

NMS(Network monitor system): Computer Network Monitor and Security Incidents Prevention System

Negura Teodor-Alexandru

University “Alexandru Ioan Cuza” of Iași,
Faculty of Computer Science

Abstract. This paper presents the design and implementation of NMS, a modular platform for monitoring computer networks and preventing security incidents. The system employs a three-tier architecture consisting of lightweight endpoint agents, client hubs with native graphical interfaces, and a centralised server performing machine learning-based anomaly detection using ONNX Runtime and LogBERT. The platform supports both custom agents and third-party syslog sources (RFC 5424), enabling integration with existing network infrastructure. Each client operates as an autonomous monitoring hub for its local network while maintaining privacy isolation at the server level. The architecture emphasises high performance through C++ implementation, secure communication via TLS, and real-time threat visualisation through Dear ImGui dashboards. This work demonstrates how modern machine learning techniques can be integrated into traditional network monitoring systems while maintaining modularity, extensibility, and strict privacy boundaries between multiple users.

Keywords: Network monitoring · Security incidents · Syslog RFC 5424 · ONNX Runtime · LogBERT · C++ · Client–Server · Anomaly detection

1 Introduction

The proliferation of connected devices and the increasing sophistication of cyber threats have made network monitoring an essential component of modern IT infrastructure. Traditional approaches relying on signature-based detection and manual log analysis are no longer sufficient to address zero-day exploits, advanced persistent threats, and the sheer volume of security events generated by contemporary networks.

The goal of this project is to build a state-of-the-art network monitoring and security incident prevention platform based on a three-tier architecture. The system addresses the problem of centralising network intelligence while respecting privacy boundaries between different users and networks. Unlike centralised solutions where all data flows to a single administrator, NMS enables each client to monitor their own network infrastructure independently, with the central server

providing computational resources for machine learning inference without exposing data across client boundaries.

The platform operates on the following principles. First, each client acts as a hub for their local network, collecting events from both custom agents and standard syslog sources. Second, the central server performs heavy computational tasks (ONNX/LogBERT inference) and returns only the results relevant to each client. Third, privacy isolation ensures that Client A cannot access data belonging to Client B, enforced through public IP identification. Fourth, modularity allows new data sources, agents, and analysis components to be added without redesigning the system.

A key innovation is the integration of LogBERT, a transformer-based language model specifically trained for log anomaly detection, executed through ONNX Runtime for high-performance C++ inference. This enables the detection of subtle attack patterns that would escape traditional rule-based systems.

2 Applied Technologies

The technology stack was carefully selected to satisfy functional requirements (network monitoring, anomaly detection, multi-user support) and non-functional requirements (performance, security, modularity, ease of deployment).

C++20 serves as the implementation language for all three components: server, client, and agent. C++ provides deterministic memory management, zero-cost abstractions, and direct access to system APIs essential for network programming and syslog collection. The use of modern C++20 features (concepts, ranges, coroutines where appropriate) ensures code clarity without sacrificing performance.

TCP with TLS (OpenSSL) provides the transport layer for client–server communication. TCP guarantees ordered, reliable delivery of monitoring events, while TLS 1.3 encryption protects against eavesdropping and man-in-the-middle attacks. The OpenSSL library, compiled statically into the binaries, handles certificate validation and key exchange.

Syslog (RFC 5424) defines the standard format for log messages. The platform natively accepts syslog from any compliant source on UDP/TCP port 514, enabling integration with third-party software such as rsyslog, syslog-ng, Filebeat, and network equipment (routers, firewalls, switches). This satisfies the requirement for testing with third-party agents.

JSON with Length-Prefix Framing serves as the application-level message format. JSON provides human-readable, self-describing messages that simplify debugging and Wireshark analysis. Each message is prefixed with a 4-byte length field (network byte order) to enable efficient parsing over the TCP stream without delimiter scanning.

ONNX Runtime executes the pre-trained LogBERT anomaly detection model. ONNX (Open Neural Network Exchange) provides a framework-agnostic format for machine learning models, while ONNX Runtime offers optimised C++

inference with optional CUDA acceleration. This allows the server to classify thousands of log events per second.

