CS4850/02 Fall 2025

SP 110 Blue Linux Anomaly

Noor Aissat, Geovanni Cuevas, John Fuentes, Katheryn Robles

Sharon Perry

12/7/2025

https://github.com/john-fuentes/LinuxAnomalyDetection

https://sp-110-blue-linux-anomaly.vercel.app/

STATS and STATUS	
LOC	6002
Components/Tools	11 Filebeat, Linux auditd, Apache Kafka, Flask, Next.js, Isolation Forest (scikit-learn), Elasticsearch, Supabase, Docker, SQLAlchemy, React
Hours Estimated	365
Hours Actual	284
Status	Project is 100% complete and working as designed based on requirements

TABLE OF CONTENTS

1. Introduction

This project focuses on building a full-stack cybersecurity monitoring platform that detects unusual user activity on Linux virtual machines by utilizing machine learning. The system collects audit log data from Linux hosts, analyzes the logs, and presents the results through a secure web dashboard for review by security analysts. The primary goal of the platform is to improve visibility into system activity and provide early detection of potentially malicious behavior before issues escalate into full security incidents.

Log data is generated by the Linux auditd service and forwarded via Filebeat into a Kafka streaming pipeline that ensures reliable, high-throughput ingestion. A Flask Kafka consumer aggregates these stream events, converts them into machine-learning compatible features, and applies an Isolation Forest anomaly detection model to determine whether observed behavior deviates significantly from established baselines. Detection results and raw log records are stored in Elasticsearch for fast querying and long-term analysis. A Flask backend exposes RESTful APIs that provide authenticated access to this stored data, and a Next.js application delivers an interactive visualization interface for end users. Supabase and SQLAlchemy are used to manage user accounts, role-based access control, and virtual machine ownership records.sumer aggregates these stream events, converts them into machine-learning compatible features, and applies an Isolation Forest anomaly detection model to determine whether observed behavior deviates significantly from established baselines. Detection results and raw log records are stored in Elasticsearch for fast querying and long-term analysis. A Flask backend exposes RESTful APIs that provide authenticated access to this stored data, and a Next.js application delivers an interactive visualization interface for end users. Supabase and SQLAlchemy are used to manage user accounts, role-based access control, and virtual machine ownership records.

This project integrates multiple distributed technologies into a cohesive monitoring system capable of processing logs at scale while offering accessible tools for investigation and reporting. The system emphasizes modularity, scalability, and maintainability, enabling each major subsystem such as log ingestion, machine learning detection, database storage, backend APIs, and frontend visualization to evolve independently. Ultimately, the platform serves as a practical demonstration of how streaming pipelines and applied machine learning can be combined to enhance Linux host security monitoring in real-world environments.

2. Challenges and Assumptions

2.1 Performance and Scalability Challenges

One of the primary challenges in developing this system was designing a data pipeline capable of handling large volumes of audit logs without introducing noticeable delays. Linux systems can generate thousands of audit events per second during normal operation, which requires reliable buffering and efficient downstream processing to ensure no events are lost. Kafka was selected to decouple log ingestion from log processing, enabling the system to scale horizontally as throughput increases. However, maintaining low latency requires careful tuning

of consumer workloads, partitioning strategies, and Elasticsearch indexing performance to prevent bottlenecks in either streaming or data storage operations.

2.2 Data Collection and Machine Learning Challenges

One of the most significant challenges encountered during development was obtaining sufficient volumes of realistic normal user activity data necessary for training the machine learning model. Since anomaly detection relies on learning baseline behavioral patterns, the system required datasets representing everyday system usage such as web browsing, file access, and standard login activity. Generating reliable baseline data proved difficult due to the limited availability of consistent user interaction data across virtual machines. To address this issue, the team manually simulated realistic user behavior patterns and captured audit logs during extended normal usage periods to build initial training datasets. Additional challenges included transforming semi-structured audit events into stable numerical features and ensuring consistent preprocessing so that the Isolation Forest model received valid and meaningful input representations.

2.3 Security and Reliability Challenges

Because Linux audit logs may contain sensitive details about system activity and user behavior, implementing strong access controls was an essential design requirement. Secure authentication through Google SSO and role-based authorization enforced by the backend minimized exposure risks. Reliable system operation posed another challenge due to the distributed nature of the architecture, where failures in any component could disrupt data flow. Implementing Kafka message retention and containerized service recovery mechanisms helped reduce these risks by allowing the system to resume processing without losing collected logs or detection results. could disrupt data flow. Implementing Kafka message retention and containerized service recovery mechanisms helped reduce these risks by allowing the system to resume processing without losing collected logs or detection results.

2.4 System Assumptions

This project assumes that all monitored Linux virtual machines are properly configured with auditd, Filebeat services and continuous log forwarding to Kafka remains reliable. It also assumes that adequate normal activity data exists, or can be generated, to maintain effective machine learning training baselines. Stable network communication, sufficient cloud resource provisioning, and standard cybersecurity knowledge among end users are also assumed to ensure proper system functionality and effective interpretation of anomaly detection results.

3. Requirements

3.1. Requirements Overview

This project involves developing a web-based cybersecurity monitoring platform designed to detect unusual user activity across Linux virtual machines. The system analyzes real audit logs generated by the Linux auditd service and applies machine learning techniques to identify anomalous events that may indicate security threats. The primary goal is to give security

analysts an easy-to-use tool for continuously monitoring host activity and reviewing suspicious behavior in real time or through historical analysis. security threats. The primary goal is to give security analysts an easy-to-use tool for continuously monitoring host activity and reviewing suspicious behavior in real time or through historical analysis.

Log data is collected directly from Linux hosts using auditd and forwarded by Filebeat into a Kafka streaming pipeline. A Flask backend consumes these logs, converts them into numerical feature vectors, and evaluates each event using an Isolation Forest anomaly detection model. Detection results are forwarded through Kafka and then stored alongside raw log records in Elasticsearch. A Flask-based backend exposes secure REST APIs that allow a Next.js frontend to present dashboards, log tables, and visual summaries to end users. User identity and access management are handled through Supabase using SQLAlchemy integration and Google Single Sign-On authentication. stored alongside raw log records in Elasticsearch. A Flask-based backend exposes secure REST APIs that allow a Next.js frontend to present dashboards, log tables, and visual summaries to end users. User identity and access management are handled through Supabase using SQLAlchemy integration and Google Single Sign-On authentication.

3.2. Project Goals

The primary goal of this project is to detect abnormal and potentially malicious user activity on Linux systems using machine learning. The system must support near real-time discovery and review of anomalies within the monitoring dashboard so that analysts can respond quickly to threats or suspicious behavior. It must authenticate users securely using Google Single Sign-On and email verification while enforcing role-based access controls for admins, analysts, and viewers. The system must work directly with real Linux audit data rather than simulated or synthetic logs, allowing it to establish a baseline of normal activity using historical records. a baseline of normal activity using historical records.

Another goal is to provide powerful visualization tools that help analysts explore detection results over time. This includes enabling users to filter anomaly results by date, virtual machine, or severity and visualize trends using line graphs and summary charts. Finally, the system must be designed with scalability in mind, allowing additional hosts, Kafka consumers, ML workers, and Elasticsearch capacity to be added without restructuring the pipeline.

3.3. Design Constraints

3.3.1 Environment

Linux virtual machines generate security audit logs using the auditd service. These logs are forwarded by Filebeat into Kafka running within Docker containers in a cloud environment. All supporting system components, including Kafka brokers, the Python ML microservice, Flask backend APIs, Elasticsearch databases, and the Next.js frontend, operate within containerized services orchestrated through Docker Compose. Primary deployment occurs on Digital Ocean Droplets. Sufficient networking bandwidth and compute capacity must be provisioned to ensure the uninterrupted streaming and processing of log events under peak loads.forwarded by Filebeat into Kafka running within Docker containers in a cloud environment. All supporting system components, including Kafka brokers, the Python ML microservice, Flask backend APIs, Elasticsearch databases, and the Next.js frontend, operate within containerized services orchestrated through Docker Compose. Primary deployment occurs on Digital Ocean Droplets.

Sufficient networking bandwidth and compute capacity must be provisioned to ensure the uninterrupted streaming and processing of log events under peak loads.

3.3.2 User Characteristics

The platform is intended for use by security analysts, system administrators, and machine learning practitioners who are comfortable working with system logs and security dashboards. These users expect rapid data availability and simple filtering tools that allow them to quickly correlate anomalies across machines and time intervals. Although users are technically proficient, the interface design must remain intuitive and clean to ensure rapid investigation and minimal training requirements.

3.3.3 System Constraints

The system must process large volumes of Linux logs in near real time while maintaining reliability and fault tolerance. Kafka is required to remain highly available so that spikes in throughput do not cause data loss or permanent backlog. Elasticsearch must support rapid indexing, searching, and aggregation queries for both raw and processed data. Authentication services must enforce secure role-based access control. Compliance with security standards such as NIST guidelines and CIS benchmarks must be maintained due to the sensitive nature of collected data.

