

AI PROMPTS FORM

This document details the application of generative AI in the research and development of the Network Traffic Profiler application. Each table represents a prompt sent to an AI.

Context:	PCAP data extraction to Pandas dataframe object
Team Member:	Tim: <input checked="" type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt	Create python code that takes in a PCAP file from a filepath, extracts packets and aggregates them into network flows using the 5-tuple source IP, destination IP, source port, destination port and protocol, returning a Pandas dataframe object. Use Scapy to read the PCAP file.
Response	<p>Below is clean, self-contained Python code that:</p> <ul style="list-style-type: none">✓ Accepts a PCAP file path✓ Uses Scapy to read packets✓ Extracts packet-level metadata (IP addresses, ports, protocol, timestamp, size)✓ Aggregates packets into network flows using the 5-tuple✓ Returns a Pandas DataFrame of flow statistics <p><input checked="" type="checkbox"/> Python Code: PCAP → Flow Extraction with Pandas (5-Tuple Aggregation)</p> <pre>from scapy.all import rdpcap import pandas as pd import os def extract_flows(): # --- Configuration --- PCAP_FILE = "test.pcap" # --- 1. Load PCAP and Extract Packet-Level Data --- if not os.path.exists(PCAP_FILE): print(f"Error: PCAP file '{PCAP_FILE}' not found.") # Exit or handle the error appropriately exit() print(f"Loading packets from {PCAP_FILE}...") # Read file into packets packets = rdpcap(PCAP_FILE) # Initialise an empty container array rows = [] for i, pkt in enumerate(packets): # Check if the packet has an IP layer (to filter out non-IP protocols like ARP/LLC) if "IP" in pkt: src_ip = pkt["IP"].src dst_ip = pkt["IP"].dst protocol = pkt["IP"].proto</pre>

```

        size = len(pkt)
        ts = pkt.time

        # Extract port information from TCP or UDP
        layers
        if pkt.haslayer("TCP"):
            src_port = pkt["TCP"].sport
            dst_port = pkt["TCP"].dport
        elif pkt.haslayer("UDP"):
            src_port = pkt["UDP"].sport
            dst_port = pkt["UDP"].dport
        else:
            # Set port to none for non IP protocols
            src_port = None
            dst_port = None

        # Append row of data
        rows.append([src_ip, dst_ip, src_port,
dst_port, protocol, size, ts, i])

        # Create DataFrame with packet-level data
        df_packets = pd.DataFrame(rows, columns =
            ["src_ip", "dst_ip", "src_port", "dst_port",
"protocol",
            "packet_size", "timestamp", "packet_index"
        ])

        print(f"Successfully extracted {len(df_packets)} IP
packets.")

        # --- Aggregate to Flow-Level Features ---

        # 5-tuple that forms a unique key for each network
        flow_key = ["src_ip", "dst_ip", "src_port",
"dst_port", "protocol"]

        print("Aggregating packets into network flows...")
        # Group packets that share the same tuple together
        df_flows = df_packets.groupby(flow_key).agg(
            # Core flow metrics
            packet_count=("packet_size", "count"),
            byte_count=("packet_size", "sum"),
            avg_packet_size=("packet_size", "mean"),

            # Time metrics
            start_time=("timestamp", "min"),
            end_time=("timestamp", "max"),

            # Drill-down link (stores the index range in the
            original PCAP)
            first_packet_index=("packet_index", "min"),
            last_packet_index=("packet_index", "max")
        ).reset_index()

        # Compute flow duration in seconds
        df_flows["duration"] = df_flows["end_time"] -
df_flows["start_time"]

```

```

# Convert protocol number to a readable string
protocol_map = {6: 'TCP', 17: 'UDP'}
df_flows['protocol_name'] =
df_flows['protocol'].map(protocol_map).fillna(df_flows['proto
col'])

# --- Output Results ---

# Drop raw timestamps for a cleaner output
df_flows = df_flows.drop(columns=["start_time",
"end_time"])

print(f"\nFlow extraction complete. Generated
{len(df_flows)} unique flows.")
print("\n--- Flow Summary Head ---")
print(df_flows)

extract_flows();

```

Context:	Validation Checklist
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input checked="" type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	Give me a checklist of validation specification for a network traffic profiler that can be applied on these extracted csv data (src_ip, dst_ip, src_port, dst_port, protocol, packet_count, byte_count, avg_packet_size, first_packet_index, last_packet_index, duration, protocol_name)
Response:	<p>CSV Validation Checklist</p> <p>1. Column & Schema Checks</p> <ul style="list-style-type: none"> • All required columns exist: src_ip, dst_ip, src_port, dst_port, protocol, packet_count,