LogBERT is a transformer-based model specifically designed for log anomaly detection. Pre-trained on large corpora of system logs, LogBERT understands the semantic structure of log messages and can identify anomalies that deviate from learned patterns, including novel attack vectors not covered by traditional signatures.

SQLite provides lightweight, embedded database functionality. On the client side, SQLite stores file integrity baselines, agent configuration, and cached results from the server. On the server side, SQLite stores processed events and alerts, partitioned by client public IP for privacy isolation.

Dear ImGui powers the native graphical dashboard on the client. Unlike web-based interfaces requiring REST APIs and browser dependencies, Dear ImGui renders directly through OpenGL, providing immediate-mode GUI with minimal latency. The dashboard displays real-time alerts, endpoint status, statistics, and log streams.

Docker containerises the server component, encapsulating ONNX Runtime, the LogBERT model, OpenSSL, and all dependencies into a reproducible deployment unit. Clients and agents run as native binaries without containerisation to minimise resource overhead on endpoints.

nlohmann/json is a header-only C++ library for JSON parsing and serialisation. Its intuitive API and excellent performance make it the standard choice for modern C++ projects requiring JSON support.

3 Application Structure

The system comprises three distinct components, each compiled as a separate binary from a shared CMake codebase. Figure 1 illustrates the high-level architecture.

3.1 Server Component

The server runs inside a Docker container on the operator’s infrastructure (for this thesis, the author’s machine). It accepts TLS connections from multiple clients on port 1514 and performs the following functions:

- **Connection Management:** A dedicated listener thread accepts incoming TLS connections, extracts the client’s public IP address for identification, and dispatches each connection to a worker thread from the pool.
- **Protocol Handling:** Worker threads parse incoming JSON messages, validate their structure, and route them to the appropriate processing pipeline based on message type (REGISTER, BATCH_EVENT, HEARTBEAT).
- **Event Batching:** Incoming events are accumulated into batches of 32–64 entries per client before being forwarded to the inference pipeline, optimising throughput for the LogBERT model.

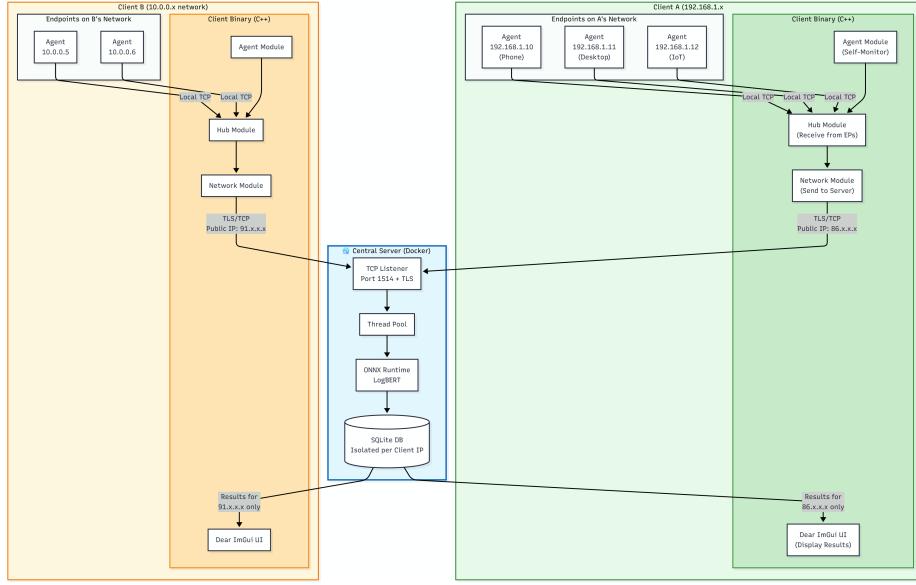


Fig. 1. High-level architecture of NMS showing the three-tier structure: Server, Clients, and Endpoint Agents.

- **ONNX Inference:** A dedicated inference thread tokenises batched log messages using the LogBERT vocabulary and executes the ONNX model to classify each event as normal or anomalous.
- **Storage:** Processed events and detected anomalies are stored in SQLite, keyed by the client’s public IP address. This ensures strict privacy isolation—queries for Client A’s data never return Client B’s records.
- **Response Generation:** Results (alerts, statistics) are serialised as JSON and transmitted back to the originating client over the same TLS connection.