3.4. Functional Requirements

3.4.1 User Login

The system will require users to authenticate before accessing any anomaly detection data or dashboards. Users must be able to create accounts using Google Single Sign-On or verified email registration. The login page must guide users through the authentication process and display error messages when credentials are invalid. Secure session management will be implemented to maintain login persistence and support account sign out from any page. Once a user logs out, their session token must be invalidated immediately. The system also needs to provide recovery options, allowing users to reset credentials that are forgotten or lost.maintain login persistence and support account sign out from any page. Users must also have access to account recovery functionality in the event that credentials are forgotten or lost.

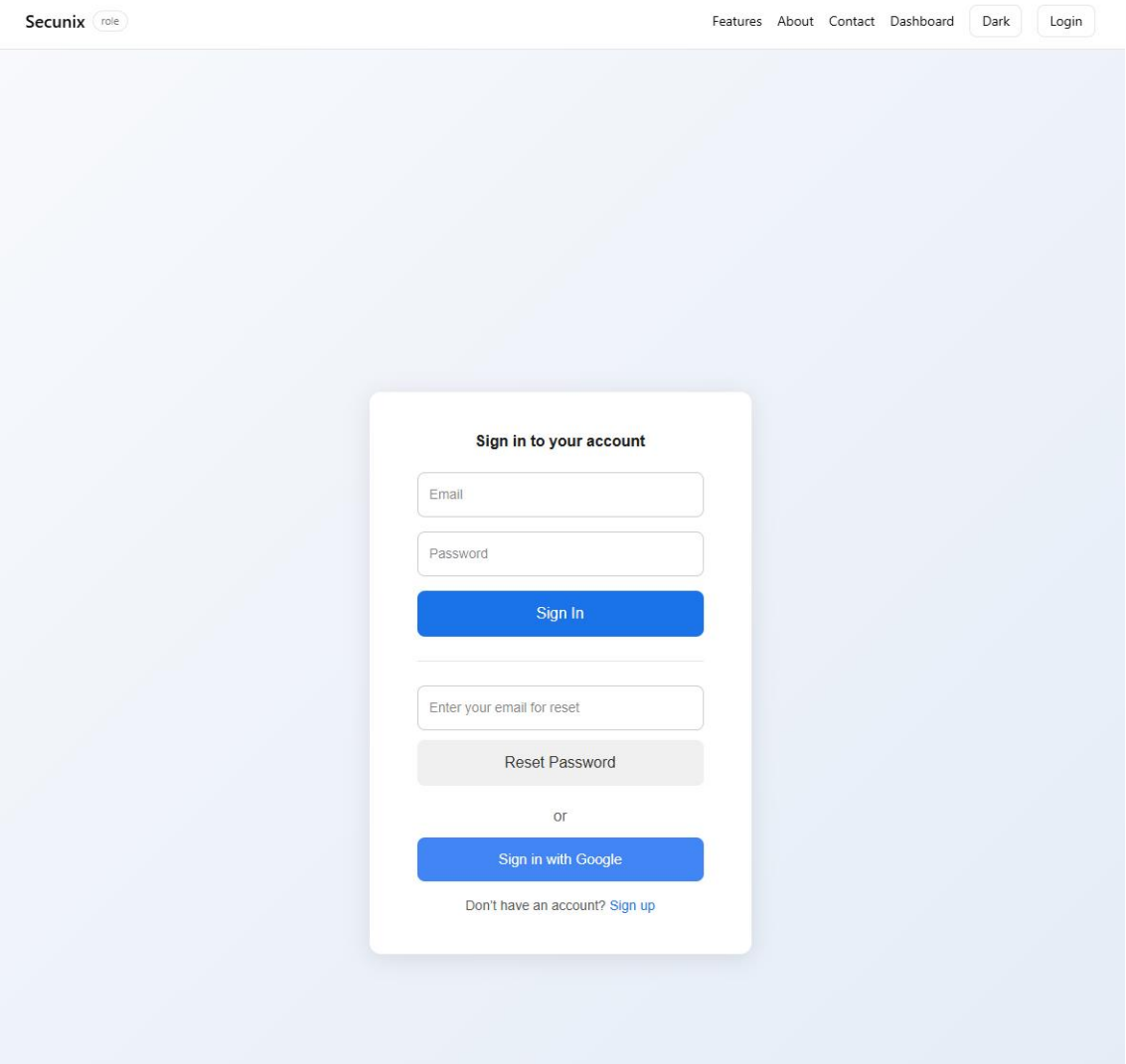

Figure 1- Login Page

3.4.2 Authentication

Authentication services will utilize Google Single Sign-On and verified email-based account creation. During initial registration, verification codes will be issued to confirm user identity before granting access. These verification codes must expire in five minutes to prevent unauthorized use and must be eligible for reissuance upon user request. After successful authentication, the backend will generate a signed JWT that is stored in an HTTP-Only cookie. This token will be added to all API requests going to the backend from the frontend and will be validated to enforce access restrictions. Google Single Sign-On and verified email-based account creation. During initial registration, verification codes will be issued to confirm user identity. These verification codes must expire after five minutes and be eligible for reissuance upon request. After authentication, a JWT is created and stored as an HTTP-Only cookie to authenticate the user during subsequent API requests to the backend.

3.4.3 Display Landing Page

The system will provide a public landing page that is accessible to unauthenticated users. This page will present an overview of the platform's purpose, key features, and system capabilities. The landing page will provide clear navigation options for both account creation and login, directing users to authentication workflows before enabling system access. The landing page should not reveal any sensitive system information and will only act as the entry point for all user workflows leading to the authenticated dashboard.

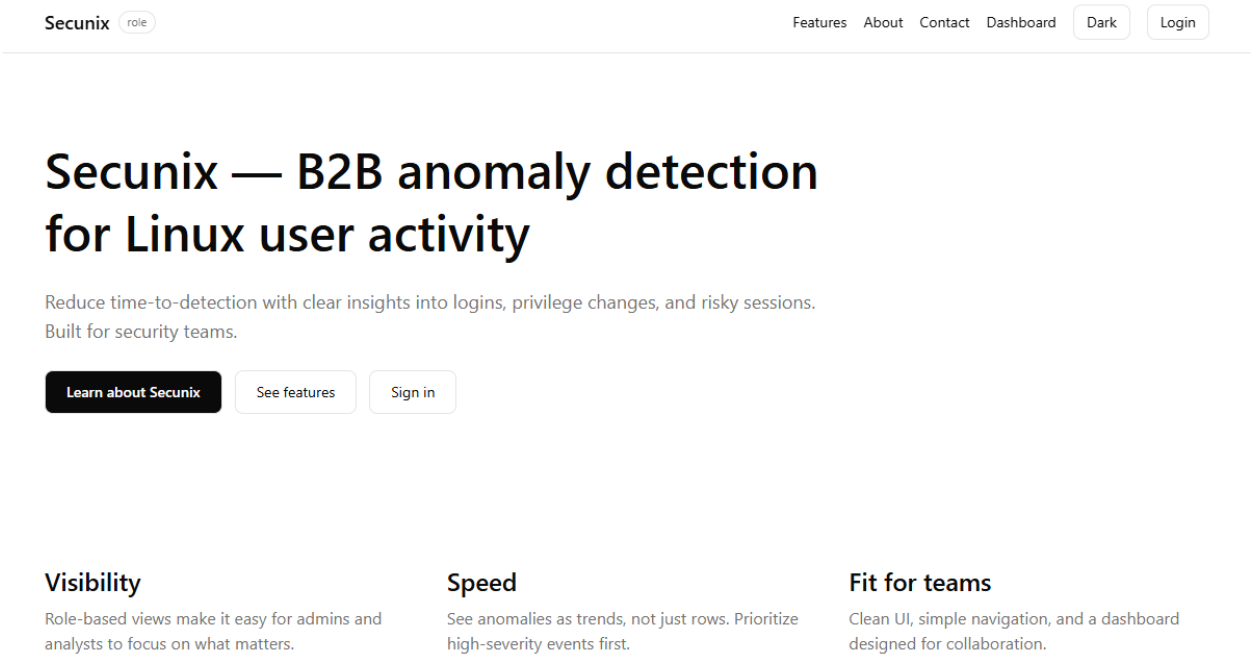

Figure 2 - Landing Page

3.4.4 Real-Time Detection Dashboard

Once authenticated, users will be directed to the detection dashboard, which serves as the primary user interface for monitoring anomalies. This dashboard must display continuously updating anomaly records produced by the machine learning model as they become available. Each reported detection will include metadata such as timestamp, severity score, associated virtual machine, user context, and a short description of the detected activity.a short description of the detected activity.

