

Sunday, 23 February 2025

Dizertație

AI Powered Hybrid IDS for Cloud Networks = Sistem de detecție a intruziunilor asistat de AI în

Aws keys for AI_HIDS user:

Access key: AKIAXYKJU4QHZ6FTC76Q

Secret access key: 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

1. Observabilitate = componentă cheie în securitate

- Un sistem de detecție eficient **nu înseamnă doar predicții**, ci și **vizibilitate clară** asupra amenințărilor.
- Grafana sau un API REST oferă un **mod concret de a investiga, corela și reacționa** la evenimente.

2. Integrare între detecție automată (AI) și răspuns uman (SOC/human feedback)

- Un API deschide poarta către **feedback uman** (ex: marcare ca fals pozitiv).
- Acest lucru îmbunătățește securitatea în timp prin *model retraining*, dar și prin audit.

3. Se aliniază cu bunele practici din industrie

- Orice sistem HIDS real este **parte dintr-un ecosistem mai mare**: dashboarduri, alerte, integrări, UI.
- AWS, Azure Sentinel, Splunk, etc. — toate oferă **interfețe REST + dashboards**.

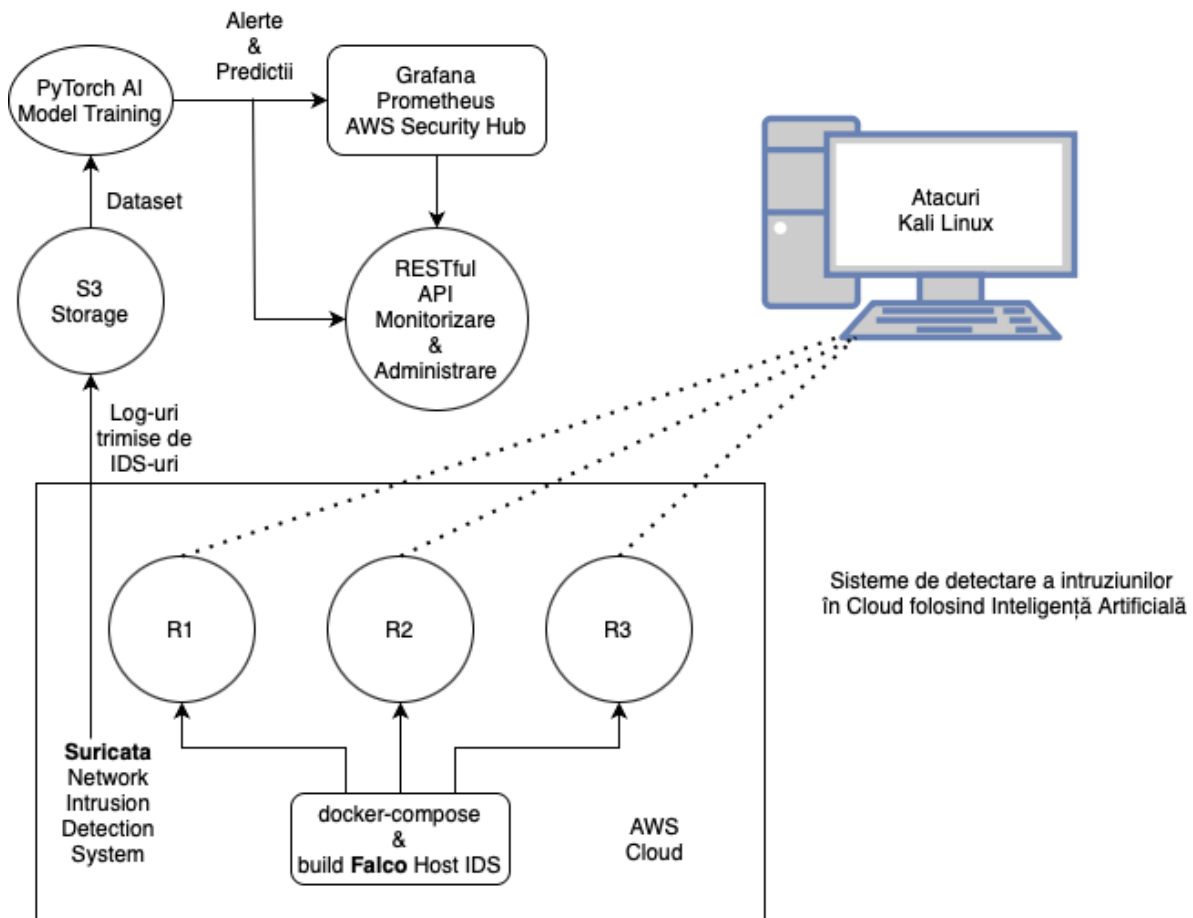
4. Susține obiectivele academice

- Demonstrezi **maturitate în proiectare**, înțelegere completă a întregului ciclu de securitate:
 - colectare → analiză → detecție → vizualizare → reacț

Tipuri de atacuri detectate:

- **DDoS** (trafic anormal crescut)
- **Port scanning** (când cineva caută porturi deschise)
- **SQL Injection, XSS** (atacuri web)
- **Acces neautorizat SSH** (conexiuni suspecte)

- **Exfiltrare de date** (descărcări mari de fișiere)



De implementat:

- ◆ **Protecția modelului AI** → Să nu fie manipulat sau atacat
- ◆ **Securitatea colectării și transmiterii log-urilor** → Evitarea spoofing-ului
- ◆ **Monitorizare și auditare constantă** → Detectare și răspuns rapid
- ◆ **Hardening-ul infrastructurii cloud** → Minimizarea suprafeței de atac
- ◆ **Conformitate cu standardele de securitate** → ISO 27001, NIST, GDPR

📌 Cum implementăm un AI-IDS securizat pentru Cloud?

Vom îmbunătăți arhitectura clasică prin:

- ✓ Utilizarea **IAM Roles și Policies stricte** pentru AWS
- ✓ **Securizarea API-ului de predicție** (rate limiting, autentificare)
- ✓ **Criptarea log-urilor și modelelor AI**
- ✓ **Hardening pentru Suricata**
- ✓ **Zero Trust Security Model**



PROIECT IMPARTIT PE CATEGORII:

- ✓ **Cloud-based IDS** cu NIDS + HIDS
- ✓ **AI/ML pentru detecție** folosind PyTorch
- ✓ **Securitate puternică**
- ✓ **Monitorizare și alertare**



Aspect	Suricata / Falco (HIDS)	Prometheus / Grafana (monitoring)
Scop	Detectare de amenințări și comportamente suspecte	Monitorizarea stării și performanței sistemului
Tip de date	Loguri de securitate (evenimente, semnături, reguli)	Metrici (CPU, memorie, trafic, latență etc.)
Granularitate	Eveniment individual (ex: "shell deschis în container")	Serii temporale continue (ex: "CPU 80% timp de 5 minute")
Frecvență	Pe bază de eveniment (event-driven)	La intervale regulate (time-driven, ex: la 15s)
Format	JSON structurat, focus pe contextul atacului	Time series + labeluri (ex: metrică + server + timp)
Detectează anomalii?	Da, prin semnături sau ML	Nu direct, dar poate alerta pe praguri

antrena modelul AI folosind ambele surse (event + metrică) → *IDS + behavior-aware AI model*.

Suricata/Falco:

-  „A fost detectat un pachet HTTP suspect cu user-agent necunoscut”
-  „Un shell a fost deschis de un proces necunoscut în container”

Prometheus/Grafana:

-  „CPU-ul pe containerul `falco` a fost 95% timp de 3 minute”
-  „Au fost 450 de request-uri/secundă către EC2-ul 10.1.1.2”

Centralizezi vizualizarea și controlul sistemului tău HIDS cu o interfață unificată pentru:

- date brute (Suricata/Falco logs),
- predicții AI (anomalie sau nu),

- metrice de performanță (Prometheus),
- grafice și statusuri actualizate live (prin Grafana sau front-end custom).

Definirea Arhitecturii Generale




 Înainte de a scrie cod, trebuie să decidem **unde** și **cum** vom implementa sistemul.

 **Componente principale:**




1. **Suricata (NIDS) & Falco (HIDS)** – pentru captarea traficului și activităților de pe servere.
2. **AWS S3** – pentru colectarea și stocarea logurilor.
3. **AWS Lambda** – pentru procesarea logurilor.
4. **PyTorch AI Model** – pentru analiza și detecția atacurilor.
5. **Amazon RDS/Elasticsearch** – pentru stocarea și vizualizarea rezultatelor.
6. **Grafana/Prometheus** – pentru monitorizare și alertare.

Hybrid IDS (Like Yours)

The best modern systems use both:

- Rule-based engine for known attacks 
- AI/ML model for anomaly detection 
- Human-in-the-loop or alert triage layer 

 **Decizii cheie:**

-  Modelul va fi antrenat pe logurile capturate de **Suricata/Falco**.
-  Acesta va fi implementat fie în **AWS Lambda**
-  Vom folosi **FastAPI** pentru API și **Grafana** pentru monitorizare.