- byte_count, avg_packet_size, first_packet_index, last_packet_index, duration, protocol_name
- Required fields are not missing.
- Field types match expected (int, float, str).
- Numeric fields respect min/max bounds:
 - src_port, dst_port:** 0–65535
 - protocol:** 0–255
 - packet_count, byte_count:** ≥ 0
 - avg_packet_size:** 1–65535
 - first_packet_index, last_packet_index:** ≥ 0
 - duration:** ≥ 0

2. IP Address Checks

- Valid IPv4 or IPv6 (RFC-compliant).
- $\text{src_ip} \neq \text{dst_ip}$.
- No multicast addresses:
 - IPv4: 224.0.0.0 – 239.255.255.255
 - IPv6: ff00::/8
- No broadcast (255.255.255.255 for IPv4).
- No loopback:
 - IPv4: 127.0.0.0/8
 - IPv6: ::1

3. Protocol & Port Checks

- protocol is valid IANA number:
 - 1 = ICMP
 - 6 = TCP
 - 17 = UDP
- protocol_name must match protocol (case-insensitive):
 - TCP \leftrightarrow 6
 - UDP \leftrightarrow 17
 - ICMP \leftrightarrow 1
- TCP or UDP flows must have valid port numbers (0–65535).
- ICMP flows must have:
 - $\text{src_port} = 0$
 - $\text{dst_port} = 0$

4. Packet & Byte Checks

- $\text{packet_count} \geq 0$
- $\text{byte_count} \geq \text{packet_count}$
- If $\text{packet_count} == 0 \rightarrow \text{byte_count}$ must be **0**
- avg_packet_size must approximately equal:
 $\text{byte_count} / \text{packet_count}$

tolerance: **±5 bytes**

5. Packet Index Checks

- $\text{first_packet_index} \leq \text{last_packet_index}$
- Index numbers must be non-negative.
- Large gaps ($> 10,000$) can be flagged as anomalies (optional rule).

	<p>6. Duration Checks</p> <ul style="list-style-type: none"> • duration ≥ 0 • If packet_count == 1 \rightarrow duration ≤ 0.001 seconds (single packet \rightarrow almost zero duration) • If packet_count > 1 \rightarrow duration must not be zero • Multi-packet flows lasting longer than 1 hour should be flagged as suspicious. <p>7. Protocol-Port Sanity Checks</p> <ul style="list-style-type: none"> • If DNS (port 53) present: <ul style="list-style-type: none"> ◦ duration < 10s • If HTTPS (port 443): <ul style="list-style-type: none"> ◦ packet_count ≥ 2 • If protocol = 6 (TCP): <ul style="list-style-type: none"> ◦ Ports should not be 0 unless reset or malformed. • If protocol = 17 (UDP): <ul style="list-style-type: none"> ◦ Very high packet_count (>10,000) flagged (optional). <p>8. Duplicate Detection</p> <p>No duplicate flows using the deduplication key: (src_ip, dst_ip, src_port, dst_port, protocol, first_packet_index)</p>
--	--

Context:	Vibe Code to implement Validation on the extracted PCAP data
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input checked="" type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	<p>Based on the following network flow validation rules (checklist below), generate complete Python code to validate extracted network flow data from a PCAP file</p> <p>Extracted data: (src_ip, dst_ip, src_port, dst_port, protocol, packet_count, byte_count, avg_packet_size, first_packet_index, last_packet_index, duration, protocol_name)</p> <p>Checklist:</p> <ol style="list-style-type: none"> 1. Column & Schema Checks <ol style="list-style-type: none"> All required columns exist. Required fields are not missing. Field types match expected (int, float, str). Numeric fields respect min/max bounds: <ol style="list-style-type: none"> src_port, dst_port: 0–65535 protocol: 0–255 packet_count, byte_count: ≥ 0 avg_packet_size: 1–65535 first_packet_index, last_packet_index: ≥ 0 duration: ≥ 0 2. IP Address Checks <ol style="list-style-type: none"> Valid IPv4/IPv6 $\text{src_ip} \neq \text{dst_ip}$