The server does not include a graphical interface; all visualisation occurs on the client side.

3.2 Client Component

The client is the central piece of the architecture from the user’s perspective. Each user runs one client instance on their primary machine (e.g., laptop), which serves as the hub for their entire local network. The client binary integrates four modules:

- **Agent Module:** Monitors the local machine itself, collecting syslog entries from `/dev/log` (Linux) or the Event Log (Windows), tracking file integrity through cryptographic hashes stored in SQLite, and observing process creation and network connections.

- **Hub Module:** Listens for incoming connections from endpoint agents within the local network. Two listener ports are exposed:
 - *Port 1515 (TCP):* Accepts connections from custom NMS agents using the JSON protocol with length-prefix framing.
 - *Port 514 (UDP/TCP):* Accepts standard syslog messages (RFC 5424) from third-party sources such as rsyslog, routers, firewalls, and other network equipment.
- **Network Module:** Maintains a persistent TLS connection to the central server, batches collected events (both self-generated and received from endpoints), transmits them to the server, and receives processed results.
- **UI Module:** Renders the Dear ImGui dashboard, displaying real-time alerts, endpoint status, event statistics, and live log streams. The UI reads directly from a local cache populated by responses from the server, requiring no REST API or web server.

Figure 2 shows the internal structure of the client binary.

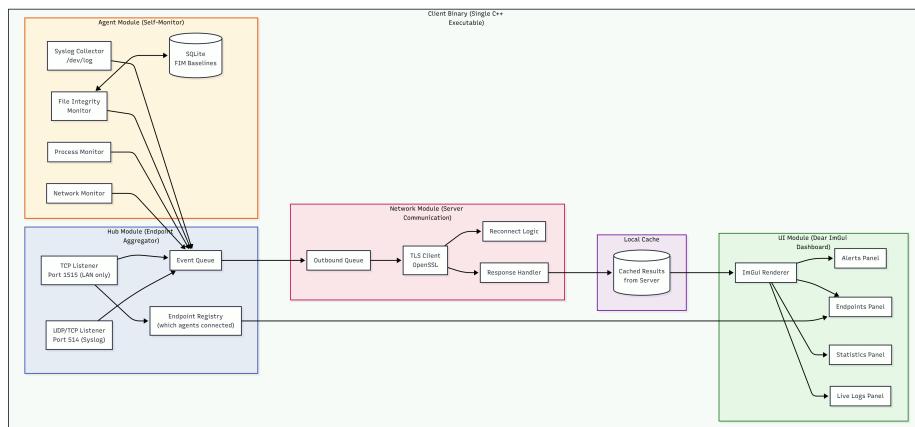


Fig. 2. Internal architecture of the client binary showing the four integrated modules.

3.3 Agent Component

The agent is a lightweight binary designed for deployment on endpoint devices within the client’s network (desktops, servers, IoT devices). Its sole responsibility is collecting local events and forwarding them to the client hub. The agent has no graphical interface and minimal resource footprint.

Key characteristics of the agent include static linking to eliminate runtime dependencies, a single configuration file specifying the hub address and port, automatic reconnection logic with exponential backoff, and local buffering in SQLite when the hub is temporarily unreachable.

The agent communicates with the client hub over plain TCP (port 1515) using the JSON protocol. TLS is not used for agent–hub communication because both parties reside on the same local network, and the overhead would be unnecessary for constrained devices.

3.4 Third-Party Integration

A critical requirement is the ability to receive logs from third-party software without installing custom agents. This is achieved through the standard syslog interface (port 514). Any device or software capable of emitting RFC 5424 syslog can send events to the client hub.

Example configurations for common third-party sources:

rsyslog (Linux): Add to `/etc/rsyslog.conf`:

```
1 *.* @@192.168.1.100:514      # TCP
2 *.* @192.168.1.100:514      # UDP
```

Network Router: Configure in the administration panel:

```
1 Syslog Server: 192.168.1.100
2 Syslog Port: 514
3 Protocol: UDP
```

This dual-input architecture (custom agents on port 1515, standard syslog on port 514) ensures maximum flexibility and compatibility with existing infrastructure.