EC2 has:

Suricata - IDS and IPS(Intrusion Prevention System) Like a [firewall](#), Suricata provides traffic filtering and monitoring, and provides network administrators with the ability to write and enforce detection rules. Suricata is able to detect common attack vectors such as [port scanning](#), [denial-of-service](#), [pass-the-hash](#), and [brute-force attacks](#)

✅ **NIDS (Network IDS)** → its primary use

- Suricata “sniffs” network traffic on a network interface
- it detects attacks, anomalies, scans, malware, etc., in live traffic
- example: monitoring traffic at the edge of a cloud VPC or on a router interface

⚡ **HIDS-like behavior (but limited)**

- Suricata can analyze **local PCAP files or logs** — this is sometimes called “offline mode”
- but it doesn’t monitor system calls, processes, file changes like a true HIDS
- you *could* point Suricata at local traffic mirror ports, but that’s not its core strength

Falco - IDS

✅ **HIDS (Host IDS)**

- Falco watches what’s happening *inside* a host (or a container)
- it inspects **syscalls, process launches, file modifications, privilege escalations, unexpected network connections from inside the host**
- it’s excellent for Kubernetes and Docker environments, because it can monitor containers at runtime

Example in your project

- Deploy **Suricata** in the cloud to monitor VPC traffic at the network edge → NIDS
- Deploy **Falco** inside your EC2s or Kubernetes nodes → HIDS
- Collect logs from both → send to S3, process with Lambda, analyze with your PyTorch AI/ML pipeline

Cloud:

- Setează cont AWS, configurează CLI
- Creează VPC, EC2, S3 cu Terraform

An **EBS volume for logs** on AWS refers to using an **Elastic Block Store (EBS) disk** attached to an EC2 instance **specifically to store application, system, or service logs**.

Why use an EBS volume for logs?

1. Persistence

Logs are written to an EBS volume, so even if the EC2 instance stops or is terminated, the data on the volume is still there (unless you set it to delete on termination).

2. Separation of concerns

You can mount a separate volume just for logs (e.g., `/var/log`), so if logs grow large, they don't fill up your root disk.

3. Scalability

You can **resize** or **change the volume type** (e.g., to increase IOPS or throughput) if you need more space or performance.

4. Backup and recovery

You can snapshot the EBS volume regularly to S3, so you have backups of your logs.

- Activează Traffic Mirroring → Suricata pe EC2
- Configurează EKS → Falco DaemonSet
- Configurează S3 / DB → centralizare loguri

Pasii urmati:

- main.tf pentru a crea VPC, EC2 instances and S3 bucket

When choosing the **best EC2 instance type** for **Falco** (an open-source runtime security monitoring tool) and **Suricata** (an IDS/IPS network threat detection engine), you'll want to focus on instance types that balance **CPU**, **memory**, and **network performance**. Both tools are **resource-intensive**, particularly **Suricata** in high-throughput environments.

Falco is focused on monitoring system calls and kernel activities. It does not require a ton of resources but benefits from good **CPU performance** for processing system-level events.

Type chosen: `t3.micro`. Reason: Very lightweight; can monitor a small VM or container node with minimal CPU/RAM.

Suricata is a network IDS/IPS/NGFW (Next-Generation Firewall), and it is more resource-intensive than Falco. It benefits from both **high CPU** and **high network throughput**.

Type chosen: `t3.medium`. Reason: Suricata needs more CPU and RAM; `t3.medium` gives 2 vCPUs + 4 GB RAM, which is minimal

- how to get the instance ami and details using aws cli:

```
aws ec2 describe-instances --query 'Reservations[*].Instances[*].[InstanceId, ImageId]' --output text
```

```
aws ec2 describe-images --image-ids <ImageId> --query 'Images[*].[ImageId, Name, Description]' --output table
```

both `t3.micro` and `t3.medium` fully support `hvm` and `x86_64` AMIs (the ami available in the region)

- create an SSH key (for AWS)

```
ssh-keygen -t rsa -b 4096 -f hids
```

```
mkdir -p ~/.ssh
```

```
mv my-ssh-key.pub ~/.ssh/my-ssh-key.pub
```

```
mv my-ssh-key ~/.ssh/my-ssh-key
chmod 600 ~/.ssh/my-ssh-key
```

- no hardcoded values in `main.tf` for the keys

Best practice setup

- ✓ `variables.tf` → declare
- ✓ `terraform.tfvars` → assign

- different security groups declared in the `main.tf` file:
Why separate security groups?

1. Separation of concerns

- Suricata → focuses on intrusion detection at the network level.
- Falco → focuses on runtime security inside the OS or container.

2. Even if both start with SSH allowed, over time:

- Suricata might need custom ports for mirroring or monitoring traffic.
- Falco might need ports for agent communication, logging, or integrations.

```
ssh -i ~/.ssh/my-key admin@<INSTANCE_PUBLIC_IP>
```

`terraform apply -replace=aws_instance.suricata_server` cand modificam ceva la instanță

Falco GPG key

IPv6 disabled — good security hygiene.

After first initial and apply of the terraform config:

Import the role into Terraform

```
terraform import aws_iam_role.ec2_s3_write_role ec2-s3-write-role
```

Import the key into Terraform:

```
terraform import aws_key_pair.debian_key hids
```

```
terraform import aws_s3_bucket.log_bucket hids-logs
```

when installing **Falco on EC2** via **Terraform**, you're mixing **infrastructure provisioning** with **system-level configuration**, which can get messy and fragile. Based on your use case and typical DevOps patterns, here's a breakdown:

✓ Recommended: Install Falco in Docker

Why this is better:

- **Separation of concerns:** Terraform sets up infrastructure; Docker manages runtime environments.
- **Fewer compatibility issues:** Falco installation often requires matching kernel headers, which is tricky on fresh EC2 AMIs.
- **Reproducibility:** You can version your Dockerfile or `docker-compose.yml`, and reuse it anywhere.
- **Portability:** You can run the same Falco container on local, EC2, or even Kubernetes.

Falco default rules

You **do not need** to run `apt install falco` inside the container.

The `falcosecurity/falco` image is **built specifically** to run Falco with all dependencies included.

You simply **mount necessary host directories** and start the container with the right privileges and options (as you're doing).

```
sudo docker run -d --name falco --privileged \
-v /var/run/docker.sock:/host/var/run/docker.sock \
-v /dev:/host/dev \
-v /proc:/host/proc:ro \
-v /boot:/host/boot:ro \
-v /lib/modules:/host/lib/modules:ro \
-v /usr:/host/usr:ro \
-v /etc:/host/etc:ro \
-v /opt/falco/logs:/var/log/falco \
falcosecurity/falco:latest \
sh -c "falco -o file_output.enabled=true -o file_output.filename=/var/log/falco/falco.log"
```

Goal: Write an AWS Lambda function that:

1. Is triggered when a `.log` file lands in S3 (from Suricata or Falco)

2. Reads and parses that log file
3. Extracts features / structures the data
4. Saves a cleaned/structured version back to S3 (`/processed/`)
5. (Later) that processed version will be consumed by the PyTorch model

Added to s3 terraform config for aws lambda:

- **Bucket Notification Block** (for Lambda triggers):
 - Set up `aws_s3_bucket_notification` (as shown before).
- **IAM Permissions:**
 - Ensure `aws_lambda_permission` allows S3 to invoke the Lambda.

Why `etl_processor.py`?

- **ETL** stands for:
 - **Extract** (from S3 log files),
 - **Transform** (into structured JSON),
 - **Load** (back into a different S3 path).
- **processor** indicates it's doing logic or transformations, not just reading or storing.

So:

👉 `etl_processor.py` = a clear, descriptive name for a Lambda function that processes raw Suricata/Falco logs from S3.

Lambda Function: Purpose

- Triggered when a `.log` file lands in `s3://.../raw/`
- Parses the log (Suricata format)
- Outputs structured JSON to `s3://.../processed/filename.json`

zip function.zip etl_processor.py =zips the lambda function python code

Ce face acest cod după update-ul prin care adăuga etichete pe baza de reguli

Suricata	Falco
detectează acces la metadata AWS	detectează shell-uri suspecte
verifică user-agent necunoscut	detectează comenzi <code>sh</code> , <code>bash</code>
verifică dacă status \neq 200	detectează <code>user=root</code> + <code>BusyBox</code>
verifică dacă IP sursă nu e intern	analizează textul din <code>rule</code>

Terraform Deploy Flow

Each time you update `etl_processor.py`, you must:

1. Rebuild the ZIP:
`cd lambda`
2. `zip ./dist/etl_processor.zip etl_processor.py`
3. Run:
4. `cd terraform`
5. `terraform apply`

The AI model should learn to classify log entries (from Suricata or Falco) as either:

- **Benign**
- **Suspicious**
- **Malicious**

DATASET folosit: <https://www.unb.ca/cic/datasets/ids-2017.html>

Modelul AI – Etape de Implementare

1. Pregătirea datelor

Suricata (exemplu de features):

- `src_ip, dest_ip, method, url, status, user_agent`

Falco (exemplu de features):

- `proc.exe_path, proc.name, proc.pname, user.name, rule, priority`

Acțiuni:

- Normalizează câmpurile comune (e.g., IP, comenzi, useri).
- Aplică encoding pentru stringuri (ex. one-hot sau embedding pentru comenzi și useri).
- Poți folosi PyTorch sau scikit-learn pentru un model de tip RandomForest / LSTM / Feedforward NN.