	<ul style="list-style-type: none"> c. No multicast addresses. d. No broadcast (255.255.255.255) addresses. e. No loopback addresses (127.0.0.1 / ::1) <p>3. Protocol & Port Checks</p> <ul style="list-style-type: none"> a. protocol exists in IANA_PROTOCOLS (1=ICMP, 6=TCP, 17=UDP). b. protocol_name matches protocol. c. TCP/UDP flows have valid ports. d. ICMP flows have src_port = dst_port = 0. <p>4. Packet & Byte Checks</p> <ul style="list-style-type: none"> a. packet_count ≥ 0. b. byte_count ≥ packet_count. c. If packet_count = 0 → byte_count = 0. d. avg_packet_size ≈ byte_count / packet_count (tolerance ±5). <p>5. Packet Index Checks</p> <ul style="list-style-type: none"> a. first_packet_index ≤ last_packet_index. <p>6. Duration Checks</p> <ul style="list-style-type: none"> a. duration ≥ 0. b. Single packet flows: duration ≤ 0.001s. c. Multiple packets: duration ≠ 0. d. Multi-packet flows > 1 hour flagged. <p>7. Protocol-Port Sanity Checks</p> <ul style="list-style-type: none"> a. DNS (port 53) → duration < 10s. b. HTTPS (port 443) → packet_count ≥ 2. <p>8. Duplicate Detection</p> <ul style="list-style-type: none"> a. No duplicate flows (key: src_ip, dst_ip, src_port, dst_port, protocol, first_packet_index).
Response:	<p>Here is your complete, clean, final Python code implementing all validation rules from your checklist, including schema checks, IP rules, protocol/port logic, packet/byte checks, duration rules, duplicate detection, and dataset-level output</p> <p>step 2: validate extracted features against rules</p> <pre>import os import pandas as pd import ipaddress import numpy as np ===== FIELD SCHEMA & IANA_PROTOCOLS ===== FIELD_SCHEMA = { "src_ip": {"type": str, "required": True}, "dst_ip": {"type": str, "required": True}, "src_port": {"type": int, "min": 0, "max": 65535, "required": True}, "dst_port": {"type": int, "min": 0, "max": 65535, "required": True}, "protocol": {"type": int, "min": 0, "max": 255, "required": True}, "packet_count": {"type": int, "min": 0, "required": True}, "byte_count": {"type": int, "min": 0, "required": True}, "avg_packet_size": {"type": float, "min": 1.0, "max": 65535.0, "required": True}, "first_packet_index": {"type": int, "min": 0, "required": True}, }</pre>

```

"last_packet_index": {"type": int, "min": 0, "required": True}, "duration": {"type": float, "min": 0.0, "required": True}, "protocol_name": {"type": str, "required": True}, }
IANA_PROTOCOLS = { 1: "ICMP", 6: "TCP", 17: "UDP", }
=====
Helper Function: Validate IP
=====
def validate_ip(ip_str): try: ipaddress.ip_address(ip_str) return True except: return False
def is_private_ip(ip_str): try: return ipaddress.ip_address(ip_str).is_private except: return False
=====
Row-level validation
=====
def validate_row(row): errors = []
# -----
# Schema / type / bound checks
# -----
for field, rules in FIELD_SCHEMA.items():

    if rules.get("required", False) and pd.isna(row.get(field)):
        errors.append(f"{field}: required but missing.")
        continue

    value = row.get(field)
    if pd.isna(value):
        continue

    expected_type = rules["type"]

    if expected_type == int:
        try:
            value = int(value)
        except Exception:
            errors.append(f"{field}: expected int, got {type(value).__name__}")
            continue

    if expected_type == float:
        try:
            value = float(value)
        except Exception:
            errors.append(f"{field}: expected float, got {type(value).__name__}")
            continue

    if not isinstance(value, expected_type):
        errors.append(f"{field}: expected {expected_type.__name__}, got {type(value).__name__}")