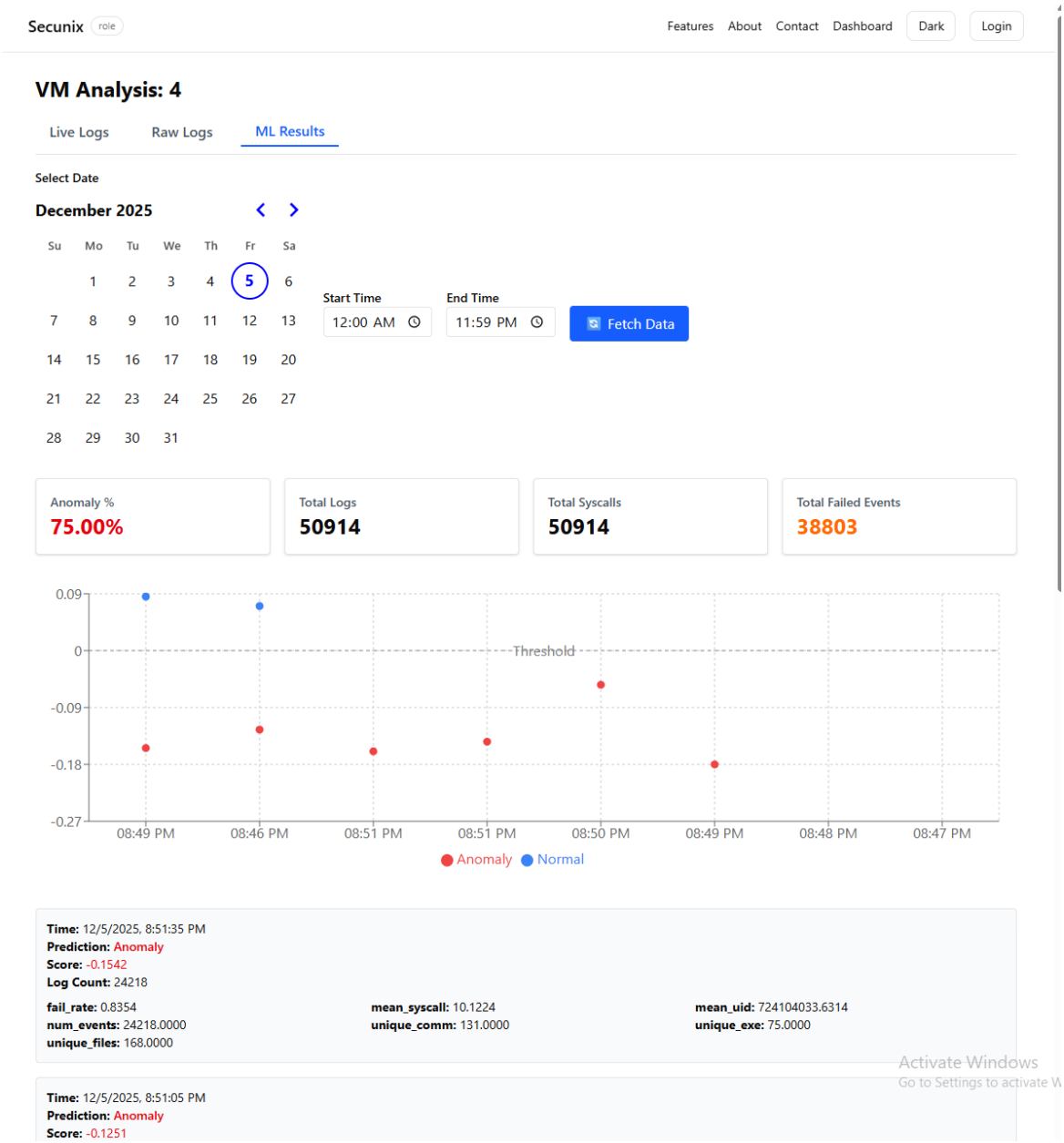

Figure 3 - User ML Results Page

3.4.5 History Page

The system will provide a history page for reviewing previously detected anomalies and logs collected. The history page will allow users to filter based on date and time. This will allow users to view logs within a given time frame of an anomaly to help them pinpoint the cause. Access to this page will require authenticated user status. time frame of an anomaly to help them pinpoint the cause. Access to this page will require authenticated user status.

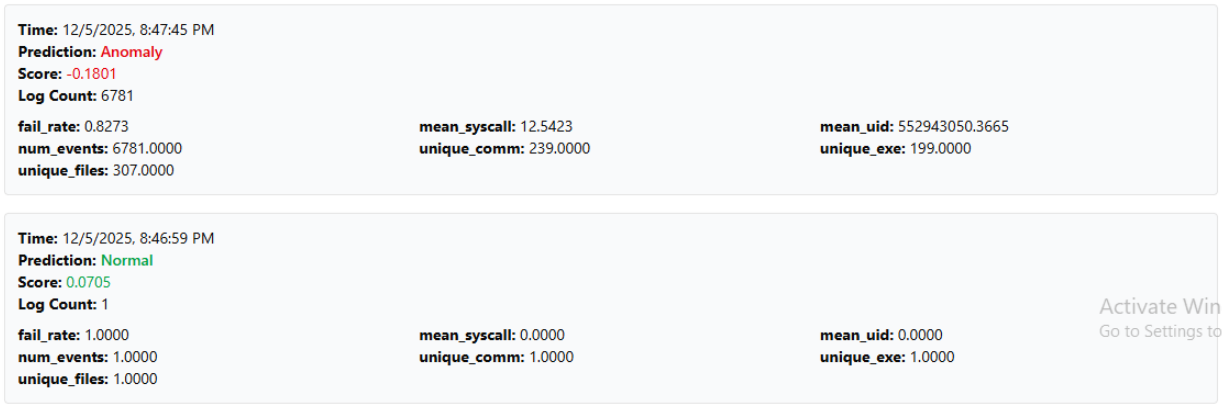

Figure 4 – Anomaly History Page

3.4.6 Data Visualization

The platform will provide built-in data visualization features that present anomaly patterns graphically over time. Visual outputs will include line charts showing anomaly frequency trends, distribution charts summarizing severity levels, and aggregated visual summaries by virtual machine or time range. will include line charts showing anomaly frequency trends, distribution charts summarizing severity levels, and aggregated visual summaries by virtual machine or time range.

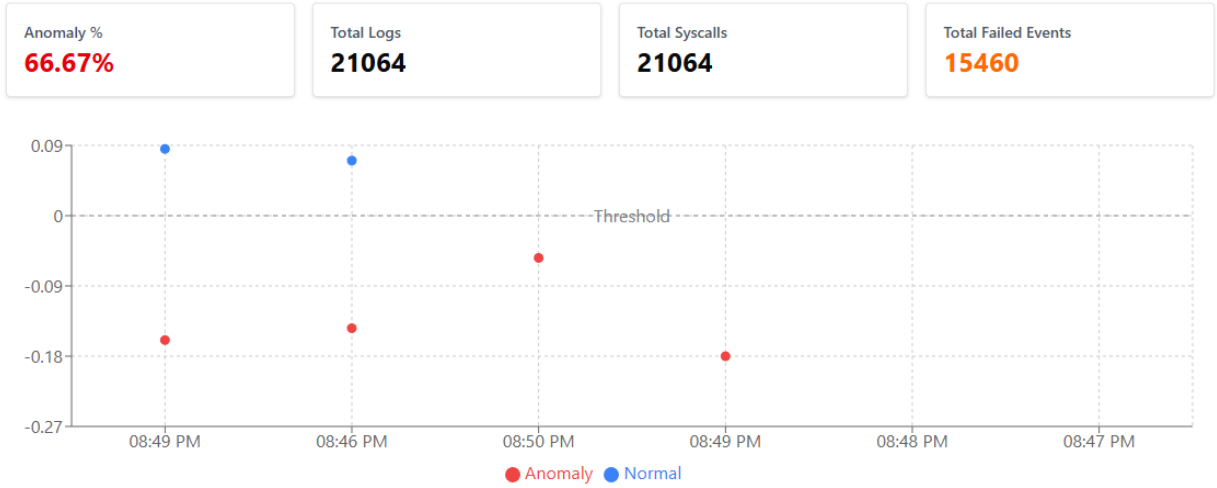

Figure 5 - Data Visualization Graph

3.4.7 Collection and Processing of Linux Audit Logs

The system will continuously collect raw security audit logs generated by the Linux auditd service. Filebeat will monitor the audit log file (/var/log/audit/audit.log) and forward all new records to the Kafka audit-logs topic for ingestion. A Flask Kafka consumer service will preprocess these records, transform them into feature representations suitable for machine learning analysis, and index all original raw log entries into the Elasticsearch raw_logs index for storage and retrospective analysis. Kafka consumer service will preprocess these records, transform them into feature representations suitable for machine learning analysis, and index all original raw log entries into the Elasticsearch raw_logs index for storage and retrospective analysis.

3.4.8 Anomaly Detection

The system will perform real-time anomaly detection using the Isolation Forest machine learning model on preprocessed log events. Each event will receive a binary classification indicating whether it is anomalous as well as a numeric anomaly confidence score. Model predictions will be published to the Kafka ml-results topic and subsequently indexed within the Elasticsearch ml_results database index. All detection results must be permanently viewable via the dashboard and historical analysis tools. indexed within the Elasticsearch ml_results database index. All detection results must be permanently viewable via the dashboard and historical analysis tools.