2. Construirea modelului

Sugestie 1 (baseline rapid):

- Model: `RandomForestClassifier` din scikit-learn.
- Target: `malicious` (etichetare manuală inițială, apoi semiautomatizată).
- Avantaj: interpretabil, rapid.

AMALES:

Tip de model: Sistem AI de tip IDS (Intrusion Detection System)

- **Scop:** clasificarea fiecărui eveniment (log) ca fiind:
 - 0 → **Benign** (comportament normal)
 - 1 → **Atac** (comportament anormal / malițios)

Este un **sistem de detecție a intruziunilor (IDS)** bazat pe învățare automată, de tip **binary classification**.



Avantaje ale abordării în două etape

1. Faza de pre-antrenare (cu datasetul oficial)

- Ai un **set de date curat, etichetat**, deja procesat și larg acceptat în cercetare.
- Poți:

- verifica pipeline-ul de `preprocessing + training + eval`,
- testa modele diferite (RandomForest, XGBoost, etc.),
- înțelege mai bine relația dintre features și label.

2. Faza de fine-tuning sau reantrenare (cu Suricata / Falco)

- Folosești loguri reale, poate chiar din infrastructura ta sau din scenarii simulate.
- Poți:
 - adapta modelul la tipul de date produs de HIDS-urile tale,
 - rafina comportamentul modelului pe zgomot real,
 - construi un model *domain-specific*.

Workflow de antrenare și inferență

A. Training (offline)

1. Rulează un job local (Jupyter Notebook sau EC2 spot) care:
 - Citește toate fișierele din `processed/`.
 - Preprocesează datele (tokenizare, normalizare).
 - Salvează modelul antrenat în format `.pt` (PyTorch) sau `.pkl` (scikit-learn).
2. Urcă modelul într-un bucket S3: `hids-models/latest/model.pt`.

B. Inferență (real-time or batch)

1. Creează un nou Lambda sau API Gateway + Lambda care:
 - Se activează pe baza unui nou obiect în `processed/`.
 - Încarcă modelul din `hids-models/latest/model.pt`.
 - Rulează predicția.
 - Trimite rezultatul spre:
 - un alt bucket `predictions/`,
 - o bază de date (e.g., DynamoDB),

- sau un endpoint Prometheus (prin pushgateway) pentru Grafana.

Grafana + Prometheus Vizualizare

1. **Prometheus pushgateway:** AI-ul poate trimite scorul/anomalia detecției ca metrică.

- `intrusion_score{source="suricata", rule="X", model="v1"}`

2. **Grafana panels:**

- Un panou per model + sursă (Falco/Suricata).
- Anomalii în timp.
- Top useri/containere suspecte.



Posibilă extensie REST API

Da, o interfață RESTful (Flask/FastAPI) pentru a oferi:

- Vizualizare scoruri,
- Query logs by time/rule,

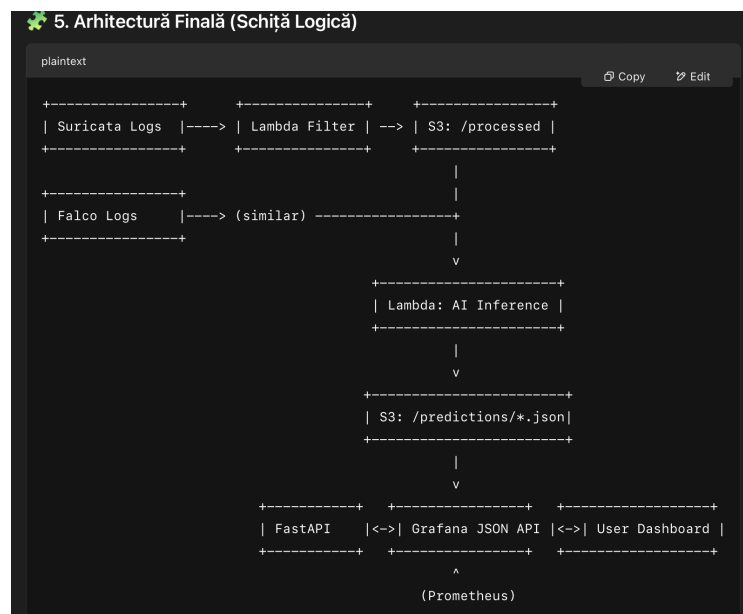
MODEL AI:

Etape:

1. Convertire `.json` în `.csv` sau `DataFrame`.
2. Preprocesare: transformări, codificare categorică, scalare.
3. Împărțire train/val/test.
4. Antrenare
5. Salvare (sau TorchScript e deep

Infrastructură

Deployment:



date:

model.

ca `.pkl`

`.pt` dacă learning).

AI

- **Lambda #2 (Inference):** rulează la încărcarea fișierelor noi în **processed/** și face inferență → predicții.
- **Output:** JSON cu flag `{"is_intrusion": true/false}` + scor + date sursă.
- Stocat în: **predictions/** în același bucket.

Vizualizare & API

Grafana:

- **Surse de date:**
 - Prometheus → metrice de sistem.
 - **Loki** sau **CloudWatch Logs** → loguri brute.
 - **S3 JSON (via API)** → predicții model.
- Vizualizări:
 - Timeline cu evenimente suspecte.
 - Corelații resurse-suspiciune.
 - Anomalie rate.

RESTful API (FastAPI):

- Endpoint-uri:
 - **GET /predictions** → toate evenimentele detectate.
 - **GET /predictions/{id}** → detalii despre o predicție.
 - **GET /stats** → anomalii pe intervale.
- Scop:
 - Suport pentru Grafana (data source JSON API plugin)
 - Acces extern pentru dashboard-uri securizate.

Etape de implementare a modelului AI:

1. Pregătirea datelor

Suricata (exemplu de features):

- `src_ip, dest_ip, method, url, status, user_agent`

Falco (exemplu de features):

- `proc.exepath, proc.name, proc.pname, user.name, rule, priority`

Acțiuni:

- Normalizează câmpurile comune (e.g., IP, comenzi, useri).
- Aplică encoding pentru stringuri (ex. one-hot sau embedding pentru comenzi și useri).
- Poți folosi PyTorch sau scikit-learn pentru un model de tip RandomForest / LSTM / Feedforward NN.

2. Construirea modelului

Workflow de antrenare și inferență

A. Training (offline)

1. Rulează un job local (Jupyter Notebook sau EC2 spot) care:
 - Citește toate fișierele din `processed/`.
 - Preprocesează datele (tokenizare, normalizare).
 - Salvează modelul antrenat în format `.pt` (PyTorch) sau `.pkl` (scikit-learn).
2. Urcă modelul într-un bucket S3: `hids-models/latest/model.pt`.

B. Inferență (real-time or batch)

1. Creează un nou Lambda sau API Gateway + Lambda care:

- Se activează pe baza unui nou obiect în `processed/`.
- Încarcă modelul din `hids-models/latest/model.pt`.
- Rulează predicția.
- Trimite rezultatul spre:
 - un alt bucket `predictions/`,
 - o bază de date (e.g., DynamoDB),

Grafana + Prometheus Vizualizare

1. **Prometheus pushgateway:** AI-ul poate trimite scorul/anomalia detecției ca metrică.
 - `intrusion_score{source="suricata", rule="X", model="v1"}`
2. **Grafana panels:**
 - Un panou per model + sursă (Falco/Suricata).
 - Anomalii în timp.
 - Top useri/containere suspecte.



Posibilă extensie REST API

Da, o interfață RESTful (Flask/FastAPI) pentru a oferi:

- Vizualizare scoruri,
- Query logs by time/rule,
- Re-analiză a unui log,
- Admin UI (opțional),
este utilă **fără să divaghezi** de la tema securității — din contră, adaugă valoare practică lucrării tale.

AI Training Notebook – Hybrid IDS (HIDS)

Obiectiv: Antrenarea unui model AI pe date procesate din Suricata (rețea) și Falco (host) pentru detecția intruziunilor.

Recomandări:

- `notebooks/`: conține Jupyter Notebooks pentru antrenare și testare.
- `data/`: copie locală a fișierelor procesate din S3 (pentru dezvoltare offline).
- `models/`: modele antrenate și salvate (`.pt` dacă folosești PyTorch).
- `utils/`: funcții de preprocesare reutilizabile.
- `README.md`: explicații despre cum se rulează notebook-urile, dependențe, etc.