3.5 Data Flow Summary

The complete data flow from endpoint to dashboard proceeds as follows:

1. Endpoint agent collects syslog entries and local events.
2. Agent serialises events as JSON and sends to client hub (port 1515).
3. Third-party sources send RFC 5424 syslog to client hub (port 514).
4. Client hub aggregates events from all sources, including self-monitoring.
5. Client batches events and transmits to server over TLS (port 1514).
6. Server identifies client by public IP, parses events, runs ONNX inference.
7. Server stores results in SQLite (partitioned by client IP).
8. Server returns alerts and statistics to client.
9. Client caches results and renders in Dear ImGui dashboard.

Figure 3 illustrates this pipeline.

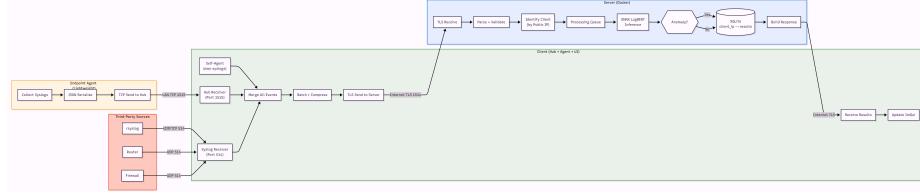


Fig. 3. Data flow from endpoint agents through the client hub to the server and back to the client UI.

3.6 Privacy Isolation Model

Multiple clients can connect to the same server simultaneously, each monitoring their own network. Privacy is enforced through public IP identification: the server extracts the source IP from each TLS connection and uses it as a partition key for all database operations.

When Client A (public IP 86.123.45.67) requests their data, the server executes:

```

1 SELECT * FROM events WHERE client_ip = '86.123.45.67';
2 SELECT * FROM alerts WHERE client_ip = '86.123.45.67';

```

Client B (public IP 91.200.10.20) receives only their own records, with no visibility into Client A's data. This simple yet effective mechanism provides baseline privacy isolation suitable for the thesis demonstration. Future enhancements could add authentication tokens for stronger identity verification.

3.7 Deployment Architecture

Table 1 summarises the deployment model for each component.

Table 1. Deployment architecture for NMS components.

Component Build	Deployment	Dependencies
Server	CMake → C++ Docker container	ONNX Runtime, OpenSSL, SQLite
Client	CMake → C++ Native binary	OpenSSL, SQLite, Dear ImGui, OpenGL
Agent	CMake → C++ Native binary (static)	SQLite (embedded)

4 Communication Protocol

Communication occurs at two levels: agent-to-hub (LAN) and client-to-server (WAN). Both use JSON as the message format but differ in transport characteristics.

4.1 Message Framing

All JSON messages are transmitted using length-prefix framing to enable efficient parsing over TCP streams. Each message consists of:

- **Length Field:** 4 bytes, unsigned 32-bit integer in network byte order (big-endian), specifying the length of the following JSON payload in bytes.
- **Payload:** Variable-length UTF-8 encoded JSON string.

This approach avoids the need to scan for delimiter characters and handles JSON strings containing newlines correctly.

Listing 1.1. Message frame structure.

1	+-----+-----+
2	4 bytes N bytes
3	Length (N) JSON Payload
4	(big-endian) {"type": "EVENT", ...}
5	+-----+-----+

4.2 Message Types

The protocol defines six message types, each identified by the `type` field:

Listing 1.2. Generic message schema.

```
1 {  
2   "type": "REGISTER" | "HEARTBEAT" | "BATCH_EVENT" |  
3   "RESULTS" | "COMMAND" | "ACK",  
4   "timestamp": "2025-11-28T14:30:00Z",  
5   "payload": { ... }  
6 }
```

REGISTER: Sent by the client upon initial connection to the server, or by an agent upon connection to the hub. Contains identification information.

Listing 1.3. REGISTER message from client to server.

```
1 {  
2   "type": "REGISTER",  
3   "timestamp": "2025-11-28T14:30:00Z",  
4   "payload": {  
5     "client_version": "1.0.0",  
6     "hostname": "user-laptop",  
7     "os": "Linux 6.1.0",  
8     "endpoints_count": 3  
9   }  
10 }
```

HEARTBEAT: Sent periodically (default: every 30 seconds) to indicate liveness and provide status metrics.