```

```

        continue

    if "min" in rules and value < rules["min"]:
        errors.append(f"{field}: {value} < minimum {rules['min']}")

    if "max" in rules and value > rules["max"]:
        errors.append(f"{field}: {value} > maximum {rules['max']}")

# -----
# IP Address Validation
# -----
if not validate_ip(row["src_ip"]):
    errors.append("src_ip: invalid IPv4/IPv6 address")

if not validate_ip(row["dst_ip"]):
    errors.append("dst_ip: invalid IPv4/IPv6 address")

if row["src_ip"] == row["dst_ip"]:
    errors.append("src_ip and dst_ip must not be identical")

# SRC IP checks
if validate_ip(row["src_ip"]):
    ip_obj = ipaddress.ip_address(row["src_ip"])
    if ip_obj.is_multicast:
        errors.append("src_ip is a multicast address")
    if ip_obj.is_loopback:
        errors.append("src_ip is a loopback address")
    if isinstance(ip_obj, ipaddress.IPv4Address) and ip_obj ==
ipaddress.IPv4Address("255.255.255.255"):
        errors.append("src_ip is a broadcast address")

# DST IP checks
if validate_ip(row["dst_ip"]):
    ip_obj = ipaddress.ip_address(row["dst_ip"])
    if ip_obj.is_multicast:
        errors.append("dst_ip is a multicast address")
    if ip_obj.is_loopback:
        errors.append("dst_ip is a loopback address")
    if isinstance(ip_obj, ipaddress.IPv4Address) and ip_obj ==
ipaddress.IPv4Address("255.255.255.255"):
        errors.append("dst_ip is a broadcast address")

# -----
# Protocol & Port Validation
# -----
protocol = row["protocol"]
src_port = row["src_port"]

```

```

dst_port = row["dst_port"]

if protocol not in IANA_PROTOCOLS:
    errors.append(f"Unknown protocol number: {protocol}")

if row["protocol_name"].upper() != IANA_PROTOCOLS.get(protocol,
 "").upper():
    errors.append(f"Protocol mismatch: protocol={protocol} but
protocol_name={row['protocol_name']}")

if protocol in (6, 17) and (src_port is None or dst_port is None):
    errors.append("TCP/UDP flows must include src_port and dst_port")

if protocol == 1 and (src_port != 0 or dst_port != 0):
    errors.append("ICMP should not use ports (should be 0)")

# -----
# Packet & Byte Validation
# -----
if row["packet_count"] == 0 and row["byte_count"] != 0:
    errors.append("packet_count=0 but byte_count > 0")

if row["byte_count"] < row["packet_count"]:
    errors.append("byte_count < packet_count (impossible)")

if row["packet_count"] > 0:
    expected_avg = row["byte_count"] / row["packet_count"]
    if abs(row["avg_packet_size"] - expected_avg) > 5:
        errors.append(
            f"avg_packet_size inconsistent (expected ~{expected_avg:.2f}, got
{row['avg_packet_size']})"
        )

if not (20 <= row["avg_packet_size"] <= 1500):
    errors.append("avg_packet_size outside typical Ethernet range (20–1500
bytes)")

# -----
# Packet Index Validation
# -----
if row["last_packet_index"] < row["first_packet_index"]:
    errors.append("last_packet_index < first_packet_index")

# -----
# Duration Validation
# -----
if row["duration"] < 0:

```

```

    errors.append("duration < 0")

    if row["packet_count"] == 1 and row["duration"] > 0.001:
        errors.append("duration > 0 for packet_count=1")

    if row["packet_count"] > 1 and row["duration"] == 0:
        errors.append("duration=0 but multiple packets exist")

    if row["duration"] > 3600:
        errors.append("flow duration unusually long (>1 hour)")

# -----
# Protocol-Port Sanity Checks
# -----
if row["src_port"] == 53 or row["dst_port"] == 53:
    if row["duration"] > 10:
        errors.append("DNS traffic duration unusually long")

if row["src_port"] == 443 or row["dst_port"] == 443:
    if row["packet_count"] < 2:
        errors.append("HTTPS flow has suspiciously low packet count")

return errors

=====
=====

Dataset-level validation
=====

=====

def validate_dataset(df, make_csv, pcap_basename):
    all_errors = {}

    for field in FIELD_SCHEMA.keys():
        if field not in df.columns:
            raise ValueError(f"Missing required column: {field}")

    duplicate_keys =
    ["src_ip","dst_ip","src_port","dst_port","protocol","first_packet_index"]
    dupes = df[df.duplicated(duplicate_keys, keep=False)]

    for idx in dupes.index:
        all_errors[idx] = ["Duplicate flow detected"]

    for idx, row in df.iterrows():
        row_errors = validate_row(row)
        if row_errors:
            all_errors.setdefault(idx, []).extend(row_errors)