În notebook-urile de training:

1. **Descarci fișierele `.json` din S3** în folderul `ai/data/suricata/` și `ai/data/falco/`.
2. Apoi, **apelezi funcțiile de preprocesare** din `utils/preprocessing.py` peste aceste fișiere **pentru a obține un `DataFrame` gata de antrenare**.

Etapele AI din proiectul tău HIDS

1. Training (local)



Ce se întâmplă aici:

- Descarci fișierele procesate din S3 (`processed/suricata/` și `processed/falco/`).
- Rulezi un **notebook Jupyter** local (`ai-hids/ai/training_notebook.ipynb`).
- Aplici funcțiile de:
 - Parsare JSON → `DataFrame`
 - Preprocesare (curățare, encoding, normalizare)
 - Extracție de caracteristici
 - Antrenare model (de ex: `RandomForest`, `SVM`, etc.)
- Salvezi modelul antrenat local: `ai/models/hids_model.pkl`.



Poți salva și un `scaler.pkl` sau `encoder.pkl` dacă ai făcut preprocesări care trebuie repetate la inferență.

Inferență (în cloud)

☁ Ce se întâmplă aici:

- Suricata/Falco rulează în cloud → trimit loguri în S3.
- Lambda procesează și salvează loguri în `processed/`.
- Un nou **serviciu (ex: AWS Lambda sau un container cu REST API)**:
 - Încarcă modelul antrenat (din S3 sau inclus în imagine).
 - Încarcă noile loguri procesate.
 - Rulează funcțiile de preprocesare + `model.predict(...)`.
 - Trimite predicțiile:
 - spre **Grafana** (via Prometheus PushGateway / Loki).
 - sau spre o interfață REST/alerting.

De ce faci training local?

- Pentru că training-ul poate consuma resurse mari (CPU, RAM, GPU).
- Ai mai mult control și poți testa în Jupyter Notebook.
- După ce ești mulțumită, salvezi modelul în S3 → și îl folosești în cloud pentru inferență.

Etapă	Locație	Tehnologii implicate
-----	-----	-----
Training	Local	Jupyter, Scikit-learn, Pandas
Salvare model	Local → S3	pickle, boto3
Inferență	Cloud	Lambda / Container / API + Model
Vizualizare	Cloud	Grafana + Prometheus/Loki

ROLUL FISIERELOR DIN FOLDERUL AI:

ai/

|—— notebooks/

| |—— training_suricata.ipynb

```
|   |—— training_falco.ipynb
|
|—— data/
|
|   |—— suricata/
|
|   |—— falco/
|
|—— models/
|
|   |—— model_suricata.pt
|
|   |—— model_falco.pt
|
|—— utils/
|
|   |—— preprocessing.py
|
|   |—— model.py
|
|   |—— predict.py
|
|—— README.md
```

utils/preprocessing.py

Rol: Conține funcții pentru încărcarea și procesarea datelor brute din fișierele JSON deja procesate de Lambda și salvate în S3 (pe care le-ai descărcat local în `data/suricata/` și `data/falco/`).

Fișierul tău de preprocesare:

- Extrage informații relevante din logurile Suricata și Falco
- Normalizează și codifică aceste date
- Le pregătește pentru a fi introduse într-un **model de machine learning (ML)**

Dar: fără o coloană `label`, modelul nu are „adevărul” pe care să îl învețe. Deci **modelul tău AI încă nu știe ce e „atac” și ce nu.**

Funcționalități tipice:

- Încărcarea fișierelor `.json` din `data/`

- Extragerea caracteristicilor relevante (e.g. `src_ip`, `method`, `url`, `status`, `user_agent` pentru Suricata)
- Conversia în Pandas DataFrame
- Codificare, normalizare, label encoding

Codul:

Funcție	Ce face nou?
<code>load_json_files</code>	Încărcare universală JSON / JSON Lines
<code>preprocess_suricata</code>	Codifică label-uri + normalizare numerică
<code>extract_falco_features</code>	Extractor simplu pe baza regex din câmpul <code>output</code>
<code>preprocess_falco</code>	Aplică <code>extract_falco_features</code> pe fiecare log Falco

`notebooks/training_suricata.ipynb` și `training_falco.ipynb`
 Conțin:

Încărcarea datelor din `data/`
 Preprocesare cu funcțiile din `preprocessing.py`
 Antrenarea modelului definit în `model.py`
 Salvarea modelului în `models/`

`utils/`
 Conțin:

`model.py`
 Definirea arhitecturii modelului (ex: rețea neurală simplă)
 Clasa PyTorch Net

`predict.py`
 Funcții pentru a încărca un model salvat
 Realizarea inferenței (predicții)

`preprocessing.py`
 Cod pentru citirea datelor din fișiere `.json`
 Funcții pentru extragerea caracteristicilor (feature extraction)
 Preprocesarea datelor (vectorizare, normalizare, etc.)

 `utils/model.py`

Rol: Definește arhitectura modelului AI. Acest fișier este folosit atât în etapa de antrenare, cât și pentru inferență. **Definește arhitectura rețelelor neuronale** pentru datele Suricata și Falco — atât pentru **antrenare**, cât și pentru **inferență**.

Funcționalități tipice:

- Clase de modele (e.g., `SuricataNet(nn.Module)`, `FalcoNet(nn.Module)`)
- Funcții de creare și configurare a modelului

Ce trebuie să conțină **model.py**:

1. Clase de modele

- `SuricataNet(nn.Module)` — adaptată la numărul de features rezultate din `preprocess_suricata()`
- `FalcoNet(nn.Module)` — adaptată la cei 4 features extrași din output în `preprocess_falco()`

2. Funcții opționale de configurare (dacă vrei flexibilitate, dar nu sunt obligatorii)

- ex: `def create_model(input_size: int) -> nn.Module:`

Utils/ directory:

✓ **Rol:** Cod reutilizabil — modularizare logică de antrenare, inferență, preprocesare.

- `preprocessing.py`: citire, curățare, encoding, transformări
- `predict.py`: cod pentru a aplica modelul pe noi loguri
- `model.py`: eventual, logica pentru salvare/încărcare model, metadata, scoring etc.

● **Este necesar: da**, dar cu adaptare:

- **Dacă ești interesată doar de etichetele "benign/atac"**, `predict.py` va conține doar:
 - încărcare model

- preprocesare minimală a unui nou log
- `model.predict()` → label

utils/predict.py

Rol: Se ocupă cu încărcarea modelului antrenat și aplicarea lui pe date noi pentru a genera predicții. Folosit mai ales în cloud.

Funcționalități tipice:

- Încărcarea modelelor `.pt` din `models/`
- Procesarea datelor noi venite (eventual dintr-un API sau din Lambda)
- Returnarea predicțiilor (e.g., `malicious`, `benign`)

/training_suricata.ipynb & training_falco.ipynb

Rol: Conțin pașii de antrenare pentru fiecare model. Aici execuți codul în ordine pentru:

1. Încărcarea și curățarea datelor din `data/`
2. Apelul funcțiilor din `preprocessing.py`
3. Crearea modelului (din `model.py`)
4. Antrenare, validare, salvare model `.pt`

✓ **Rol:** Conține notebook-uri pentru antrenarea individuală a modelelor.

- `training_suricata.ipynb`
- `training_falco.ipynb`

🟢 **Este necesar:** da — sunt locul unde rulezi manual pașii de:

- Parsare JSON → DataFrame
- Preprocesare
- Feature extraction
- Antrenare model

- Salvare model în `ai/models/`

data/

Rol: Aici salvezi local fișierele JSON descărcate din S3 `processed/suricata/` și `processed/falco/`, pentru a putea fi citite în notebookuri.

models/

Rol: Aici salvezi modelele antrenate `.pt`, care vor fi folosite ulterior pentru inferență în cloud. ✓ **Rol:** Depozitează modelele antrenate, salvate în format `.pt` (PyTorch)

Lambda functions:

Componentă	Rol	Observații
-----	-----	-----
-----	-----	-----
<code>lambda/preprocess_logs.py`</code>	Prelucrează și normalizează logurile brute	Deja implementat ✓
<code>lambda/infer_logs.py`</code>	Încarcă modele AI și rulează inferență	Nouă funcție, fără suprapunere
S3 trigger	Poți activa inferența pe loguri noi (opțional)	Sau rulezi periodic prin EventBridge/cron

Ordinea utilizării:

1. **preprocessing.py** → folosit în notebook pentru a încărca și transforma logurile.
2. **model.py** → folosit în notebook pentru a antrena și salva modelul.
3. **predict.py** → folosit după ce modelul este antrenat, pentru a face inferențe pe loguri noi.

Arhitectură automată bazată pe evenimente S3



Fluxul complet:

1. **Suricata / Falco** → generează loguri
2. **Lambda A** (`process_logs`) → normalizează → salvează în `s3://hids-logs/processed/{falco,suricata}` aici se întâmplă etichetarea pentru model
3. **Lambda B** (`run_inference`) → declanșată automat de S3 când se scrie un nou fișier `.json`
→ încarcă modelul `.pt` → rulează inferență → salvează output (ex: `s3://hids-logs/inference-results/`)



ETAPA 1: Antrenare model AI (manual/local)



Scop: Creezi modelele `model_falco.pt` și `model_suricata.pt`.