3.4.9 Alerts

Detected anomalies must immediately appear on the detection dashboard to notify analysts without additional delay. Each alert will present contextual metadata including affected system identifiers, timestamps, severity score, and summary descriptions to facilitate investigation. Dashboard reporting remains the primary alerting method within the system.additional delay. Each alert will present contextual metadata including affected system identifiers, timestamps, severity score, and summary descriptions to facilitate investigation. Dashboard reporting remains the primary alerting method within the system.

3.4.10 System Performance and Scalability

The system must support horizontal scalability to accommodate growth in the number of monitored Linux hosts and overall log volume. Kafka consumers, machine learning processors, Elasticsearch nodes, and backend API services must be deployable in parallel to increase throughput and maintain performance. The pipeline must continue to operate efficiently when ingesting high-volume log streams while maintaining low detection-to-visualization latency and fault-tolerant processing behavior.

3.5. Non-Functional Requirements

3.5.1 Performance

The platform must process log streams with minimal latency from ingestion to dashboard display. Kafka must maintain throughput supporting thousands of events per second while guaranteeing no message loss or duplication.

3.5.2 Security

Only authenticated users are permitted to make requests to the Flask backend. Auth.js, integrated with Next.js, is used to protect application routes by enforcing authentication through middleware and redirecting unauthenticated users. Upon successful login, a session token is stored in an HTTP-only cookie, which is included with subsequent API requests. This token can only be accessed and validated by the server, preventing client-side tampering. The Flask backend validates the session on each request and enforces authorization checks to ensure users can only access resources for which they are permitted.

3.5.3 Reliability and Availability

Kafka will persist data until consumers confirm successful processing to prevent loss during service interruptions. Elasticsearch replicas will ensure read access remains available even during partial node failures. All containerized services must be able to restart without loss of operational state or stored data.

3.5.4 Scalability

System capacity must scale linearly by adding Kafka brokers, ML workers, or Elasticsearch shards as log volume increases. No redesign should be necessary for supporting additional Linux hosts or increased ingestion rates.

3.5.5 Data and Storage

Kafka provides short-term buffering and delivery guarantees for streaming log data. Elasticsearch stores all original logs and ML results indefinitely to enable forensic analysis and historical research. Direct linkages between raw audit records and ML prediction entries must allow users to inspect source events underlying each detection.

3.5.6 Testing and Validation

Unit tests must validate log preprocessing, vectorization, and ML prediction accuracy. Integration tests must confirm Kafka routing, Elasticsearch indexing, API responses, and dashboard visualization accuracy. End-to-end testing must involve real Linux attack simulations such as repeated authentication failures and privilege escalation attempts to verify detection correctness.

3.6. External Interface Requirements

3.6.1 User Interface

The user interface will consist of a responsive Next.js dashboard featuring real-time anomaly listings, historical analysis charts, session management screens, and user profile pages. Navigation must operate across desktops and tablets while maintaining consistent display fidelity in both light and dark themes.

3.6.2 Hardware Interfaces

Each monitored system must operate on Linux and support execution of auditd and Filebeat agents. Cloud servers must run Docker services required to support Kafka, ML, API, and database operations. No specialized hardware is required for system operation; optional GPU usage may be employed only for model training acceleration.

3.6.3 Software Interfaces

The Flask backend exposes REST endpoints allowing retrieval of logs, anomalies, and VM assignments. Machine learning components communicate internally using Kafka topics rather than direct API calls to support pipeline decoupling. Elasticsearch accepts writes solely from backend consumers. Supabase authentication services interact with SQLAlchemy ORM schemas to regulate access permissions.

3.6.4 Communication Interfaces

The frontend communicates exclusively via HTTP REST endpoints with the Flask backend. Inter-service log transport flows from Filebeat to Kafka, consumer microservices to Kafka results topics, and ultimately to Elasticsearch. No WebSocket components are utilized within the system architecture. All communication channels employ encryption and adhere to security best practices.

4. Design

4.1. Design Introduction

This section describes the final system design for the cybersecurity platform, which collects Linux audit logs, performs anomaly detection using machine learning, stores both raw and processed data in Elasticsearch, and presents results through a secure web interface. The goal of the platform is to identify abnormal user or system activity across monitored virtual machines and provide security analysts with both real-time visibility and effective tools for historical investigation.

The system uses a modular, distributed architecture that separates data collection, streaming, processing, storage, and visualization into independent components. This approach simplifies development and maintenance while improving scalability and flexibility. Each subsystem can be deployed, maintained, or upgraded independently without impacting the overall platform, allowing the solution to adapt as system requirements grow or change.

4.2. System Overview

The platform consists of the following major components:

- **Log Generation**: The Linux auditd service runs on each monitored VM and records security-relevant system events such as logins, file access, and command execution.
- **Log Forwarding**: Filebeat monitors the audit log file and forwards newly generated entries directly into the Kafka streaming pipeline.
- **Streaming Layer**: Apache Kafka receives incoming log events, buffers them reliably, and distributes the data to subscribed consumers for processing and storage.
- **Processing Pipeline**: A Python-based consumer reads log messages from Kafka, extracts relevant fields, and converts each event into feature vectors suitable for machine learning analysis.
- **Machine Learning**: A pre-trained Isolation Forest model evaluates these features to determine whether observed behavior falls outside normal usage patterns and labels potential anomalies.
- **Result Streaming**: Machine learning predictions are published to a dedicated Kafka topic so that they can be consumed independently without affecting log ingestion.
- **Storage**: Elasticsearch stores both raw audit logs and machine learning results, enabling fast searching, filtering, and aggregation for dashboards and historical analysis.
- **Backend API**: A Flask-based backend service exposes secure REST endpoints that allow authorized users and frontend components to query logs, predictions, and system metadata.

- **Authentication and User Data**: Supabase, integrated through SQLAlchemy, manages user accounts, role assignments, and virtual machine ownership records.
- **Frontend Interface**: The Next.js dashboard consumes backend APIs to provide real-time monitoring views, historical charts, and interactive filtering tools for analysts.
- **Deployment**: All system services are containerized and coordinated using Docker Compose, with production deployment hosted on Digital Ocean nfrastructure.

4.3. System Architecture

The system operates as a sequential processing pipeline.

4.3.1 Log Collection

The Linux auditd service runs on each monitored VM. Custom rules define monitored activities such as command execution, file access, and login events. Logs are written to:
/var/log/audit/audit.log

4.3.2 Log Forwarding

Filebeat runs on the same VM and continuously monitors the audit log file. As new logs are written, Filebeat forwards each entry to Kafka using its Kafka output module.

Logs are sent to the following Kafka topic:
audit-logs

4.3.3 Kafka Streaming Layer

Kafka buffers and distributes incoming data. It decouples log producers from consumers and enables horizontal scaling.

Two Kafka topics are used:

- **audit-logs:** Receives raw audit events.
- **ml-results:** Receives anomaly prediction outputs from the ML service.

4.3.4 Kafka Consumers

Multiple consumers subscribe to these topics:

Consumers for audit-logs:

- **Machine Learning consumer** - builds feature vectors and performs prediction.
- **Elasticsearch consumer** - stores raw logs into Elasticsearch.
- **Server-Sent Events (SSE) stream** - provides live logs to the Next.js frontend for dashboard display.

Consumer for ml-results:

- **Elasticsearch consumer** - stores ML predictions into indexed storage.

4.3.5 Machine Learning Pipeline

The ML microservice:

1. Consumes log entries from audit-logs.
2. Vectorizes relevant features using scikit-learn tools.
3. Evaluates each record using a pretrained **Isolation Forest** model.
4. Produces anomaly predictions with associated confidence scores.
5. Publishes results to the ml-results topic.

4.3.6 Backend API

The Flask backend manages:

- User authentication sessions and access control
- Supabase database interactions via SQLAlchemy
- Elasticsearch queries for both logs and predictions
- API endpoints enabling time-based log searching and filtering

4.3.7 Storage Layer

Elasticsearch

Two primary indices are defined:

raw_logs

Stores:

- VM identifier
- Timestamp
- Event type
- Process and syscall metadata
- Full unmodified log entries

ml_results

Stores:

- VM identifier
- Feature vector reference
- ML cluster and anomaly score
- Binary anomaly classification
- Prediction timestamps

Elasticsearch enables fast searches, filtering, and dashboard aggregation.

Supabase Database

Supabase (PostgreSQL) stores user authentication and relationship data:

- Users
- Roles (Admin, Analyst, Viewer)
- Virtual machine ownership

SQLAlchemy provides ORM mapping from backend Python services to Supabase records.

4.3.8 Frontend Interface

The frontend is built using Next.js and communicates exclusively through REST API calls to Flask.