Listing 1.4. HEARTBEAT message.

```
1 {
2     "type": "HEARTBEAT",
3     "timestamp": "2025-11-28T14:31:00Z",
4     "payload": {
5         "uptime_seconds": 3600,
6         "events_queued": 12,
7         "endpoints_connected": 3
8     }
9 }
```

BATCH_EVENT: Transmitted by the client to the server containing aggregated events from all sources (self-agent, custom agents, third-party syslog).

Listing 1.5. BATCH_EVENT message with multiple events.

```
1 {
2     "type": "BATCH_EVENT",
3     "timestamp": "2025-11-28T14:30:05Z",
4     "payload": [
5         "events": [
6             {
7                 "source": "self",
8                 "syslog": {
9                     "facility": 4,
10                    "severity": 5,
11                    "app": "sshd",
12                    "pid": 12345,
13                    "message": "Failed password for root from
14                      10.0.0.50"
15                }
16            },
17            {
18                "source": "192.168.1.101",
19                "syslog": {
20                    "facility": 10,
21                    "severity": 6,
22                    "app": "kernel",
23                    "pid": 0,
24                    "message": "TCP: request_sock_TCP: Possible SYN
25                      flooding"
26                }
27            }
28        ]
29    }
30 }
```

RESULTS: Sent by the server to the client containing processed alerts and statistics.

Listing 1.6. RESULTS message from server.

```
1 {
2     "type": "RESULTS",
3     "timestamp": "2025-11-28T14:30:06Z",
4     "payload": [
5         "alerts": [
6             {
7                 "id": "alert-001",
8                 "severity": "HIGH",
9                 "source": "192.168.1.101",
10                "category": "brute_force",
11                "message": "Multiple failed SSH attempts detected",
12                "confidence": 0.94
13            }
14        ],
15        "stats": [
16            "events_processed": 64,
17            "anomalies_detected": 1,
18            "processing_time_ms": 45
19        ]
20    }
21 }
```

COMMAND: Initiated by the server (or through future administrative interface) to request an action on the client or a specific endpoint.

Listing 1.7. COMMAND message.

```
1 {
2     "type": "COMMAND",
3     "timestamp": "2025-11-28T14:32:00Z",
4     "payload": [
5         "command_id": "cmd-123",
6         "action": "request_fim_report",
7         "target": "192.168.1.101"
8     ]
9 }
```

ACK: Acknowledgement confirming receipt and processing of REGISTER or COMMAND messages.

Listing 1.8. ACK message.

```
1 {
2     "type": "ACK",
3     "timestamp": "2025-11-28T14:30:01Z",
4     "payload": [
5         "status": "ok",
6         "message": "Registration successful",
7         "config": [
8             "heartbeat_interval": 30,
```

```

9     "batch_size": 64
10    }
11  }
12 }
```

4.3 Protocol State Machine

Figure 4 shows the state transitions for a client connection.

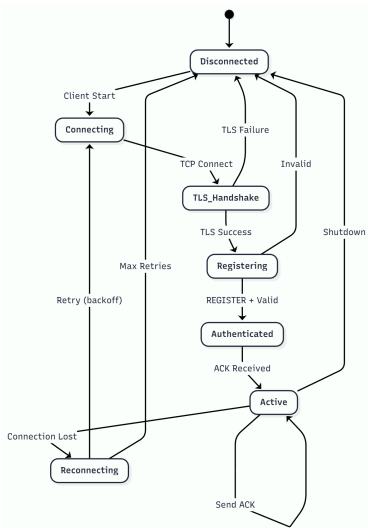


Fig. 4. State machine for client–server connection lifecycle.

The connection lifecycle proceeds as follows:

1. Client initiates TCP connection to server on port 1514.
2. TLS handshake establishes encrypted channel.
3. Client sends REGISTER message with identification.
4. Server validates and responds with ACK containing configuration.
5. Connection enters Active state; client sends HEARTBEAT and BATCH_EVENT.
6. Server sends RESULTS after processing batches.
7. On connection loss, client enters Reconnecting state with exponential back-off.