Pasul 1: Descarci datele din S3 local

bash

CopyEdit

```
aws s3 sync s3://hids-logs/processed/falco/ ai/data/falco/
```

```
aws s3 sync s3://hids-logs/processed/suricata/ ai/data/suricata/
```



Pasul 2: Rulezi notebook-urile

- `notebooks/training_falco.ipynb`
- `notebooks/training_suricata.ipynb`

→ Se folosesc funcțiile din `utils/preprocessing.py`.



Output: modelele salvate local

- `models/model_falco.pt`
- `models/model_suricata.pt`

ETAPA 2: Inferență automată cu Lambda

◆ **Scop:** De fiecare dată când apare un fișier nou în:

- `processed/falco/` → rulează `run_inference_falco`
- `processed/suricata/` → rulează `run_inference_suricata`

 **Trigger:**

- S3 bucket notifications → Lambda

 **Lambda:**

- Încarcă modelul (din ZIP sau S3)
- Rulează preprocesare + inferență
- Scrie output în `s3://hids-logs/inference-results/{falco|suricata}/`

Ce avem de făcut:

1. Cod pentru Lambda B (`run_inference.py`)

Acest script va:

- primi evenimentul S3
- citi fișierul `.json` procesat
- aplica preprocesare din `utils/preprocessing.py`
- încarcă modelul `.pt`
- generează predicții
- salvează rezultatele în S3

2. Terraform:

- resursă `aws_lambda_function` pentru `run_inference`
- eveniment `aws_s3_bucket_notification` pe prefixele:
 - `processed/falco/`
 - `processed/suricata/`

3. IAM Role + Policy:

Funcția Lambda B are nevoie de acces:

- `s3:GetObject` pentru `hids-logs/processed/*`
- `s3:PutObject` pentru `hids-logs/inference-results/*`

Structura folderelor și responsabilitățile lor

- **notebooks/**
Aici se găsesc *notebook-uri Jupyter* pentru **explorare, antrenare, testare**, adică un mediu interactiv.
În mod tipic:
 - `training_suricata.ipynb` → cod pentru antrenarea modelului Suricata
 - `training_falco.ipynb` → cod pentru antrenarea modelului Falco
 - Poți avea și notebook-uri pentru inferență/testare, dar ideal sunt separate sau într-un alt folder (**inference**/dacă vrei să faci ordine)
- **utils/**
Funcții și scripturi reutilizabile: preprocessing, definirea arhitecturii modelului (`model.py`), predicții (`predict.py`).
- **models/**
Aici salvezi modelele antrenate serializate (`model_suricata.pt`, `model_falco.pt`).
Acestea sunt artefacte binare (fișiere PyTorch `state_dict` sau modele întregi), NU cod.
- **data/**
Datele brute sau procesate (în cazul tău, logurile JSON sau extrase din S3).

Răspunsuri la întrebările tale:

1. `mode_falco.pt` și `model_suricata.pt` – ce conțin?

Sunt *modele antrenate*, de obicei serializări ale rețelelor neuronale PyTorch (fișiere `.pt`). Nu conțin cod, ci doar parametrii învățați (greutăți, bias-uri etc.). Ele sunt generate după rularea procesului de antrenare (de exemplu, în notebook-urile din `notebooks/`).

2. Unde se face training-ul?

În notebook-urile din folderul `notebooks/`. Aceste notebook-uri:

- încarcă datele cu `preprocessing.py`
- încarcă definiția modelului din `model.py`
- antrenează modelul pe datele preprocesate
- salvează modelul antrenat în `models/` ca fișier `.pt`

3. Unde se face inferența?

Ideal ar fi să ai o parte separată pentru inferență, fie în notebook-uri dedicate (ex. `inference_suricata.ipynb`), fie într-un script sau funcție în `utils/predict.py`.

Ulterior, codul de inferență poate fi rulat în cloud, într-o Lambda function sau alt mediu.

Sumar workflow antrenare

1. În `notebooks/training_suricata.ipynb`:
 - `from utils.preprocessing import preprocess_suricata`
 - `from utils.model import SuricataNet`
 - Încarci și preprocesezi datele, antrenezi modelul, salvezi modelul în `models/model_suricata.pt`
2. În `notebooks/training_falco.ipynb`: similar, dar cu Falco.
3. În `utils/predict.py`:
 - Funcții pentru inferență (încărcare model, preprocess, predict)

Pașii pentru testare

1. Verifică dacă ai datele pregătite local

- Logurile preprocesate în folderele:
 - `data/suricata/`
 - `data/falco/`

Să conțină fișiere JSON sau JSONL conform formelor așteptate.

2. Verifică dacă ai modelele antrenate

- Fișierele cu greutatea ale modelelor:
 - `models/model_suricata.pt`
 - `models/model_falco.pt`

Dacă nu ai încă modelele antrenate, va trebui să rulezi notebook-urile de training corespunzătoare pentru a le genera.

3. Pregătește mediul Python

- Ai nevoie să ai instalate librăriile:

```
bash
```

[CopyEdit](#)

```
pip install torch pandas scikit-learn
```

- Asigură-te că fișierul `predict.py` e actualizat cu codul final din mesajul anterior.

4. Rulează scriptul de predicție

În terminal, din directorul proiectului:

```
bash
```

[CopyEdit](#)

```
python predict.py
```

5. Interpretarea rezultatelor

- Scriptul va printa un array de probabilități pentru fiecare intrare din datele preprocesate.
- Valorile aproape de 1 indică posibilă intruziune (sau comportament suspect), iar valorile aproape de 0 indică normal.

6. Verificări suplimentare

- Dacă scriptul aruncă erori legate de dimensiuni, verifică să fie potrivită forma datelor de input cu așteptările modelului.
- Dacă modelele lipsesc, rulează notebook-urile de training pentru Suricata și Falco:
 - `notebooks/training_suricata.ipynb`
 - `notebooks/training_falco.ipynb`

```
import torch
```

```
from torch.utils.data import DataLoader, TensorDataset
```

```
import torch.optim as optim
```

```
import torch.nn.functional as F
```

```
from utils.model import SuricataNet
```

```
from utils.preprocessing import preprocess_suricata
```

```
# 1. Preprocesăm datele (exemplu)
```

```
df = preprocess_suricata("data/suricata")
```

```
# 2. Pregătim tensorii pentru antrenare
```

```
X = torch.tensor(df.values, dtype=torch.float32)
```

```
# Presupunem că ai o coloană 'label' cu 0/1 pentru clasificare
```

```
y = torch.tensor(df['label'].values, dtype=torch.float32).unsqueeze(1) # shape (N,1)
```

```
dataset = TensorDataset(X, y)
```

```
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

```
# 3. Inițializăm modelul și optimizatorul
```

```

model = SuricataNet(input_size=X.shape[1])

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.BCEWithLogitsLoss() # combină sigmoid + binary cross entropy


# 4. Loop simplu de antrenament

model.train()

for epoch in range(10): # 10 epoci

    for batch_x, batch_y in dataloader:

        optimizer.zero_grad()

        outputs = model(batch_x) # output = logits

        loss = criterion(outputs, batch_y)

        loss.backward()

        optimizer.step()

    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

```

Ce tip de AI poți avea?

1. Supervised Learning — alegerea finală

- Tip: Clasificator (de exemplu, RandomForest, Logistic Regression, Neural Network)
- Date: caracteristici + etichete (ex: 0 = benign, 1 = atac)
- Cum funcționează: învață să prezică dacă o linie de log e periculoasă sau nu

👉 Necesită adăugarea unei coloane `label` în `df`:

```

python
CopyEdit
df["label"] = (df["dest_ip"] ==
"169.254.169.254").astype(int)

```

Random Forest Classifier

Model clasic, bazat pe arbori de decizie

✓ Avantaje:

- **Ușor de folosit:** Nu ai nevoie de GPU, funcții de activare sau tuning complicat.
- **Funcționează bine** pentru date tabulare (cum sunt logurile Suricata/Falco).
- **Robust** la date zgomotoase.
- Îți oferă **importanța fiecărei trăsături** (feature importance) – util pentru interpretare.

✗ Dezavantaje:

- Nu e bun la învățarea din date secvențiale, cum ar fi text sau imagini.
- Dacă datele sunt foarte mari, modelul poate deveni lent.

Caracteristici utile (features) FALCO:

- `evt_type`
- `user`
- `user_uid`
- `process`
- `command`
- `exe_flags` (poate fi binarizat per flag)
- `priority` (convertit la scor numeric)
- `container_name`

✓ Se pot extrage și categorice (transformate în one-hot) sau numerice. Se poate adăuga o **etichetă de clasă** (`label`) cum ar fi `malicious=1`, dacă e cunoscută natura evenimentului.

Caracteristici utile (features) SURICATA:

- `method` (categorică)
- `url` (eventual categorizare prin tokenizare sau hashing)

- `status` (numeric)
- `user_agent` (preprocesare / hashing)
- `src_ip, dest_ip` (pot fi convertite în tipuri / ASN-uri / grupuri sau ignorate dacă e nevoie de abstractizare)

✅ Dacă ai informații despre evenimente legitime vs. suspicioase (de ex. acces la metadata API de pe o instanță compromisă), poți marca logurile ca `malicious=1` sau `0`.

Condiții pentru a fi potrivite pentru RandomForest:

- Transformare într-un **DataFrame** (ex: Pandas)
- Fiecare log → un rând, fiecare câmp → coloană (feature)
- Câmpuri text (categorice) convertite în numerice (OneHotEncoder, LabelEncoder, hashing)
- Adăugare câmp `label` (ex: `0 = benign, 1 = malicious`) pentru supraveghere

Modelele de ML (inclusiv RandomForest) **nu pot învăța din șiruri de caractere** ("root", "PUT", "sh"), ci doar din **valori numerice**.

Am ales label encoding

✅ LabelEncoder

- **Ce face:** Atribue fiecărei valori un cod numeric.
- "root" → 0, "admin" → 1, "user" → 2
- **Exemplu simplu, rapid.**
- ❌ **Problema:** Creează o **ordine falsă**. Modelul poate crede că `admin (1) > user (2)`.

🟢 **Folosește dacă:** variabila are **doar două valori** (ex: `malicious: yes/no`) sau dacă **nu e importantă ordinea numerică în model** (e ok la RandomForest).

Cum ajută modelul tău AI?

Într-o afacere reală:

- Nu vei avea etichete (`label=1`) în timp real.
- AI-ul tău trebuie să **observe patternuri** suspecte în:
 - Comportamentul rețelei (Suricata)
 - Comportamentul la nivel de sistem/container (Falco)
- Exemplu:
 - Ai un spike de cereri PUT către `169.254.169.254` într-un container nou.
 - Falco spune că s-a deschis un shell ca `root` în același container.
 - Modelul poate semnala o intruziune fără a fi nevoie de etichete explicite.

1. Etichetarea în Suricata — ce înseamnă „dest_ip interesant”?

Adresa `169.254.169.254` este o **adresă specială folosită de AWS** pentru serviciul de **metadata** (ex. tokens IAM, zona de disponibilitate, roluri EC2).

➡ Când vezi într-un log Suricata că cineva face **GET/PUT** către această adresă, e un semn că:

- Un **atacator** din instanță EC2 încearcă să fure metadata (ex. tokenuri de acces).
- Este o **tehnică comună de escaladare a privilegiilor** sau de **exfiltrare** în AWS.

🔗 Deci `dest_ip == "169.254.169.254" + method in ["PUT", "GET"] + uneori url suspicios (/latest/meta-data/iam/...)` → pot fi etichetate cu `label = 1` (intruziune).

🔄 Alte reguli posibile:

- `user_agent` este necunoscut sau lipsă?
- `src_ip` nu e în rețeaua așteptată (ex. nu e `10.x.x.x`)?
- `status != 200`? (deși uneori atacurile reușite returnează și 200)

2. Etichetarea în Falco — alternativa la `exec_shell=1`

Logul Falco este mai textual, dar include fraze-cheie în output. Exemplu:

```
json
CopyEdit
```

"output": "A shell was spawned in a container with an attached terminal..."

➡ Aceasta frază indică **deschiderea unui shell interactiv** în container — comportament care nu ar trebui să apară într-o instanță în producție.

🔍 Caută în output texte suspicioase:

- `shell was spawned`
- `command=sh, command=bash`
- `user=root + proc_exepath=/bin/busybox` → BusyBox e frecvent folosit în scripturi de atac.

🎯 Etichetare:

- Dacă `user=root` și `command=sh` sau `bash` → `label = 1`
- Dacă rule conține fraze ca `Terminal shell in container` sau `Unexpected network connection` → `label = 1`

Workflow explicat pas cu pas

📌 Pas 1: Preprocesare & Etichetare

python

CopyEdit

```
df = preprocess_suricata("suricata-log.json")
# sau
df = preprocess_falco("falco-log.json")
```

```
# Adăugăm o coloană 'label' în funcție de regulile tale
df["label"] = df.apply(regula_ta_de_etichetare, axis=1)
```

📌 Pas 2: Separare în features și etichete

python

CopyEdit

```
X = df.drop("label", axis=1) # doar coloanele care
descriu datele
y = df["label"]             # ceea ce vrei ca modelul
să învețe (0 sau 1)
```

📌 Pas 3: Antrenare

python

CopyEdit

```
model = train_model(X, y)
```

Modelul învață ce tipare (patternuri) duc la `label=1` (atac) și ce duce la `label=0` (normal).

Pas 4: Predicții pe loguri noi

python

CopyEdit

```
X_new = preprocess_suricata("log_nou.json") # fără label  
acum
```

```
y_pred = predict(model, X_new)
```

Modelul îți va returna 0 sau 1 pentru fiecare intrare nouă.

Workflow (recomandat)

1. Preprocesezi logurile și adaugi etichete (`label`) → `df`
2. Împarți în `X = df.drop("label", axis=1)`, `y = df["label"]`
3. Antrenezi modelul cu `train_model(X, y)`
4. Salvezi și folosești în predicții noi cu `predict(model, X_new)`

Ce urmează:

1. Etichetăm datele (adăugăm coloana `label`)
2. Antrenăm un model simplu (de exemplu, `RandomForest` sau rețea neurală)
3. Salvăm modelul (`.pt` sau `.pkl`)
4. Îl folosim pentru predicții (inference)

Pașii implicați în dezvoltarea modelului

1.  Parsare JSON → `DataFrame`

Obiectiv: Conversia logurilor brute în structură tabelară.

```
import pandas as pd
```

```
import json
```

2. Preprocesare

Obiectiv: pregătirea datelor pentru model (curățare + encoding).

a. Curățare:

- Eliminăm coloane nefolositoare: `timestamp`, `output`, etc.
- Completăm NaN (ex: dacă un log Falco nu are `url`).

b. Encoding categorice:

```
from sklearn.preprocessing import LabelEncoder
```

```
encoder = LabelEncoder()
```

3. Extracție de caracteristici (feature engineering)

Obiectiv: Alegerea câmpurilor utile pentru model.

Extragere manuală:

- `evt_type`, `priority`, `rule`, `process`, `command`, `method`, `status`, `user_agent`, `url` pot fi utile.
- Unele pot fi transformate (ex: `url` → lungime URL).
- Adăugare de features binare (ex: `"has_shell" = 1 if process in ['sh', 'bash'] else 0`).

4. Antrenare model RandomForest

a. Definim features și label:

```
python
```

```
CopyEdit
```

```
features = ["status", "url_len", "is_put",  
            "is_terminal_shell", "user_enc"]
```

```
X = df[features]
```

```
y = df["label"] # Ai adăugat tu manual: 0 pentru benign,  
1 pentru atac
```

b. Antrenare:

python

 CopyEdit

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
```


```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)
```

```
clf = RandomForestClassifier(n_estimators=100,
random_state=42)
```

```
clf.fit(X_train, y_train)
```

c. Evaluate:

python

 CopyEdit

```
from sklearn.metrics import classification_report
```

```
y_pred = clf.predict(X_test)
```

```
print(classification_report(y_test, y_pred))
```

Rules for labels:



1. Etichetarea în Suricata — ce înseamnă „dest_ip interesant”?

Adresa 169.254.169.254 este o **adresă specială folosită de AWS** pentru serviciul de **metadata** (ex. tokens IAM, zona de disponibilitate, roluri EC2).



Când vezi într-un log Suricata că cineva face **GET/PUT** către această adresă, e un semn că:

- Un **atacator** din instanță EC2 încearcă să fure metadata (ex. tokenuri de acces).
- Este o **tehnică comună de escaladare a privilegiilor** sau de **exfiltrare** în AWS.



Deci `dest_ip == "169.254.169.254" + method in ["PUT", "GET"]` + uneori url suspicios (`/latest/meta-data/iam/...`) → pot fi etichetate cu `label = 1` (intruziune).



Alte reguli posibile:

- `user_agent` este necunoscut sau lipsă?
- `src_ip` nu e în rețeaua așteptată (ex. nu e 10.x.x.x)?
- `status != 200`? (deși uneori atacurile reușite returnează și 200)

🧠 2. Etichetarea în Falco — alternativa la `exec_shell=1`

Logul Falco este mai textual, dar include fraze-cheie în output. Exemplu:

json

CopyEdit

```
"output": "A shell was spawned in a container with an attached terminal..."
```

➡ Aceasta frază indică **deschiderea unui shell interactiv** în container — comportament care nu ar trebui să apară într-o instanță în producție.

🔍 Caută în output texte suspicioase:

- `shell was spawned`
- `command=sh, command=bash`
- `user=root + proc_exepath=/bin/busybox` → BusyBox e frecvent folosit în scripturi de atac.