Core features include:

- Real-time dashboards
- Historical anomaly charts
- Log streaming tables
- VM filtering
- User profile management
- Organization-level access control
- Dark/Light Mode UI

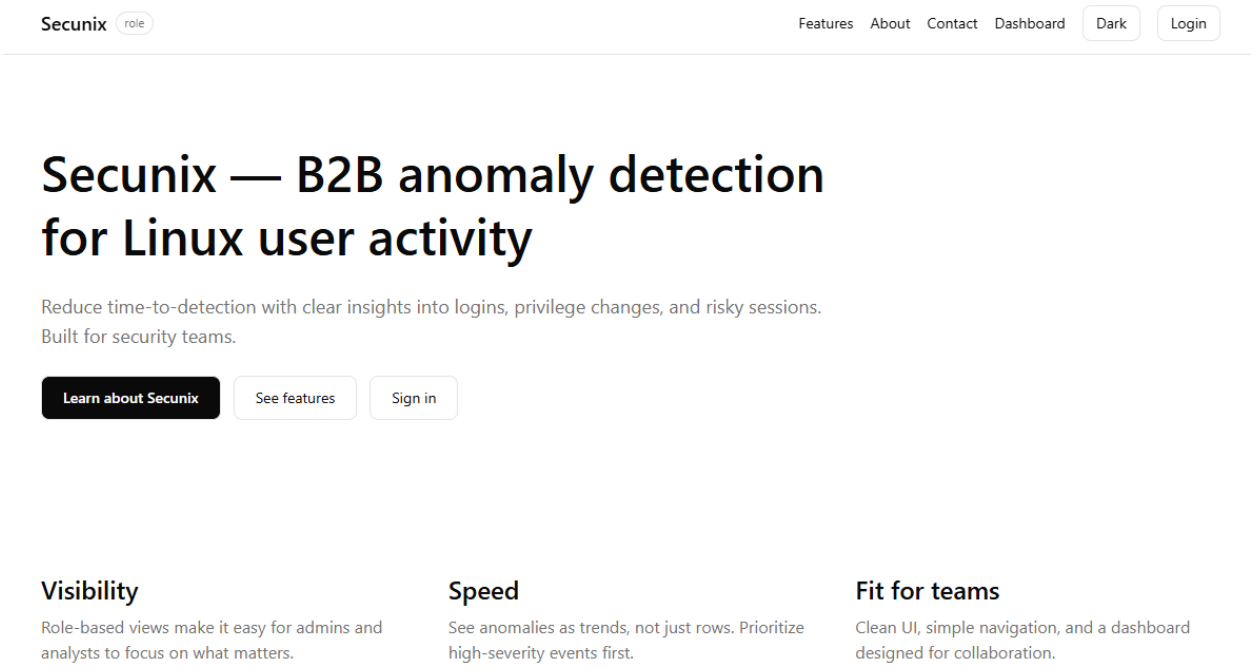

Figure 6 - Landing Page (Light Mode)

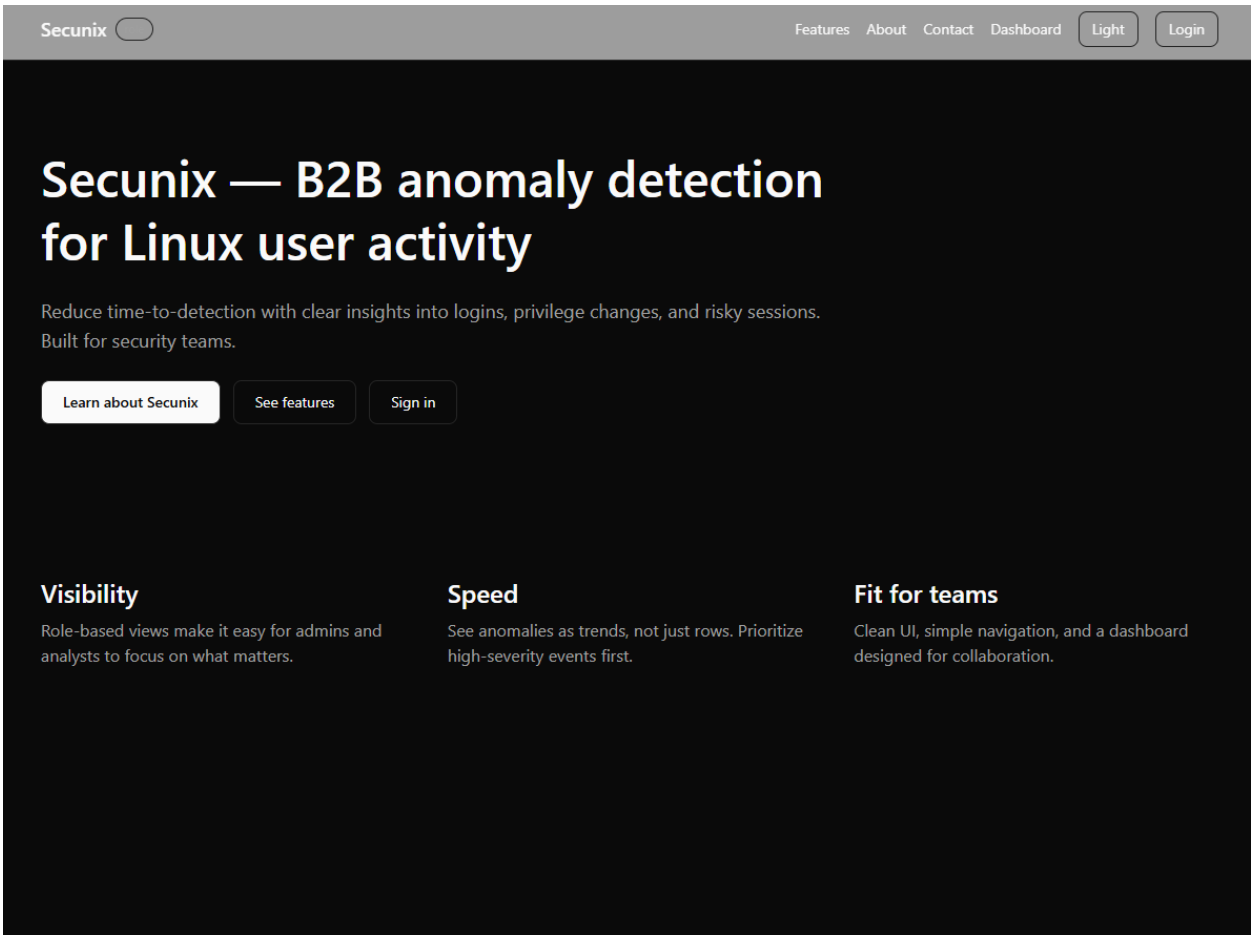

Figure 7 - Landing Page (Dark Mode)