```

```

df["is_valid"] = True
df["error_reason"] = ""

for idx, errs in all_errors.items():
    df.at[idx, "is_valid"] = False
    df.at[idx, "error_reason"] = "; ".join(errs)

print("Validated dataset")

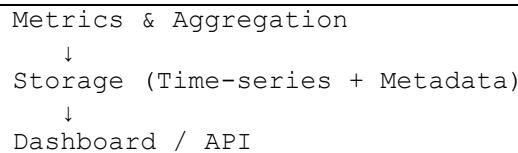
# =====
# EXPORT AS CSVB
# =====
if make_csv:
    output_path = "csv_output/" + pcap_basename + "_validated.csvb"
    df.to_csv(output_path, index=False)
    print(f"Validated dataset saved as CSVB → {output_path}")

return df

```

Sophia

Context:	Tech Stack for Network Traffic Profiler Project
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input checked="" type="checkbox"/>
Prompt:	We are developing a network traffic profiler as part of a larger system - an AI driven IDS firewall. Our system should take a PCAP file as input, run it through an AI model and provide extracted data and anomaly stats in a dashboard. What tech stack do you recommend for this project?
Response:	<p>That's a solid and very realistic project scope. I'll assume you want something production-grade, modular, and scalable (not just a research prototype), with PCAP → AI inference → analytics → dashboard.</p> <p>I'll break this down by architecture layer, then give two concrete stack options (Python-centric and mixed stack), and finish with design tips specific to IDS/traffic profiling.</p> <hr/> <h3>1. High-Level Architecture (Recommended)</h3> <pre> PCAP Ingestion ↓ Packet Parsing & Flow Extraction ↓ Feature Engineering ↓ AI Inference (Anomaly / Classification) ↓ </pre>



You want **clear separation** between:

- traffic parsing
- ML inference
- analytics / visualization

2. Core Tech Stack (Recommended)

◊ Packet Processing & Feature Extraction

Language: Python (dominant ecosystem here)

Libraries

- Scapy – flexible packet parsing (good for research & prototyping)
- PyShark – wrapper over tshark (stable, slower but accurate)
- dpkt – faster, lower-level parsing
- Zeek (optional but powerful) – for production-grade flow extraction

👉 Recommendation

- Use **Zeek** for flow-level features if possible
- Fall back to **Scapy/dpkt** if you want full control

Why Zeek?

- Battle-tested in IDS environments
- Produces rich logs (conn, dns, http, ssl, etc.)
- Easy to pipe into ML pipelines

◊ Feature Engineering & Profiling

Python stack

- pandas – feature aggregation
- numpy – numerical ops
- scikit-learn – normalization, PCA, baseline models
- tsfresh – time-series feature extraction (very useful)

	<p>Typical features</p> <ul style="list-style-type: none"> • Flow duration • Packet size statistics • Inter-arrival times • Protocol distributions • Entropy of ports/IPs • DNS/HTTP behavioral features
	<p>◊ AI / ML Layer</p> <p>Choose based on your goals:</p> <p><i>Option A: Classical + Unsupervised (Most IDS-friendly)</i></p> <ul style="list-style-type: none"> • Isolation Forest • LOF • One-Class SVM • Autoencoders <p><i>Option B: Deep Learning (Higher complexity)</i></p> <ul style="list-style-type: none"> • PyTorch or TensorFlow • LSTM / GRU (temporal modeling) • Transformer-based flow models • Graph neural networks (for host-level behavior) <p>👉 Recommendation</p> <ul style="list-style-type: none"> • Start with Isolation Forest + Autoencoder • Add deep models later for behavioral modeling
	<p>◊ Backend / API Layer</p> <p>Framework</p> <ul style="list-style-type: none"> • FastAPI (highly recommended) <ul style="list-style-type: none"> ◦ Async ◦ Auto-generated OpenAPI docs ◦ Easy integration with ML models <p>Responsibilities</p> <ul style="list-style-type: none"> • Upload PCAP • Trigger analysis jobs • Serve anomaly stats

	<ul style="list-style-type: none"> • Serve dashboard data <hr/> <p>◊ Data Storage</p> <p>You will need multiple storage types:</p> <ol style="list-style-type: none"> <i>1. Metadata / Results</i> <ul style="list-style-type: none"> • PostgreSQL <ul style="list-style-type: none"> ◦ Analysis runs ◦ Model outputs ◦ Alerts <i>2. Time-Series Metrics</i> <ul style="list-style-type: none"> • TimescaleDB (Postgres extension) <ul style="list-style-type: none"> ◦ Traffic rates ◦ Anomaly scores over time <i>3. Optional Search / Correlation</i> <ul style="list-style-type: none"> • OpenSearch / Elasticsearch <ul style="list-style-type: none"> ◦ Query traffic patterns ◦ Correlate alerts ◦ SIEM-style usage <hr/> <p>◊ Dashboard / Visualization</p> <p><i>Option A: Web Dashboard (Best UX)</i></p> <ul style="list-style-type: none"> • Frontend <ul style="list-style-type: none"> ◦ React + TypeScript ◦ Chart.js / Recharts / D3.js • Backend <ul style="list-style-type: none"> ◦ FastAPI <p><i>Option B: Security-friendly Stack</i></p> <ul style="list-style-type: none"> • Grafana <ul style="list-style-type: none"> ◦ Connects directly to TimescaleDB ◦ Minimal frontend work • Kibana (if using OpenSearch) <p>👉 Recommendation</p> <ul style="list-style-type: none"> • Grafana for early versions
--	---

