Automated Log Analyzer

REVA B. BONDE

ML AUTOMATION

Automated CyberSecurity Log Analyzer Agent

This is a ML based CyberSecurity Agent that automates tasks of a Log Analyzer. This is built in COLAB and coded entirely in Python. This agent automates the process of:

1. Collecting logs: from APIs, files, servers or monitoring tools.
2. Cleaning and parsing them: convert messy raw logs into structured form.
3. Detecting anomalies: using rules and machine learning.
4. Creating alerts & HTML reports: no need for manual analyzing.

Importance of this project:

- **Automated Log analysis is a core SOC(Security Operations Center) Task:** Companies generate GBs or TBs of logs everyday due to security breaches behind login attempts, system events, network requests, errors, etc. Manually reading them is impossible.
- **SOC analysts rely on automated agents:** Tools like Splunk, Wazuh, ELK Stack, SIEM Solutions, Crowd Strike, MS Sentinel (tools used for threat detection and log analysis) follow the pipeline:

Collect ⟶ **Parse** ⟶ **Analyze** ⟶ **Detect** ⟶ **Alert**

This pipeline is also used in the project.

Technologies used:

✔ API integration
✔ Data cleaning
✔ Cybersecurity event understanding
✔ Machine learning for anomaly detection
✔ Automation (agent behavior)
✔ HTML reporting (Jinja2 templates)
✔ Practical SOC workflow

Architecture:

01. Define Metadata:

A dictionary with details used to understand the project.

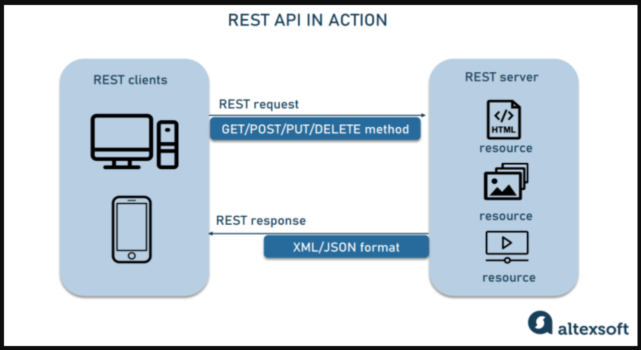
02. Import required libraries:

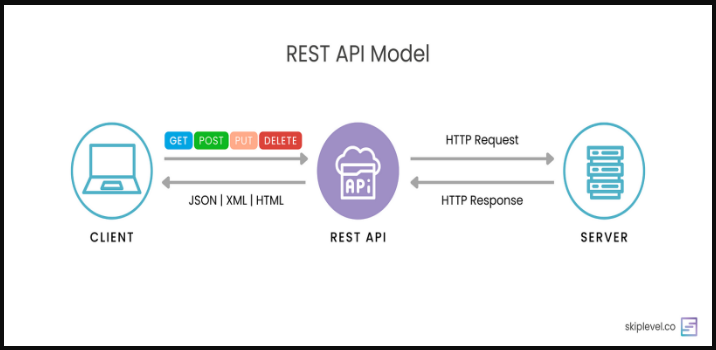
Standard libraries – os, json, tempfile, datetime and csv.

Third –party libraries – pandas, scikit-learn, requests, matplotlib, and jinja2.

03. Fetch logs from API:

requests.get() fetches logs from external API endpoint. (API Endpoint is the URL where a client(browser, app, script) sends requests to access data or service from a server). Companies often send requests to SIEM Endpoints, REST APIs , etc.





04. Clean and preprocess logs:

Using pandas remove missing values, convert timestamps, filter noise and extract fields.

05. Detect anomalies:

Algorithms such as Isolation Forest(works by isolating anomalies instead of profiling normal data) detect unusual activities in logs.

06. Create HTML reports and send alerts:

Jinja2 summarizes findings and display charts mimicking SOC Dashboards. Slack API sends alerts to the admin.