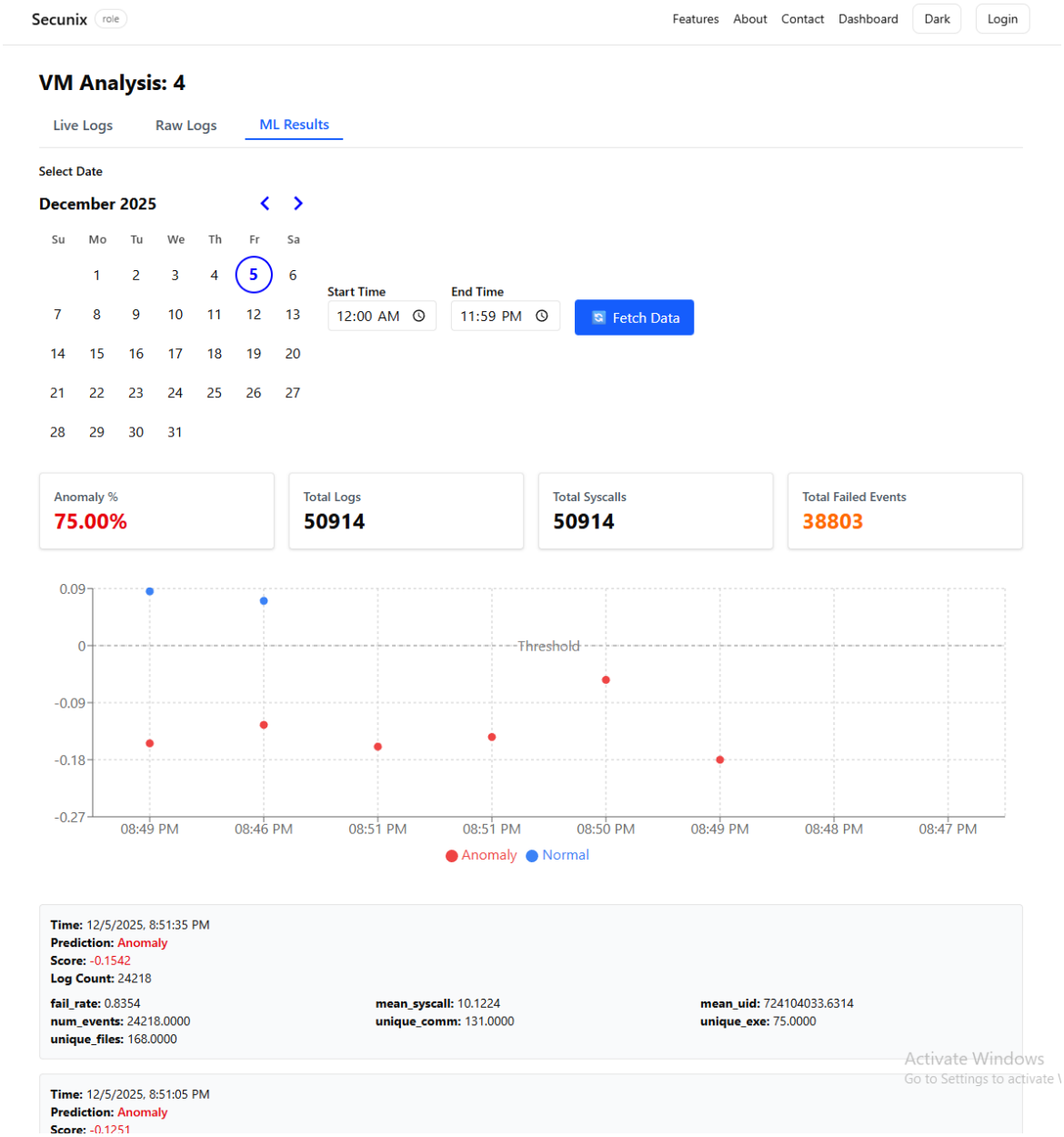

Figure 8 - Detection Dashboard Interface

4.4. System Constraints

4.4.1 Performance

- Kafka handles large-scale ingestion spikes.
- Isolation Forest with batch processing maintains real-time inference throughput.
- Elasticsearch enables fast analytics on large datasets.

4.4.2 Scalability

- Additional Kafka consumers and ML services can be deployed without redesign.
- Elasticsearch can scale via sharding and node replication.

4.4.3 Reliability

- Kafka ensures message durability.
- Independent service containers allow for safe restarts.
- Elasticsearch replication prevents data loss.

4.4.4 Security

- API access is authenticated via Auth.js with Google SSO.
- Supabase manages user data protection.
- Role-based controls limit system permissions.
- Elasticsearch write access is restricted to backend services only.

4.5. Data Flow Summary

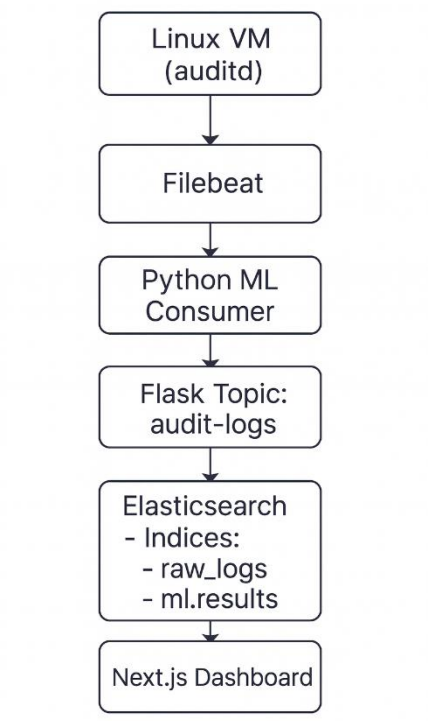

Figure 9 - System Architecture and Data Pipeline

5. Development

5.1. Technical Implementation Summary

System logs originate from Linux virtual machines and are forwarded via Filebeat to the Kafka **audit-logs** topic. A Python-based consumer service reads these log entries, transforms relevant fields into numerical feature vectors, and uses the Isolation Forest machine learning model to classify potential anomalies. Prediction results are then published to the **ml-results** Kafka topic for downstream processing and storage.1.1 Machine Learning Pipeline

System logs originate from Linux virtual machines and are forwarded via Filebeat to the Kafka **audit-logs** topic. A Python-based consumer service reads these log entries, transforms relevant fields into numerical feature vectors, and applies the Isolation Forest machine learning model to classify potential anomalies. Prediction results are then published to the **ml-results** Kafka topic for downstream processing and storage.

5.1.2 Kafka Consumers

Multiple Kafka consumers subscribe to the **audit-logs** topic:

- The machine learning service, which processes logs and performs anomaly detection
- An Elasticsearch consumer, which stores raw log data within the **raw_logs** index
- A Server-Sent Events stream, which provides live updates to the Flask backend for real-time dashboard display

A dedicated consumer subscribes to the **ml-results** topic and stores model predictions within a time-series Elasticsearch index for efficient querying and visualization.

5.1.3 Elasticsearch API Endpoints

The Flask backend exposes API endpoints that allow authenticated users to query both raw log data and machine learning detection results. Users can filter queries by time range to retrieve relevant records for investigation or reporting.

5.1.4 Frontend and Backend Architecture

The backend is implemented as a Flask application responsible for authentication handling, database interactions, and API requests. The frontend is built using Next.js and consumes these

APIs to render dashboards and data visualizations. Authentication is handled using Auth.js with Google Single Sign-On support.

5.1.5 User Authentication Features

Core authentication functionality includes routes for:

- User registration and login
- Email verification with expiring codes
- Password reset workflows
- Session validation and logout

Verification codes expire after five minutes and may be reissued if needed.

5.1.6 Frontend Features

The frontend provides an interactive user interface that includes real-time dashboards, visual charts, log-streaming tables, and filtering tools by user, virtual machine, or time range. Additional interface features include support for light and dark display modes, sidebar navigation, and profile management tools. Role-based access ensures users only see the views and functionality enabled for their assigned permissions.

5.2. Database Connection and Data Handling

5.2.1 Supabase Integration and ORM Schema

Supabase serves as the primary database for managing user accounts and virtual machine records. User entries store information such as user ID, username, email address, name fields, and verification codes used during account confirmation. Virtual machine records include fields such as VM identifiers, display names, and associated user ownership links.

SQLAlchemy is used as the object-relational mapping (ORM) tool within the Flask backend, allowing the application to interact with Supabase using structured Python models rather than raw SQL queries. This approach improves readability and maintainability while simplifying database operations. Secure credentials used to connect to Supabase are stored in backend configuration files to protect sensitive access tokens and prevent secrets from being committed to source code repositories. Together, Supabase and SQLAlchemy provide a secure

and scalable data layer that supports authentication workflows and VM management across the system.

5.2.2 Elasticsearch Configuration and Index Mapping

Elasticsearch is deployed as a containerized service using Docker. Configuration settings, including the selected image, container ports, and instance naming, are defined in the project's docker-compose.yml file. Once running, the Flask backend connects to Elasticsearch through the official Python client library, which provides programmatic access to indexing and query operations.

Index structures are defined within a dedicated setup script named elasticsearch_setup.py, where schemas (field mappings) are created for both the **raw_logs** index and the **ml_results** index. These mappings define each field's data type and behavior, similar to column definitions within a relational database. Establishing consistent index schemas ensures that incoming log records and machine learning results are indexed uniformly, allowing reliable filtering, aggregation, and dashboard visualization across the platform.

5.3. Project Setup Guide

This section outlines how to set up the full system from scratch, including the Linux environment, log forwarding, machine learning pipeline, backend and frontend services, and deployment.

1. Provision a VM running Linux Use a cloud provider (e.g., Digital Ocean) or local virtualization to create a Linux-based virtual machine.5.3.1 Prepare the Linux Virtual Machine
2. Provision a VM running Linux Use a cloud provider (e.g., AWS EC2) or local virtualization to create a Linux-based virtual machine.
3. Install and configure auditd
    - Install the audit daemon: sudo apt install auditd
    - Define custom audit rules in /etc/audit/rules.d/audit.rules to monitor system events like command execution, file access, and login activity.
    - Restart the service: sudo service auditd restart
    - Logs will be written to /var/log/audit/audit.log

5.3.2 Set Up Filebeat for Log Forwarding

1. Install Filebeat Download and install Filebeat from Elastic's official repository.
2. Configure Filebeat to forward logs to Kafka
    - Edit filebeat.yml to use the Kafka output module.
    - Specify:
        - Input path: /var/log/audit/audit.log
        - Kafka broker hostname and port
        - Kafka topic: audit-logs
    - Start Filebeat: sudo service filebeat start

5.3.3 Configure Kafka and Zookeeper

1. Include Kafka and Zookeeper in docker-compose.yml Define services for both Kafka and Zookeeper with appropriate ports and environment variables.
2. Create Kafka topics
    - audit-logs: for raw logs from Filebeat
    - ml-results: for processed ML predictions