	<ul style="list-style-type: none"> • React dashboard for polished product <hr/> <p>◊ Job Processing / Scalability</p> <p>PCAP processing is CPU-heavy.</p> <p>Use</p> <ul style="list-style-type: none"> • Celery or RQ • Redis or RabbitMQ <p>For large PCAPs:</p> <ul style="list-style-type: none"> • Chunk PCAPs • Parallelize flow extraction • Async inference 						
	<h3>3. Deployment & Ops</h3> <p>◊ Containerization</p> <ul style="list-style-type: none"> • Docker (mandatory) • Docker Compose (dev) • Kubernetes (prod / scaling) <p>◊ Model Versioning</p> <ul style="list-style-type: none"> • MLflow • Store: <ul style="list-style-type: none"> ◦ Model versions ◦ Feature schemas ◦ Metrics <hr/> <h3>4. Example “Golden Stack” (Strong Recommendation)</h3> <p> IDS-Optimized Stack</p> <table> <thead> <tr> <th>Layer</th> <th>Technology</th> </tr> </thead> <tbody> <tr> <td>PCAP Parsing</td> <td>Zeek + Scapy</td> </tr> <tr> <td>Feature Engineering</td> <td>Pandas, NumPy, tsfresh</td> </tr> </tbody> </table>	Layer	Technology	PCAP Parsing	Zeek + Scapy	Feature Engineering	Pandas, NumPy, tsfresh
Layer	Technology						
PCAP Parsing	Zeek + Scapy						
Feature Engineering	Pandas, NumPy, tsfresh						

ML	PyTorch + scikit-learn
API	FastAPI
Jobs	Celery + Redis
DB	PostgreSQL + TimescaleDB
Search	OpenSearch
Dashboard	Grafana + React
Deployment	Docker + Kubernetes

5. Design Tips Specific to AI-Driven IDS

Avoid Common Mistakes

-  Don't feed raw packets directly into ML
-  Don't treat anomaly score as alert without context
-  Don't ignore temporal correlation

Best Practices

- Profile **flows**, not packets
- Add **baseline learning per host/network**
- Combine:
 - statistical anomalies
 - behavioral anomalies
- Store **raw features** for retraining
- Support **offline PCAP + live capture later**

6. Roadmap Suggestion

Phase 1

- PCAP → flows → anomaly score
- Grafana dashboard

Phase 2

- Host behavior modeling
- Alert classification

	<p>Phase 3</p> <ul style="list-style-type: none"> • Online traffic capture • Feedback loop (human-in-the-loop) <hr/> <p>If you want, I can:</p> <ul style="list-style-type: none"> • Design a concrete microservice diagram • Propose a feature schema for flows • Suggest AI models per traffic type • Compare Zeek vs custom parsing <p>Just tell me how deep you want to go.</p>
Implemented:	Python as main development language with accompanying libraries, start with IsolationForest for ML and eventually progress onto deep learning with PyTorch/Tensorflow

Context:	Folder structure set up
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input checked="" type="checkbox"/>
Prompt:	We are developing a network traffic profiler, which contains of this pipeline: Take in PCAP file -> extract stats -> validate stats -> clean up and prepare for ML -> train/run through model -> display data on dashboard Using python as main development language, what folder structure do you recommend?
Response:	<p>For a network traffic profiler with a clear, multi-stage pipeline and Python as the primary language, I'd recommend a pipeline-oriented, modular structure that cleanly separates data ingestion, processing, ML, and presentation, while still being practical for iteration.</p> <p>Below is a production-friendly structure that works well for ML systems and data pipelines.</p> <hr/> <h2>High-Level Principles</h2> <ul style="list-style-type: none"> • One responsibility per module • Pipeline stages map to folders • Clear separation of <ul style="list-style-type: none"> ◦ core logic ◦ experiments ◦ configuration ◦ interfaces (CLI / API / dashboard) • Easy to test and scale

