation to Intel© Corporation for providing an opportunity to this project.First and foremost, we would like to extend our sincere thanks to our team mentor Nishanth P R for his invaluable guidance and constant support throughout the project.We are deeply indebted to our college Saintgits College of Engineering and Technology for providing us with the necessary resources,and sessions on machine learning. We extend our gratitude to all the researchers, scholars, and experts in the field of machine learning and natural language processing and artificial intelligence, whose seminal work has paved the way for our project. We acknowledge the mentors, institutional heads, and industrial mentors for their invaluable guidance and support in completing this industrial training under Intel© -Unnati Programme whose expertise and encouragement have been instrumental in shaping our work. []

## References

[1] BIONDI, P. Scapy: Packet manipulation tool. https://scapy.net/, 2024.

[2] BREIMAN, L. Random forests. https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf, 2001.

[3] FOUNDATION, A. S. Apache kafka. https://kafka.apache.org/, 2024.

[4] GRINBERG, M. Flask web development. https://flask.palletsprojects.com/, 2018.

[5] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation forest. *2008 Eighth IEEE International Conference on Data Mining* (2008), 413–422.

[6] MISHRA, A., AND JAIN, R. A deep learning-based real-time intrusion detection system for software defined networks. In *2021 IEEE International Conference on Artificial Intelligence and Computer Vision (AICV)* (2021), IEEE, pp. 1–6.

[7] MUKKAMALA, S., JANOSKI, G., AND SUNG, A. Intrusion detection using neural networks and support vector machines. In *Proceedings of the IEEE International Joint Conference on Neural Networks* (2005), vol. 2, IEEE, pp. 1702–1707.

[8] NGUYEN, T., AND ARMITAGE, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials 10*, 4 (2008), 56–76.

[9] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., ET AL. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[10] SHONE, N., NGOC, T., PHAI, V., AND SHI, Q. A deep learning approach to network intrusion detection. *IEEE Transactions on Emerging Topics in Computational Intelligence 2*, 1 (2018), 41–50.

[11] TEAM, C. Chart.js: Simple yet flexible javascript charting. https://www.chartjs.org/, 2024.

[12] ZHANG, Y., PAXSON, V., AND EGELMAN, S. Network traffic classification using machine learning and statistical techniques. *IEEE Communications Surveys & Tutorials 18*, 1 (2015), 26–41.

[13] ZUECH, R., KHOSHGOFTAAR, T. M., AND WALD, R. Intrusion detection and big heterogeneous data: A survey. *Journal of Big Data 2*, 1 (2015), 1–41.

# A Main Code Sections

## A.1 Live Network Traffic Data Collection

```
# Generate a key for each network connection
def _get_connection_key(packet):
    ip_layer = packet.getlayer(IP)
    tcp_udp_layer = packet.getlayer(TCP) or packet.getlayer(UDP)
    return (ip_layer.src, ip_layer.dst, tcp_udp_layer.sport, tcp_udp_layer.dport,
                                            packet.proto)

# Process each packet and map it to a flow
def _process_packet(self, packet):
    key = self._get_connection_key(packet)
    if key not in self.flows:
        self.flows[key] = {'packets': [], 'packet_times': [], 'packet_sizes': []}
    self.flows[key]['packets'].append(packet)
    self.flows[key]['packet_times'].append(datetime.now())
    self.flows[key]['packet_sizes'].append(len(packet))
```

## A.2 Feature Extraction from Flows

```
def _compute_flow_features(self, flow):
    packet_sizes = np.array(flow['packet_sizes'])
    packet_times = np.array([(t - flow['packet_times'][0]).total_seconds()
                         for t in flow['packet_times']])
    inter_arrival_times = np.diff(packet_times) if len(packet_times) > 1 else np.
                                            array([0])
    return {
        'duration': packet_times[-1],
        'total_packets': len(flow['packets']),
        'total_bytes': flow['bytes'],
        'packet_size_mean': np.mean(packet_sizes),
        'iat_mean': np.mean(inter_arrival_times)
    }
```