5.3.4 Deploy the Machine Learning Microservice

1. Implement the ML consumer
    - Use kafka-python to consume messages from audit-logs
    - Apply an Isolation Forest model (via sklearn) to detect anomalies
    - Publish results to ml-results topic
2. Containerize the ML service
    - Create a Dockerfile with python:3.10-slim
    - Install dependencies: joblib, sklearn, kafka-python
    - Run the consumer script on container start

5.3.5 Set Up Elasticsearch

1. Add Elasticsearch to docker-compose.yml
    - Specify image, container name, and exposed ports
2. Define index mappings
    - Create elasticsearch_setup.py to define schemas for:
        - raw_logs: stores unprocessed audit logs

        - ml_results: stores ML predictions
3. Connect Flask to Elasticsearch
    - Use the Elasticsearch Python client
    - Configure host and port in Flask's config file

5.3.6 Build and Configure the Flask Backend

1. Implement core routes
    - Sign-up, login, email verification, password reset, session validation, logout
    - API endpoints to query raw_logs and ml_results from Elasticsearch
2. Integrate Supabase with SQLAlchemy
    - Define user and VM schemas
    - Store Supabase credentials in .env and load with python-dotenv
3. Install dependencies
    - List all required packages in requirements.txt:
        - Flask, Flask-CORS, SQLAlchemy, supabase, elasticsearch, kafka-python, sklearn, etc.
4. Containerize the backend
    - Dockerfile:
        - Base image: python:3.10-slim
        - Copy code, install dependencies, expose port 5000
        - Run Flask app

5.3.7 Build and Containerize the Next.js Frontend

1. Implement frontend pages
    - Sign-up and login pages
    - Dashboard to display logs and ML results
2. Configure Auth.js for Google SSO
    - Enable account creation and login via Google
3. Containerize the frontend
    - Dockerfile:
        - Install dependencies with npm install
        - Build with npm run build

- Serve with npm start
- Expose port 3000

5.3.8 Orchestrate All Services with Docker Compose

1. Define all services in docker-compose.yml
    - Kafka, Zookeeper, Elasticsearch, Flask backend, Next.js frontend, ML consumer
2. Launch the system
    - Run: docker-compose up

5.3.9 Deploy to Digital Ocean

1. Install Docker and Docker Compose on Digital Ocean5.3.9 Deploy to AWS EC2
2. Install Docker and Docker Compose on EC2
    - SSH into your instance and install both tools
3. Clone the project repository
    - Include all source code, Dockerfiles, and docker-compose.yml
4. Run the full stack
    - Execute docker-compose up to deploy the system
    - Supabase remains cloud-hosted and connects via API

6. Test Plan

6.1. Overview

6.1.1. Purpose

The purpose of this document is to define the complete testing strategy for the system that collects Linux audit logs, processes them through a machine learning pipeline, stores both raw and processed data in Elasticsearch, and exposes the results through a secure web platform. This document provides:

- A description of the end-to-end data flow from Filebeat → Kafka → ML service → Elasticsearch
- The responsibilities of each major component, including Kafka consumers, backend services, frontend UI, and authentication workflow
- How system technologies (Supabase, SQLAlchemy, Elasticsearch, Kafka, Filebeat, Docker) integrate
- Deployment considerations and required configuration across all services
- The test scope, test cases, procedures, environments, and expected deliverables

The overall goal is to ensure the system can be validated, deployed, and maintained reliably across different environments.

6.1.2. Scope

This test plan covers technical testing of:

- The machine learning pipeline (log ingestion, feature extraction, anomaly detection, result publishing)
- Kafka topics, consumers, and message flow
- Elasticsearch index mappings, schemas, and query behavior
- Backend functionality including authentication, role-based access control, and API endpoints
- Frontend components such as dashboards, log viewers, filters, and Google SSO
- Supabase + SQLAlchemy data storage and ORM integration
- Full deployment flow across Linux, Docker Compose, Filebeat, and Digital Ocean
- Full deployment flow across Linux, Docker Compose, Filebeat, and AWS EC2

This document defines the components under test, how they interact, and what constitutes successful system behavior.

6.2. Scope of Testing

6.2.1. In Scope

The following areas are included to verify both component-level correctness and full end-to-end operation.

Backend & Data Pipeline Testing

- Validate Filebeat → Kafka ingestion of audit logs
- Verify Kafka topic creation, message routing, and consumer functionality
- Test the ML microservice for:
- Log consumption
- Feature vectorization
- Anomaly prediction
- Publishing results to the ML results topic
- Verify Elasticsearch ingestion into raw_logs and ml_results
- Confirm backend API endpoints return accurate, filtered data

Authentication & User Management

- Test Supabase integration and SQLAlchemy ORM behavior
- Validate full authentication flow:
    - Sign-up, login, logout
    - Email verification and expiration timing
    - Password reset
    - Session validation

- Verify role-based access control for Admin, Analyst, and Viewer roles

System Integration & Deployment Testing

- Verify Docker Compose orchestration across all services
- Confirm networking, startup order, and environment configuration
- Validate deployment functionality on Digital Ocean
- Validate deployment functionality on AWS EC2

Frontend Application

Dashboard & UI Components

- Validate correct rendering of dashboard components (charts, log tables, anomaly counts)
- Ensure anomaly graph updates when new ML results arrive
- Verify real-time log streaming table correctly paginates and auto refreshes
- Confirm filtering tools (timestamp filters, VM filters, severity filters) work correctly
- Validate sorting of tables by timestamp, severity, user, and process

User Experience & Visual Consistency

- Test dark mode / light mode toggle behavior
- Confirm theme persists across:
  - Page reload
  - Logout/login
  - Navigation between Dashboard, Logs, Settings
- Validate responsive layout on desktop / tablet screens

Navigation & Routing

- Test sidebar navigation between all screens:
  - Dashboard
  - Logs
  - VM Management
  - User Profile
  - Admin Panel (Admins only)
- Confirm unauthorized users cannot access admin routes
- Test automatic redirect when unauthenticated (→ Login page)

Frontend ↔ Backend Integration

- Validate API data appears correctly on:
  - VM list page
  - Log viewer

    - Anomaly dashboard
- Confirm “No Data” states render properly

5.2.2. Out of Scope

The following items are *not* part of testing:

Machine Learning Model Training & Tuning

- Training, retraining, or tuning the ML model
- Model evaluation (accuracy, precision/recall)
- Dataset quality analysis
    - Only inference behavior will be tested

Usability Testing

- Formal UX studies
- Accessibility compliance
- A/B testing

Third-Party Platform Testing

- Testing internal functionality of Supabase, Kafka, Elasticsearch, Google SSO, or Docker

Only integration points are tested

Linux OS-Level Validation

- OS stability, kernel behavior, or long-term auditd performance testing

6.3. Test Cases

Test Case ID	Description	Expected Result
AUTH-01	Create account	Account created, verification email sent
AUTH-02	Request reset password	Reset link emailed
AUTH-03	Login with correct credentials	User redirected to dashboard
AUTH-04	Resend verification code	New code sent to email and accepted
AUTH-05	Login with incorrect credentials	Login denied
AUTH-06	Email verification code expires after 5 minutes	Expired error displayed. User is not verified and cannot access the system.
AUTH-07	Google SSO Login	User signs in with Google account and session is created

ML-01	Test ml model accuracy on normal user activity	No anomalies recorded
ML-02	Test ml model accuracy on abnormal user activity	Anomalies recorded
PIPELINE-01	Test ability to have multiple VM's per user	Logs and anomalies recorded for each VM
FE-01	Light/Dark mode toggle	UI switches instantly; selection persists on reload
FE-02	Dashboard loads with charts and summaries	Charts render with correct data; no blank or broken components
FE-03	Sidebar navigation	All pages load, correct section highlighted
FE-04	Real-time log updates	New logs stream into table without page reload
FE-05	User profile + settings page	Settings load and update confirmation shown

6.4. Test Procedures

AUTH-01

1. Open website
2. Select Sign up
3. Enter an email and password
4. Enter the verification code sent to that email
5. Account created
6. Enter credentials when redirected to Login Page
7. Expected: Login successful

AUTH-02

1. Open website
2. Select Login
3. Enter email and select Reset Password
4. Expected: Email received with link to reset password

AUTH-03

1. Open website
2. Select Login
3. Enter valid email and password
4. Expected: Successful login and directed to the VM page

AUTH-04

1. Open app
2. Select Sign up
3. Enter an email and password
4. Click the send a new verification code
5. Expected: New verification code sent through email

AUTH-05

1. Open website
2. Select Login
3. Enter invalid login credentials
4. Expected: Invalid credentials error message

AUTH-06

1. Open website
2. Select Login then Sign up
3. Fill out details including username, first and last name, email, and password
4. Select sign up
5. Wait for 5 minutes to pass and enter the credentials
6. Expected: Code expired error message. User is not verified and cannot access the system.