CODE

01. The project begins with **metadata** used for version tracking, consistent documentation, and updates and for the report to use the project name. Industrial tools have metadata such as SIEM tools, open-source libraries, docker containers and APIs. Metadata helps reporting , integration and CI/CD Development.
02. **Installing Dependencies**:

```
!pip install -q scikit-learn pandas matplotlib jinja2 requests
```

! : Is used to run a shell command.
-q flag : Quiet mode. Hides long pip installation logs
Libraries installed: import

Sr.	Libraries installed	Purpose
1.	Pandas	Dataframe, csv reading, log preprocessing(read, clean, filter)
2.	Scikit-learn	ML Models, anomaly detection
3.	Matplotlib (.pyplot)	Charts, embed into HTML reports, plot anomaly scores
4.	Jinja2	Templating engine, generates HTML reports
5.	requests	Send http requests, fetch logs via API(GET,POST) , send authentication logs, download json logs
6.	os	Handle file path, env variables, read/write log files, check existence of folders
7.	csv	Read raw comma separated values, write processed files
8.	tempfile	Temporary logs storage , temporary reports, temporary testing
9.	datetime	Convert strings to datetime, sort chronologically
10.	json	Read logs from API, convert Python dict to json, export logs
11.	Isolation Forest (from ensemble)	Unsupervised algorithm for anomaly detection(isolates rare/unusual points), fast, handles high dimensional data
12.	Standard Scaler (from preprocessing)	Normalize features(convert all values on a similar scale so that model treats each feature fairly)
13.	Template (from jinja2)	Dynamic HTML reports, mimics professional SIEM that generates pdfs/reports

03. Fetching logs + loading data + Preprocessing :

```
def fetch_logs(api_url):
  response = requests.get(api_url)
  if response.status_code == 200:
    return response.json()
  else:
    return []
```

-GET request to logging API
-checks status of request
-converts json to dict/list

Why APIs for logs?
To simulate real SOC scenario. Modern systems don't log onto local disks anymore. Logs are received from Firewalls, Cloud servers(AWS,AZURE), Authentication systems(OAuth,IAM), SIEM tools(Splunk , Sentinel), Application monitoring devices.

API response is turned to pd Dataframe:

```
[//example api response {
  "timestamp": "2025-11-26T10:15:23",
  "ip": "192.168.1.45",
  "event": "LOGIN_FAILED",
  "attempts": 3
 }]
```

```
df = pd.read_csv("synthetic_logs.csv")
```

Companies export logs in csv format

Preprocessing:
Convert timestamp strings to datetime objects(for ML's understanding):

```
pd.to_datetime()
```

missing values can create crashes during execution, NaN poisoning scaler, false anomalies

```
df.dropna()
```

Feature engineering:

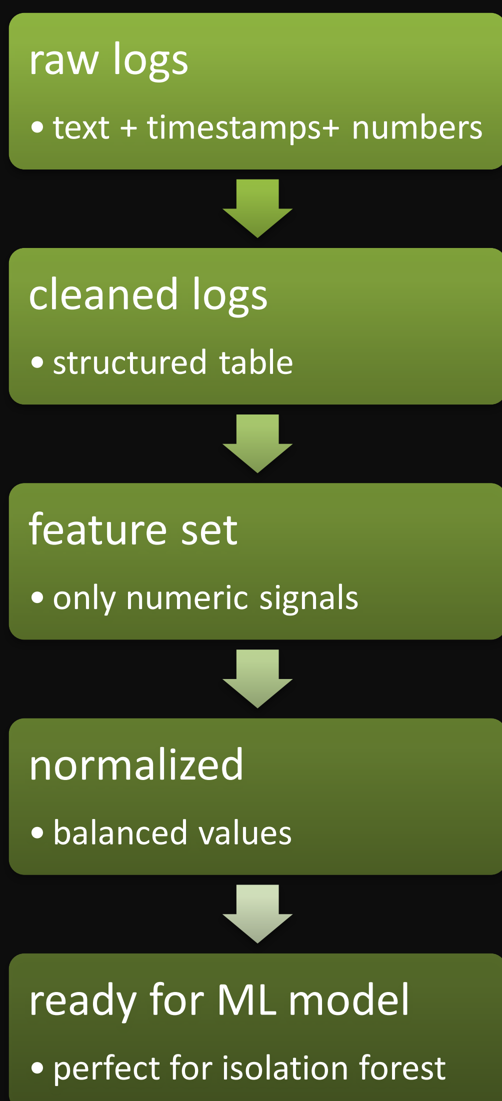
Model uses signal patterns instead of raw logs.