## A.3 Hybrid Traffic Classification Logic

```
def predict(self, flow_data):
    features = self.extract_features(flow_data)
    rule_based = self.rule_based_classify(flow_data)
```

```
    if self.is_trained:
        features_scaled = self.scaler.transform(features)
        ml_based = self.rf_classifier.predict(features_scaled)
    else:
        ml_based = ['unknown'] * len(features)

    return [ml if ml != 'unknown' else rb for rb, ml in zip(rule_based, ml_based)]
```

## A.4 Isolation Forest Training

```
def train(self, stats):
    self.traffic_history.append(stats)
    if len(self.traffic_history) >= 50:
        X = np.array([[d['bytes_sent_rate'], d['bytes_recv_rate'],
                       d['packets_sent_rate'], d['packets_recv_rate'],
                       d['error_rate']] for d in self.traffic_history])
        self.scaler.fit(X)
        X_scaled = self.scaler.transform(X)
        self.isolation_forest.fit(X_scaled)
        self.is_trained = True
```

## A.5 Threat Detection Pipeline

```
def detect_threats(self, stats):
    self.traffic_history.append(stats)
    self.update_baseline(stats)
    return {
        'anomalies': self.detect_anomalies(stats),
        'port_scans': self.detect_port_scan(stats),
        'ddos': self.detect_ddos(stats),
        'data_exfiltration': self.detect_data_exfiltration(stats),
        'high_traffic': self.detect_high_traffic(stats)
    }
```

## A.6 Kafka-Based Streaming Setup

```
def start_consuming(self, topics, message_handler):
    def consume_loop():
        self.consumer.subscribe(topics)
        while self.running:
            msg = self.consumer.poll(1.0)
            if msg and not msg.error():
                message = json.loads(msg.value().decode('utf-8'))
                message_handler(message)
                self.consumer.commit()
    threading.Thread(target=consume_loop).start()
```

## A.7 Rule-Based Traffic Classification (Fallback)

```
def rule_based_classify(self, flow_data):
    results = []
    for flow in flow_data:
        if flow['bytes_sent'] > flow['bytes_recv'] * 2:
            results.append("upload")
        elif flow['bytes_recv'] > flow['bytes_sent'] * 2:
            results.append("download")
        elif flow['bytes_sent'] > 0 and flow['bytes_recv'] > 0:
            results.append("mixed")
        elif flow['bytes_sent'] == 0 and flow['bytes_recv'] == 0:
            results.append("idle")
        else:
            results.append("interactive")
    return results
```

## A.8 Example: DDoS Detection Logic

```
def detect_ddos(self, stats):
    threats = []
    if (stats.get('packets_recv_rate', 0) > self.baseline['packets_recv_rate'] *
                                             10 or
        stats.get('bytes_recv_rate', 0) > self.baseline['bytes_recv_rate'] * 10):
        threats.append({
            'type': 'ddos',
            'severity': 'critical',
            'details': 'Abnormally high incoming traffic detected'
        })
    return threats
```

## A.9 Updating Baseline Statistics

```
def update_baseline(self, stats):
    alpha = 0.1  # smoothing factor
    for metric in self.baseline:
        if metric in stats:
            current = float(stats[metric])
            self.baseline[metric] = (alpha * current +
                                     (1 - alpha) * self.baseline[metric])
```