Recommended Folder Structure

```
network_traffic_profiler/
    ├── README.md
    ├── pyproject.toml / requirements.txt
    ├── .env
    └── .gitignore

    ├── data/
        ├── raw/                                # Original PCAP files
        ├── extracted/                           # Parsed stats (CSV / Parquet /
JSON)
        ├── validated/                           # Post-validation datasets
        ├── processed/                           # Cleaned & ML-ready data
        ├── features/                            # Feature matrices for models
        └── models/                             # Serialized trained models

    └── src/
        └── ntprofiler/
            ├── __init__.py
            ├── ingestion/
            │   ├── __init__.py
            │   ├── pcap_reader.py
            │   └── flow_builder.py
            ├── extraction/
            │   ├── __init__.py
            │   ├── stats_extractor.py
            │   └── protocol_features.py
            ├── validation/
            │   ├── __init__.py
            │   ├── schema.py
            │   └── validators.py
            ├── preprocessing/
            │   ├── __init__.py
            │   ├── cleaning.py
            │   ├── normalization.py
            │   └── feature_engineering.py
            └── ml/
                ├── __init__.py
                ├── datasets.py
                ├── train.py
                ├── inference.py
                └── models.py

            └── pipeline/
                ├── __init__.py
                ├── run_pipeline.py
                └── stages.py

            └── dashboard/
                ├── __init__.py
                ├── app.py                                # Streamlit / Dash / FastAPI
                └── visualizations.py
```

```
utils/
└── __init__.py
└── logging.py
└── io.py
└── timing.py

config/
└── __init__.py
└── settings.py
└── schemas.yaml

scripts/
└── ingest_pcap.py
└── train_model.py
└── run_dashboard.py

tests/
└── ingestion/
└── extraction/
└── validation/
└── preprocessing/
└── ml/
└── pipeline/

notebooks/
└── exploration.ipynb
└── feature_analysis.ipynb
```

Mapping to Your Pipeline

Pipeline Step	Folder
Take in PCAP	ingestion/
Extract stats	extraction/
Validate stats	validation/
Clean & prepare	preprocessing/
Train / run ML	ml/
Orchestration	pipeline/
Display dashboard	dashboard/

Key Design Choices (Why This Works)

1. Pipeline folder

Encapsulates the *order* and *control flow* without mixing logic:

```
# pipeline/run_pipeline.py
from ntprofiler.ingestion import pcap_reader
from ntprofiler.extraction import stats_extractor
from ntprofiler.validation import validators
from ntprofiler.preprocessing import cleaning
from ntprofiler.ml import inference
```

This keeps individual stages reusable.

2. Data versioning ready

The `data/` layout mirrors ML best practices (DVC-friendly).

3. Dashboard isolated

Your dashboard can evolve independently (Streamlit today, React + API later).

4. Config-driven

Centralized config allows:

- protocol toggles
- feature flags
- model selection
- thresholds for validation

Optional Enhancements (As You Scale)

If this becomes production-grade:

- Add `orchestration/` (Airflow / Prefect)
- Add `api/` (FastAPI for inference)
- Add `monitoring/` (data drift, model metrics)

If real-time traffic later:

	<ul style="list-style-type: none"> • streaming/ (Kafka, Zeek, Suricata inputs) <hr/> <p>Minimal Version (If You Want Lean)</p> <p>If this is an early MVP, I can also propose a pared-down version that still scales cleanly.</p> <p>Would you like:</p> <ul style="list-style-type: none"> • <input checked="" type="checkbox"/> MVP-lean structure • <input checked="" type="checkbox"/> Production-ready enterprise structure • <input checked="" type="checkbox"/> ML-research-oriented structure <p>Just tell me which direction you're aiming for.</p>
Implemented:	Split up pipeline into different folders, Streamlit for dashboard, /tests folder for testing, /data folder for different data files