4.4 Port Assignments

Table 2 summarises the network ports used by the platform.

Table 2. Network port assignments.

Port	Protocol	Direction	Encryption	Purpose
1514	TCP	Client → Server	TLS 1.3	Primary communication
1515	TCP	Agent → Client Hub	None (LAN)	Custom agent protocol
514	UDP/TCP	Third-party → Client Hub	None (LAN)	Standard syslog

5 Real World Use Cases

This section presents six scenarios demonstrating the platform’s capabilities: three successful executions and three failure handling cases.

5.1 Successful Scenarios

Scenario 1: Brute-Force SSH Detection A user deploys NMS on their home network. The client runs on their laptop, and agents are installed on a desktop PC and a Raspberry Pi server.

An attacker on the internet attempts to brute-force the SSH service on the Raspberry Pi, which is exposed through port forwarding. The agent on the Pi collects authentication failure events from `/var/log/auth.log` and forwards them to the client hub. The client batches these events and transmits them to the server.

The LogBERT model, having been trained on patterns of authentication attacks, identifies the sequence of “Failed password” messages as anomalous. The server returns a HIGH severity alert to the client, which displays it prominently in the ImGui dashboard. The user sees the alert within seconds of the attack beginning and can take action (block the IP at the router, disable password authentication).

Scenario 2: Third-Party Router Integration A small business owner wants to monitor their network without installing custom software on every device. They configure their enterprise router to send syslog to the NMS client running on their workstation.

The router sends RFC 5424 messages to port 514 (UDP) whenever significant events occur: firewall blocks, DHCP leases, VPN connections. The client hub receives these standard syslog messages, parses them according to RFC 5424, normalises them into the internal JSON format, and forwards them to the server.

The LogBERT model detects an unusual pattern: repeated firewall blocks from an internal IP address attempting to reach external command-and-control servers. This indicates a potentially compromised device. The alert appears on the dashboard with the source IP, allowing the owner to isolate the device for investigation.

Scenario 3: File Integrity Monitoring A developer uses NMS to monitor their web server. The agent is configured with a list of critical files: `/etc/passwd`, `/etc/shadow`, `nginx.conf`, and the web application directory.

On initial startup, the agent computes SHA-256 hashes of all monitored files and stores them in local SQLite. Every 5 minutes, it recomputes hashes and compares against the baseline. When the developer legitimately updates `nginx.conf`, the agent detects the change and sends an EVENT to the hub.

The server’s LogBERT model considers the context: a configuration file changed during business hours from an authorised user session. The confidence score for malicious activity is low (0.12). The event is logged but not flagged as a high-priority alert.

Later, an attacker exploits a vulnerability and modifies `/etc/passwd` at 3:00 AM. The agent detects this change. LogBERT, trained to recognise suspicious patterns (system file modification, unusual hour, no preceding legitimate administrative activity), assigns a high anomaly score (0.97). A CRITICAL alert is generated immediately.

5.2 Failure Scenarios

Scenario 4: Network Disconnection Handling A user is monitoring their home network when their internet connection fails. The client loses its TLS connection to the server.

The client’s network module detects the connection loss and enters the Reconnecting state. It attempts to reconnect with exponential backoff (1s, 2s, 4s, 8s, up to 5 minutes maximum). During this period, the hub continues to receive events from local agents and third-party sources.

Events are queued in a local buffer (up to 10,000 entries). The UI displays a “Server Disconnected” indicator but continues to show the last known state and locally collected events. When the connection is restored, queued events are transmitted in batches, ensuring no data loss.

Scenario 5: Malformed Message Rejection A misconfigured third-party device sends malformed syslog messages to the client hub—perhaps binary data or incorrectly formatted timestamps.

The hub’s syslog parser attempts to parse each message according to RFC 5424. When parsing fails, the message is logged locally (for debugging) but not forwarded to the server. A counter increments in the “Parse Errors” statistic displayed on the dashboard.

The system continues processing valid messages without interruption. The user can investigate the parse errors through the log viewer panel and identify the problematic source.

Scenario 6: Server Overload Protection During a large-scale attack (e.g., DDoS), multiple clients simultaneously send high volumes of events to the server. The server’s processing capacity is exceeded.