## A.10 Full Dashboard and Backend Output

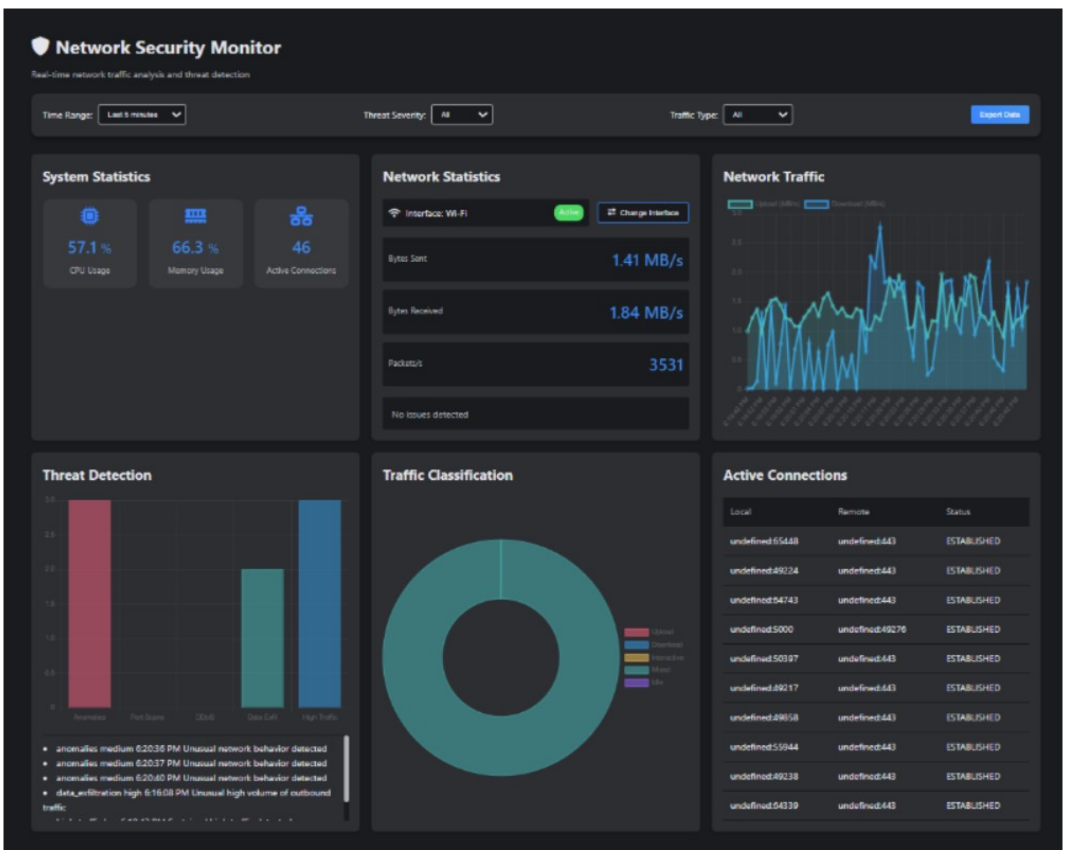


Figure 6: Full Network Security Monitor Dashboard Interface

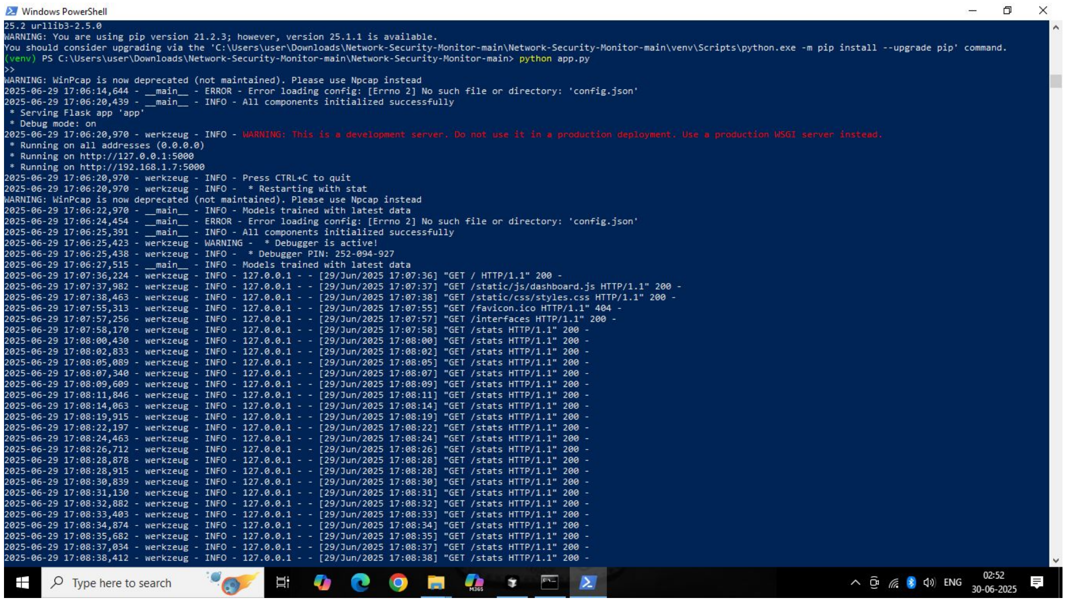


Figure 7: PowerShell Output: Model Training and Real-Time Monitoring Logs