AUTH-07

1. Open website
2. Select Login
3. Select Sign in with Google
4. Enter your google account details
5. Expected: If verified, user is logged in and session is created

ML-01

1. Start VM
2. Open website
3. Start normal internet browsing on VM
4. Expected: No anomalies recorded on dashboard

ML-02

1. Start VM
2. Open website
3. Start malicious script on VM
4. Expected: Anomalies recorded on dashboard

PIPELINE-01

1. Start two VM's registered to one user
2. Open website
3. Start system monitor on each VM to produce logs
4. Expected: Logs and ML model predictions to be displayed for each VM

FE-01

1. Open frontend application
2. Click theme toggle icon
3. Observe UI changes
4. Refresh page
5. Expected: Selected theme persists and reloads correctly

FE-02

1. Login and navigate to Dashboard
2. Check anomaly count card
3. Check Line Chart and Bar Chart (log count over time)
4. Expected: All charts render with proper scale and labels

FE-03

1. Use sidebar to switch between Dashboard → Logs → VMs → Profile
2. Observe the active highlight
3. Expected: Correct page loads; active tab highlighted

FE-04

1. Generate logs on VM (using system monitor from previous tests)
2. Stay on Log Viewer page
3. Expected**:** New logs appear automatically without reload

FE-05

1. Navigate to Profile
2. Update display name or email preferences
3. Click Save
4. Expected: "Settings saved" message appears

6.5. Test Environment

Hardware and VM Environment

- Each VM: 4GB RAM, 50GB disk space, and 4 CPU cores

Software and Services

- Kafka and Zookeeper (Docker)
- Elasticsearch (Docker)
- Filebeat (Linux)
- Flask Backend
- Next.js frontend
- Supabase cloud database
- Docker compose for orchestration and containerization
- Python 3.20, Node.js 18
- Virtualbox

Network

- Internal Docker network for service communication
- Filebeat data sent to Kafka via configured host
- Backend connected to Superbase via secure API connection

6.6. Test Data

Authentication Test Data

- Valid and invalid emails
- Invalid passwords
- Expired verification codes

ML Test Data

- Regular user activity (web browsing)
- Script that guesses root password

Pipeline Test Data

- Running system monitor application to produce logs

7. Test Report

Test Case ID	Description	Test Result	Severity	Tester	Comment
AUTH-01	Create account	Pass	None	Noor Aissat	Account created successfully, verification code emailed, login after verification works

AUTH-02	Request reset password	Pass	None	Noor Aissat	Reset link delivered to email instantly and allowed password update
AUTH-03	Login with valid credentials	Pass	None	Noor Aissat	Logged in successfully and redirected to VM page
AUTH-04	Resend verification code	Pass	None	Noor Aissat	New code sent and used successfully to verify account
AUTH-05	Login with incorrect credentials	Pass	None	Noor Aissat	Proper error message displayed, login prevented
AUTH-06	Verification code expires after 5 minutes	Pass	None	Noor Aissat	Code expired, user had to generate a new verification code to continue with account creation
AUTH-07	Google SSO login	Pass	None	Noor Aissat	Google redirects and authentication completed, session created successful
ML-01	ML model test with regular activity	Fail	Moderate	John Fuentes	Opening the browser initially resulted in an anomaly, most likely due to large volume of system logs
ML-02	ML model test with irregular activity	Pass	None	John Fuentes	While the script was running anomalies were being reported
PIPELINE-01	Test applications ability to handle multiple VM's	Pass	None	John Fuentes	Logs and anomalies were being processed successfully for both VM's and displayed on the dashboard
FE-01	Light/Dark mode toggle	Pass	None	Geovanni Cuevas	Theme persisted after reload
FE-02	Dashboard charts render	Pass	None	Geovanni Cuevas	Charts load with correct data

FE-03	Sidebar navigation	Pass	None	Geovanni Cuevas	Navigation smooth and correct
FE-04	Real-time logs	Pass	None	Geovanni Cuevas	Logs update every ~2s as expected
FE-05	Profile settings	Pass	None	Geovanni Cuevas	Settings save correctly

8. Version Control

Version control for this project was managed using GitHub, which provided a structured and dependable workflow for coordinating development across multiple team members. To maintain clear organization and prevent conflicts between in-progress features, the team adopted a branching strategy centered around isolated feature branches.

At the start of development, each team member created their own personal branch from the main branch. This separation ensured that experimental work, partial implementations, and debugging tasks could be completed without introducing instability into the production-ready codebase. For every major component, such as the Flask API, machine learning pipeline, Next.js dashboard, and Kafka integration, dedicated feature branches were created. This allowed work to progress in parallel and enabled the team to focus on specific tasks while keeping the overall system cohesive.

Each feature branch underwent regular commits documenting incremental progress, configuration changes, data model adjustments, and bug fixes. When a feature reached a stable and testable state, the developer submitted a pull request to merge their branch into the main or a shared development branch. Pull requests served as checkpoints for code review, where teammates could provide feedback, identify potential issues, and verify compatibility with other components of the system. This process helped maintain code quality and reduced the likelihood of integration problems later in the pipeline.

GitHub Issues and project boards were also used to track tasks, assign responsibilities, and monitor completion status throughout development. This structure supported transparency and ensured that all system requirements were implemented in an organized sequence.

By following a disciplined branching model and using GitHub's collaboration tools, the team maintained a clean, traceable, and well-structured codebase. This approach not only improved development efficiency but also created a reliable foundation for debugging, testing, and scaling the final project

9. Conclusion/Summary

This project successfully developed a full-stack cybersecurity monitoring platform capable of detecting unusual activity across Linux virtual machines using machine learning. The system combines real-time log collection, automated analysis, secure data storage, and interactive visualization into a single, scalable pipeline. By integrating auditd, Filebeat, Kafka, a Python-based Isolation Forest model, Elasticsearch, Flask, Next.js, and Supabase authentication, the platform demonstrates how modern streaming architecture and machine learning techniques can be applied to practical security monitoring problems.

Throughout development, several challenges were addressed, particularly the difficulty of obtaining realistic normal user activity data for training the anomaly detection model. Simulating real-world behavior and collecting extended baseline data allowed the team to overcome this limitation and build a functioning detection pipeline. Performance concerns related to high log volumes were successfully managed using Kafka's message buffering and the ability to scale consumers horizontally. Security and reliability goals were achieved through role-based authentication, encrypted communication, containerized service isolation, and persistent data storage strategies that protect system integrity even during service restarts.

System testing verified the correct operation of the data pipeline, authentication workflows, dashboard features, and machine learning detection process. While most test cases passed as expected, machine learning sensitivity to initial bursts of normal activity highlighted the ongoing need for careful model tuning and baseline refinement. This result reflects real-world challenges commonly encountered when deploying anomaly detection systems and provides valuable insight for future improvement.

Overall, this project delivers a functional and extensible security monitoring solution that meets its primary objectives of real-time detection, visualization, and secure system management. The modular design allows for future enhancements such as improved modeling techniques, automated alert escalation, reduction of false positives, and expansion to larger host environments. The completed system demonstrates both the technical feasibility and practical value of integrating distributed streaming pipelines with applied machine learning for cybersecurity monitoring.

10. Appendices

10.1. Glossary

auditd — The Linux auditing framework that records kernel-level security events.

Filebeat — A lightweight log shipping agent responsible for forwarding audit logs to Kafka.

Kafka — A distributed streaming platform used for buffering and routing logs and machine learning results.

Isolation Forest — A machine learning algorithm used to detect anomaly patterns in event data.

Elasticsearch — Search and analytics database storing both raw audit logs and ML prediction results.

Supabase — Managed PostgreSQL database service used for user accounts and VM metadata.

SSO — Single Sign-On via Google authentication.

False Positive — Normal behavior incorrectly flagged as anomalous.

False Negative — Malicious behavior not detected by the system.

10.2. References

Elastic. Filebeat. Retrieved from https://www.elastic.co/docs/reference/beats/filebeat/

Data Overload. Sliding Window Technique — reduce the complexity of your algorithm. Medium. Dec. 21, 2022. Retrieved from https://medium.com/@data-overload/sliding-window-technique-reduce-the-complexity-of-your-algorithm-5badb2cf432f

PythonAfroz. Anomaly detection using Isolation Forest (Kaggle notebook). Retrieved from https://www.kaggle.com/code/pythonafroz/anomaly-detection-using-isolation-forest

AuthJS. Getting Started — Google provider. Retrieved from https://authjs.dev/getting-started/providers/google

Elastic. Create index (full-text search tutorial). Retrieved from https://www.elastic.co/search-labs/tutorials/search-tutorial/full-text-search/create-index

Elastic. Linux detection engineering with auditd (Security Labs). Retrieved from https://www.elastic.co/security-labs/linux-detection-engineering-with-auditd

Supabase. Python Client Reference – Introduction. Retrieved from https://supabase.com/docs/reference/python/introduction

Peyrone. Establishing a connection to Apache Kafka using Python Flask. Medium. Retrieved from https://peyrone.medium.com/establishing-a-connection-to-apache-kafka-using-python-flask-9c56dcaeb0a0

Docker. Compose. Retrieved from https://docs.docker.com/compose