Features ⟶ normalize ⟶ feed to ML

Derived features:

Feature	Meaning	Importance	Threats
failed_logins	No. of failed attempts	Detect brute force	Brute force
requests_per_min	Request spikes	Detect DDoS	DDoS
unique_ips	Count of ip addresses	Detect lateral movement	Intrusion
response_time	Latency	Detect backend overload	Exploitation attempt

Stages of data processing:



04. Using ML model:

Isolation Forest algorithm:

Unsupervised(no labels required) anomaly detection (isolates anomalies rather than normal points) tree based(builds random trees) algorithm . This engine detects threats in logs.

Used in Cybersecurity intrusion detection, fraud detection, system monitoring, API usage anomaly detection.

Why we are using Isolation Forest?

This algorithm:

✔learns normal patterns on its own

✔flags deviations as anomalies

✔handles numerical logs very well

Isolation Forest:

1. randomly splits feature space
2. isolates points using cuts
3. anomalies get isolated **quickly**
   (they are few and different)
4. normal points require more splits

```
model = IsolationForest(contamination=0.05, random_state=42)

model.fit(scaled_features)

df['anomaly'] = model.predict(scaled_features)
```

contamination : tells what % of logs are anomalies

lower value = stricter model

higher value = more anomalies

.fit() : trains the model

.predict(): returns output -1(normal) and 1(anomaly)

Anomaly score:

```
df['score'] = model.decision_function(scaled_features)
```

higher score = normal

lower score = suspicious

05. Report generation , visualization and alerts:

```
from jinja2 import Template
import matplotlib.pyplot as plt
import tempfile
```

A clean report allows faster incident response

Jinja2 is a templating engine that allows dynamic HTML reports used by Flask, Ansible and Django like renderers

`{{ variable }}` → placeholder that gets replaced by actual values.

Why charts?

Humans detect patterns visually faster than numerically.

Matplotlib plotting:

```
plt.figure()                                      start new chart

plt.plot(df['timestamp'], df['score'])            draw / plot

plt.title("Anomaly Scores Over Time")

plt.savefig("anomaly_plot.png")                   save plot , embed in html .png file
```

charts are saved temporarily by using tempfile

```
html = template.render()            Injects data into html to create report
```

then file is written:

```
with open("report.html", "w") as f:
   f.write(html)
```

Email , slack alerts:

```
def send_alert(message, webhook_url):
   requests.post(webhook_url, json={"text": message})
```

this helps like a real monitoring agent.

End – to – end pipeline:

Fetch logs → Clean logs → Extract features → Normalize → Train ML model → Detect anomalies → Generate report → (Optional) Send alerts

06. Industry impact:

Skill	Value
Python automation	core SOC workflows
Log parsing	used in security jobs
ML anomaly detection	Modern threat hunting
Reporting	Needed for audit & compliance
API consumption	Connects to real systems

Summary:

Stage	What Happens	Why It Matters
Log ingestion	Fetch logs via API / read CSV	Real-world log sources
Preprocessing	Clean, parse timestamps, extract fields	Prepare structured signals
Feature engineering	Convert raw events into ML-ready numeric features	Helps model learn patterns
Normalization	Standardize values	Prevents biased ML decisions
Isolation Forest	Detect outliers / anomalies	Identifies suspicious behavior
Report generation	HTML + charts	Makes findings human-readable
Alerts (optional)	Slack/Webhook notifications	Real-time incident response