🎯 Etichetare:

- Dacă `user=root` și `command=sh` sau `bash` → `label = 1`
- Dacă rule conține fraze ca `Terminal shell in container` sau `Unexpected network connection` → `label = 1`

vrei să construiești un **model cumulativ**, care „învață” și din setul CICIDS și din Suricata **fără să o iei complet de la zero**. Din păcate, modelul tău actual (`RandomForestClassifier`) **nu permite** în mod nativ antrenare incrementală (`partial_fit`), deci nu poate fi extins direct cu noi date fără să fie reantrenat complet.

✅ Ce poți face acum (fără să pierzi tot):

🧠 Opțiunea 1: Treci la un model cu suport pentru `partial_fit`

Modele care suportă incremental learning:

Model	Scikit-learn Class	Suportă <code>partial_fit</code>
Stochastic Gradient Descent	<code>SGDClassifier</code>	✅ DA
Naive Bayes	<code>MultinomialNB</code> , <code>GaussianNB</code>	✅ DA

Passive Aggressive	PassiveAggressiveClassifier	✓ DA
Perceptron	Perceptron	✓ DA

✓ 1. preprocessing.py – nu necesită modificări majore

Acesta e deja compatibil cu `SGDClassifier`, deoarece:

- returnează `X` (features) și `y` (labels) bine preprocesate.
- folosește `LabelEncoder`, ceea ce e în regulă.

prima antrenare pe CICIDS cu `classes=...`

apoi `partial_fit()` pe logurile Suricata sau Falco.

```
LABEL_ENCODER_PATH = '../models/label_encoder_cicids.pkl'
```

Aceasta este calea unde va fi salvat **encoderul de etichete** (**LabelEncoder**) folosit pentru a transforma valorile de tip string (ex: "BENIGN", "DoS Hulk") în valori numerice (0, 1, 2, ...).

Asta e important pentru două motive:

1. Poți decoda predicțiile înapoi în clase umane (ex: 0 → "BENIGN").
2. Poți folosi același encoder când faci predicții pe date noi (Suricata/Falco), ca să menții consistența claselor.

Perfect, Nina! Îți ofer mai jos **versiuni actualizate pentru training_suricata.ipynb și training_falco.ipynb**, care:

1. **Încarcă modelul deja antrenat** pe CICIDS (`SGDClassifier`).
2. **Încarcă LabelEncoder + vectorul classes_**.
3. **Încarcă logurile Suricata/Falco (JSON)**.
4. **Le preprocesează cu același LabelEncoder și aceleași clase**.
5. **Continuă antrenarea modelului cu partial_fit()**.
6. **Salvează modelul updatat**.

Ai două surse de date:

- **Setul de bază (ex: CICIDS sau alt dataset oficial)** — are multe coloane (features specifice: porturi, servicii, payload, etc.)
- **Suricata logs** — doar `timestamp`, `src_ip`, `dest_ip`, etc.

Doar 3 coloane sunt comune. Asta înseamnă că **distribuția de features** este foarte diferită și nu poți aplica direct incremental learning (cum e `partial_fit()` din `sklearn`) decât dacă armonizezi caracteristicile.

Abordarea ta — feature engineering cu:

- **Frecvența accesărilor per IP**
- **Număr de evenimente într-un interval de timp**
- **Tipuri de protocoale folosite**
- **Media/durata sesiunilor TCP**

este foarte potrivită pentru a detecta **DoS/DDoS** (Denial of Service / Distributed Denial of Service).

De ce?

- **DoS/DDoS se caracterizează prin volume mari de trafic sau cereri repetate de la același IP sau de la multe IP-uri către o țintă, într-un interval scurt.**
- Frecvența accesărilor per IP și numărul de evenimente pe interval ajută la identificarea atacurilor în care un atacator încearcă să copleșească un serviciu cu cereri.
- Tipurile de protocoale pot arăta dacă traficul anormal este TCP, UDP, ICMP, care sunt protocoalele cel mai des folosite în atacuri DoS.
- Durata sesiunilor TCP scurte, multiple încercări de conexiuni sau resetări pot indica încercări de flooding sau alte atacuri.

Exact, dacă optezi pentru abordarea de **feature engineering manuală**, lucrurile se schimbă puțin față de metoda de antrenare directă a unui model precum `partial_fit` (presupun că

te referi la o funcție de tip fit) sau un clasificator „sud” (sau SUD classifier — nu sunt sigur, dar poate te referi la un classifier standard, cum ar fi RandomForest sau SVM).

Ce implică abordarea feature engineering?

- În loc să dai direct ca input modelelor **logurile raw sau prelucrate minimal**, construiești manual un set de caracteristici care să reflecte comportamentul rețelei (ex: frecvența accesărilor per IP, durata sesiunilor TCP, numărul de evenimente pe intervale de timp etc).
- Aceste caracteristici sunt apoi normalizate și transformate astfel încât să poată fi comparate sau combinate cu date din dataset-uri oficiale, cum ar fi cel pentru DoS/DDoS.
- Apoi, pe baza acestor features, antrenezi modelul (RandomForest, SVM, NN, etc).

Ce se schimbă?

- **Nu mai antrenezi direct pe loguri brute, ci pe vectori de caracteristici** care extrag semnificația statistică sau structurală din loguri.
- Astfel, modelul învață pe baza acestor features, deci procesul de `fit` rămâne, dar pe un nou format de date (feature vectors), nu direct pe loguri.
- Dacă înainte aveai un pipeline simplu: `loguri prelucrate → model.fit(X, y)`, acum ai: `loguri → feature engineering → vectori caracteristici → model.fit(X_features, y)`

În concluzie

- Nu renunți la antrenarea unui model, doar că inputul pentru antrenare devine un set construit de features.
- Funcțiile tale de fit/train rămân, dar trebuie să le aplici pe noul set de date (features).
- Poți păstra clasificatorul preferat, dar trebuie să adaptezi inputul.

Problema ta:

- **Dataset-ul de bază** (de ex. CICIDS) conține un set de caracteristici diferite față de cele din logurile Suricata/Falco.
- Tu vrei să pornești de la un model antrenat pe dataset-ul de bază (cicids) și apoi să faci **învățare incrementală** pe logurile Suricata/Falco.

Ce trebuie să faci?

1. Alinierea caracteristicilor (Feature alignment)

- **Problema principală** este că vectorii de input trebuie să aibă aceleași caracteristici (features) și în același format pentru ambele seturi de date.
- Deci trebuie să faci **feature engineering** pentru ambele dataset-uri, astfel încât:
 - Să extragi aceleași tipuri de caracteristici din ambele surse (de ex. frecvența accesărilor per IP, număr evenimente per timp, tip protocoale, durată sesiuni TCP etc.)
 - Să normalizezi și scalezi caracteristicile la aceeași scară.

2. Construirea unui vector comun de caracteristici

- Crează o funcție de preprocesare care primește logurile raw și returnează un vector numeric consistent (aceleași set de coloane/features) indiferent de sursă (dataset sau log Suricata/Falco).
- Acest vector va avea aceleași coloane, aceleași tipuri și aceleași scale pentru toate datele.

3. Antrenare inițială pe dataset-ul de bază (CICIDS)

- Antrenează modelul inițial pe dataset-ul oficial, folosind vectorul de caracteristici comun construit la pasul anterior.

4. Învățare incrementală pe logurile reale

- Aplică aceeași funcție de preprocesare pe logurile Suricata/Falco, obținând vectori în același format.
- Folosește `partial_fit` să continui antrenarea modelului inițial cu noile date.
- Astfel modelul va "adapta" cunoștințele pentru caracteristicile și distribuția datelor reale.

Pe scurt:

Problemă	Soluție
Dataset-uri cu caracteristici diferite	Construiește un set comun de caracteristici (feature engineering)
Model inițial pe dataset oficial	Antrenează modelul pe aceste caracteristici comune

Date noi diferite (log Suricata/ Falco)	Transformă-le cu același proces și aplică <code>partial fit</code>
--	---

Cum poți simula un atac DoS controlat în AWS?

- Folosește o instanță EC2 proprie în AWS (ex: Kali Linux).
- Ținta atacului trebuie să fie o altă instanță EC2 în același cont AWS, sau un serviciu pe care îl controlezi strict.
- Setează limite de trafic pentru a evita saturarea rețelei și să nu depășești limitele impuse de AWS (ex: rate limit, firewall-uri).

3. Recomandări practice

- Folosește **VPC Security Groups** și **Network ACLs** pentru a controla traficul.
- Rulează atacul DoS **doar în rețeaua ta VPC privată**, pentru a nu afecta alte resurse sau AWS pe scară largă.

```
# Evaluează
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
BENIGN	1.00	1.00	1.00	19419
DDoS	1.00	1.00	1.00	25724
accuracy			1.00	45143
macro avg	1.00	1.00	1.00	45143
weighted avg	1.00	1.00	1.00	45143

```
# Antrenează modelul
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

▼

RandomForestClassifier

i

?