The server implements backpressure through its thread pool and bounded queue. When the queue reaches capacity, new batches are rejected with a “SERVER_BUSY” response. Clients receiving this response reduce their transmission rate (additive decrease) and retry after a delay.

Additionally, the server prioritises HEARTBEAT messages over BATCH_EVENT to maintain connection liveness. Clients that cannot transmit events queue them locally. When the load subsides, normal processing resumes without data loss (within buffer limits).

6 Conclusion

This work presents NMS, a modular client–server platform for network monitoring and security incident prevention. The system successfully addresses the requirements of centralised log collection, real-time anomaly detection, multi-user privacy isolation, and integration with third-party infrastructure.

The three-tier architecture (Server, Client, Agent) provides clear separation of concerns. The server handles computationally intensive ML inference without storing sensitive client data beyond what is necessary for processing. Clients act as autonomous monitoring hubs for their local networks, aggregating events from custom agents and standard syslog sources. Lightweight agents minimise resource consumption on monitored endpoints.

The integration of LogBERT through ONNX Runtime demonstrates that modern transformer-based anomaly detection can be deployed in production C++ systems with acceptable performance. The native Dear ImGui dashboard provides real-time visualisation without the complexity of web-based interfaces.

6.1 Potential Improvements

Several enhancements could extend the platform’s capabilities:

- **Authentication System:** Replace IP-based identification with token-based authentication, enabling users to access their data from different networks and supporting multiple networks per account.
- **Distributed Server Architecture:** Deploy multiple server instances behind a load balancer to handle larger client populations and provide high availability.
- **Enhanced ML Pipeline:** Add incremental model retraining based on user feedback (confirmed true/false positives), improving detection accuracy over time.
- **Elastic/OpenSearch Integration:** Optionally export processed events to Elastic Stack for advanced querying, long-term retention, and integration with existing SIEM workflows.
- **Mobile Dashboard:** Develop a companion mobile application for alert notifications and basic monitoring when away from the primary workstation.
- **Encrypted Agent Communication:** Add optional TLS for agent–hub communication in zero-trust network environments.

- **Windows Native Agent:** Extend agent support to Windows Event Log collection with native API integration.

The modular architecture ensures that these improvements can be implemented incrementally without redesigning the core platform.

References

1. Gerhards, R.: The Syslog Protocol. RFC 5424, Internet Engineering Task Force (IETF) (2009). <https://www.rfc-editor.org/rfc/rfc5424>
2. Stevens, W.R.: TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley (1994).
3. Stallings, W.: Network Security Essentials: Applications and Standards. 6th edn. Pearson (2017).
4. ONNX Project: Open Neural Network Exchange. <https://onnx.ai>. Accessed: 28 November 2025.
5. Microsoft: ONNX Runtime – High Performance ML Inferencing. <https://onnxruntime.ai>. Accessed: 28 November 2025.
6. Guo, H., Yuan, S., Wu, X.: LogBERT: Log Anomaly Detection via BERT. In: International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2021).
7. Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239) (2014).
8. SQLite Consortium: SQLite Documentation. <https://www.sqlite.org/docs.html>. Accessed: 28 November 2025.
9. OpenSSL Project: OpenSSL Cryptography and SSL/TLS Toolkit. <https://www.openssl.org>. Accessed: 28 November 2025.
10. Cornut, O.: Dear ImGui – Bloat-free Graphical User Interface for C++. <https://github.com/ocornut/imgui>. Accessed: 28 November 2025.
11. Wazuh, Inc.: Wazuh – The Open Source Security Platform. <https://wazuh.com>. Accessed: 28 November 2025.
12. Elastic N.V.: Elastic Security – SIEM, Endpoint Security, and Cloud Security. <https://www.elastic.co/security>. Accessed: 28 November 2025.
13. Lohmann, N.: JSON for Modern C++. <https://github.com/nlohmann/json>. Accessed: 28 November 2025.
14. Scarfone, K., Mell, P.: Guide to Intrusion Detection and Prevention Systems (IDPS). NIST Special Publication 800-94 (2007).
15. Chen, T., Guestrin, C.: XGBoost: A Scalable Tree Boosting System. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016).