RandomForestClassifier(random_state=42)

Observație:

Logurile Suricata/Falco pot avea o structură și semnificație diferită față de CSV-urile inițiale. Așa că este important să:

- normalizezi coloanele și feature-urile,
- aplici aceeași etichetare (label 0/1),
- și să faci un feature engineering coerent.

Procesare fișier: ../data/TrafficLabelling/Thursday-WorkingHours-Afternoon-Infiltration.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Monday-WorkingHours.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Friday-WorkingHours-Morning.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

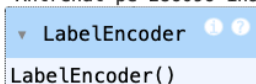
Procesare fișier: ../data/TrafficLabelling/Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Tuesday-WorkingHours.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Wednesday-workingHours.pcap_ISCX.csv
Antrenat pe 286096 instanțe.

Procesare fișier: ../data/TrafficLabelling/Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv
Antrenat pe 286096 instanțe.



✓ Etichete binare: **Benign = 0, Attack = 1**

🔒 De ce este cea mai sigură?

- **Simplifică modelul:** Clasificare binară e mai robustă și mai puțin predispusă la erori când ai puține date pentru unele atacuri (ex: **PortScan** vs **DDoS** vs **Botnet** etc.).
- **Etichetele sunt deja numerice (0/1)** în logurile Suricata/Falco → nu mai ai nevoie de `LabelEncoder`, deci elimini riscul de `KeyError`.
- **Ușor de gestionat** în producție (alarme = 1, normal = 0).
- Modelul învață **pattern general de atac**, nu specific doar la **DDoS**, ci și la **PortScan**, **Botnet**, etc., ceea ce e ideal pentru un sistem HIDS general.

```
# Compară etichetele reale cu predicțiile
for i in range(min(10, n_samples)): # primele 10 exemple
    print(f'Real: {labels[i]}, Predicted: {predicted_labels[i]}')

# Calculează acuratețea
accuracy = (predicted_labels == labels).mean()
print(f'\nAcuratețea estimată: {accuracy:.2%}')

Real: 0, Predicted: 1
Real: 1, Predicted: 1
Real: 0, Predicted: 1
Real: 1, Predicted: 1
Real: 1, Predicted: 1
Real: 0, Predicted: 1
Real: 1, Predicted: 1
Real: 0, Predicted: 1
Real: 1, Predicted: 1
Real: 1, Predicted: 1

Acuratețea estimată: 55.56%
```

Interpretare generală

Ai 9 exemple în total, iar predicțiile modelului sunt:

- **Predicted: 1** pentru toate cazurile.
- Adevăratele valori (real labels) sunt:
 - 5 cazuri cu **label 1 (atac)**
 - 4 cazuri cu **label 0 (benign)**



Acuratețe (Accuracy): 55.56%

Acuratețea este:

num

a

r

predicții corecte

total

=

5

9

≈

55.56

%

totalnumăr predicții corecte = 5 ≈ 55.56%

Adică modelul a prezis corect doar cazurile în care eticheta reală era 1.



Ce înseamnă?

Modelul tău:

- **Prezice mereu 1** (atac), indiferent de input.
- Este un model **dezechilibrat**, adică ignoră cazurile 0 (benign).
- **Detectează toate atacurile (sensibilitate = 100%)**, dar **generează multe alarme false** (adică zice că e atac și când nu e).

Consecințe practice

✓ Pro:

- Nu ratează niciun atac (important pentru securitate).
- Ar putea fi util ca **sistem de detecție conservator**: mai bine să alerteze în plus decât să rateze un atac.

✗ Contra:

- Fals pozitive multe → alerte inutile → oboseală operațională.
- Nu face diferența între trafic legitim și atacuri.

Modelul tău **detectează atacuri**, dar **nu diferențiază bine între benign și malițios**. Este o bază decentă pentru un sistem de securitate cu accent pe **detecție completă**, dar trebuie rafinat pentru a reduce alarmele false.

PENTRU FALCO

```
# Compară etichetele reale cu predicțiile
for i in range(min(10, n_samples)): # primele 10 exemple
    print(f"Real: {labels[i]}, Predicted: {predicted_labels[i]}")

# Calculează acuratețea
accuracy = (predicted_labels == labels).mean()
print(f"\nAcuratețea estimată: {accuracy:.2%}")

Real: 1, Predicted: 1
Real: 0, Predicted: 1
Real: 0, Predicted: 1

Acuratețea estimată: 33.33%
```

```
(kali@kali)-[~]
$ sudo hping3 -S 18.197.32.135 -p 80 --flood

HPING 18.197.32.135 (eth0 18.197.32.135): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
```

hping3 este un **generator de pachete TCP/IP** de linie de comandă, folosit frecvent pentru:

- Teste de penetrare (penetration testing)
- Simulări de atacuri (DDoS, port scan, flood etc.)
- Teste de firewall și IDS (ca în cazul Suricata/Falco)
- Diagnostice de rețea (asemănător cu **ping**, dar mult mai flexibil)

Este extrem de util în testarea și evaluarea unui sistem de detecție a intruziunilor (IDS), deoarece poți **controla complet** fiecare detaliu al pachetelor trimise.

◆ **SYN Flood (simulează un atac clasic DDoS)**

bash

CopyEdit

```
hping3 -S 18.197.32.135 -p 80 --flood
```

- **-S** = trimite pachete TCP cu flag-ul SYN (inițiere conexiune)
- **-p 80** = către portul 80 (HTTP)
- **--flood** = trimite cât de repede permite conexiunea (fără a aștepta răspunsuri)

🔥 Acesta este detectat de IDS-uri (Suricata/Falco) ca **SYN Flood** sau **Port Scan**.

De ce funcționează bine cu Suricata?

Suricata monitorizează pachetele la nivel de rețea și identifică anomalii sau tipare de atac. **hping3** permite generarea acestor tipare, cum ar fi:

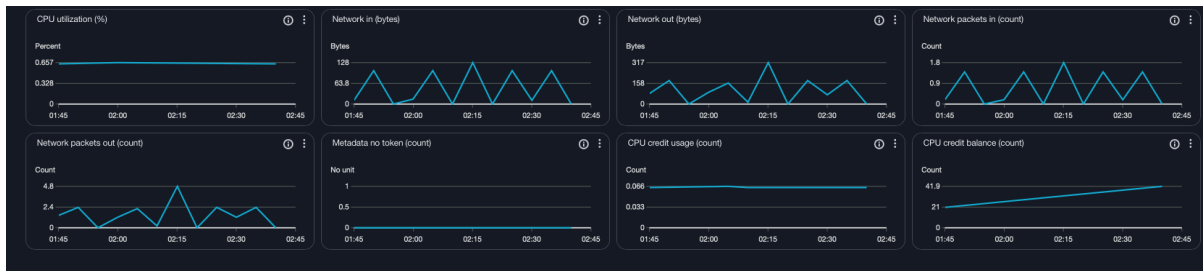
- Pachete SYN repetate fără ACK
- Pachete malformate
- Volume neobișnuite de trafic
- Conexiuni suspecte pe porturi nefolosite

Cum detectează Suricata această comandă?

Suricata are semnături care prind:

- **TCP SYN flood** (excesiv de multe conexiuni SYN fără ACK)
- **Port scan**

- TCP anomaly
- Excessive connections from a single host



Falco nu este ca Suricata — el **nu monitorizează traficul de rețea**, ci **monitorizează activități la nivel de sistem**: deschidere de fișiere sensibile, execuție de comenzi suspecte, deschidere de shell-uri neautorizate, scriere în fișiere critice etc.

Ce tipuri de „atac” poate detecta Falco?

Falco detectează comportamente precum:

- Execuție de comenzi cu `curl` sau `wget` în shell-uri neautorizate.
- Deschiderea de shell (`bash`, `sh`, etc.) în containere.
- Scriere în fișiere din `/etc`, `/bin`, etc.
- Deschidere de fișiere de tip credentiale.
- Citirea fișierelor `/proc`, `passwd`, etc.

Pe instanța Falco (prin SSH) serviciu care urmează a fi atacat

```
sudo apt install -y python3
```

```
python3 -m http.server 80
```

Falco este, de obicei, un sistem de detecție care rulează intern și scrie loguri, deci nu are un server HTTP ascultat implicit pe porturi.

1. Observație importantă:

Falco nu ascultă un port TCP pentru conexiuni, ci e un motor de detecție care monitorizează evenimentele din sistem (sisteme de fișiere, rețea, procese). De aceea, simpla expunere de porturi nu se aplică pentru Falco în mod direct.

2. Totuși, dacă vrei să simulezi atacuri asupra containerului Falco, poți ataca porturile sau serviciile pe care Falco le monitorizează pe instanța EC2 — pentru că Falco "vede" aceste atacuri prin sistemul său de monitorizare.

Cum să simulezi atacuri asupra Falco?

- Atacă serviciile și porturile gazdei (EC2) pe care Falco le monitorizează (ex: HTTP pe port 80, SSH pe 22).
- Falco va detecta aceste atacuri din loguri sau din monitorizarea sistemului.