Context:	Match compatible versions for the requirements.txt from our tech stack
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input checked="" type="checkbox"/>
Prompt:	We are developing a network traffic profiler project, and these are the Python libraries we want to use. What is the best Python version to use? Also, help me match compatible versions for the requirements.txt: Pandas, scapy, numpy, streamlit, plotly, scikit-learn, pytest.
Response:	<p>For a network traffic profiler using the libraries you listed, the best Python version to use is:</p> <p><input checked="" type="checkbox"/> Python 3.11</p> <p>Why Python 3.11?</p> <ul style="list-style-type: none"> • Fully compatible with all the libraries you mentioned • Noticeably faster execution than 3.9 / 3.10 (great for packet processing & data analysis) • Very stable and widely supported in 2025 • Avoids edge-case dependency issues that still appear with Python 3.12+ in some scientific and networking stacks <p>Python 3.10 is also acceptable, but 3.11 is the sweet spot for performance + compatibility right now.</p>

Recommended `requirements.txt` (Compatible & Stable)

The following versions are **known to work together cleanly on Python 3.11** and are well-suited for a traffic profiling / analytics project:

```
numpy==1.26.4
pandas==2.3.3
scapy==2.6.1
streamlit==1.33.0
plotly==5.22.0
scikit-learn==1.4.2
pytest==8.1.1
```

Compatibility Rationale (Short & Practical)

Library	Reason
numpy 1.26.4	Last NumPy before major ABI changes; safest for scientific stack
pandas 2.3.3	Fully compatible with NumPy 1.26.x and Python 3.11
scapy 2.6.1	Stable packet parsing & sniffing on Python 3.11
streamlit 1.33.0	Works cleanly with Plotly + Pandas 2.x
plotly 5.22.0	No rendering issues with Streamlit
scikit-learn 1.4.2	Matches NumPy 1.26 ABI, stable ML APIs
pytest 8.1.1	Fully supports Python 3.11

Recommended Setup Command

```
python3.11 -m venv venv
source venv/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
```

Versions to Avoid

- **Python 3.12** → still causes build issues for some native extensions
- **NumPy ≥ 2.0** → breaks ABI compatibility with scikit-learn & others
- **Older pandas (<2.0)** → weaker performance and feature set

	<p>If you want, I can also:</p> <ul style="list-style-type: none"> • Generate a <code>pyproject.toml</code> • Recommend OS-level dependencies for Scapy (libpcap, npcap) • Suggest a Dockerfile for reproducible builds • Validate this stack for Linux vs Windows packet capture <p>Just tell me </p>
Implemented:	Python and library version recommendations

Context:	Improvements to code
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input checked="" type="checkbox"/>
Prompt:	Please provide feedback and improvements to my data set clean up code. [pasted in code]
Response:	Below is structured, practical feedback on your dataset clean-up code, followed by a cleaned-up, improved version that keeps your intent but makes it more robust, ML-correct, and production-ready.
<h2>High-Level Assessment</h2> <p>Overall quality: good </p> <p>You're doing many things <i>correctly</i> that beginners often miss:</p> <ul style="list-style-type: none"> • Filtering invalid rows early • Dropping non-ML columns • Enforcing numeric-only features • Removing zero-variance columns • Scaling features • Doing a sanity-check anomaly preview <p>This is already a solid ML preprocessing pipeline.</p>	

Key Issues & Improvements (Most Important First)

1. ! IsolationForest Preview Should NOT Train on All Data Blindly

Right now:

```
model = IsolationForest().fit(scaled)
preds = model.predict(scaled)
```

Issue:

- You are fitting and evaluating on the same data.
- This is OK for a *preview*, but you should **make that explicit** and tune parameters.
- Default contamination may give misleading anomaly percentages.

Improvement:

- Set `contamination`
- Set `random_state`
- Clarify it's a *preview only*

```
model = IsolationForest(
    n_estimators=100,
    contamination="auto",
    random_state=42
)
```

2. ! Scaling Should Handle NaNs Explicitly

StandardScaler **does not handle NaNs well**.

If any numeric column contains NaNs, this will silently break downstream logic.

Fix:

- Check or impute before scaling

```
if numeric_df.isna().any().any():
    print("WARNING: NaNs detected - filling with column median")
    numeric_df = numeric_df.fillna(numeric_df.median())
```

3. ⚠ Zero-Variance Detection Should Use a Tolerance

Floating-point values rarely produce *exact* zero variance.

Current code:

```
zero_var = stds[stds == 0]
```

Better:

```
zero_var = stds[stds < 1e-12]
```

4. ⚠ Boolean Columns Are Numeric (and May Hurt the Model)

`select_dtypes(include=[np.number])` includes:

- `bool`
- encoded flags

These may or may not be useful.

Suggestion:

Explicitly exclude booleans unless intentional.

```
numeric_df = df.select_dtypes(include=[np.number]).copy()  
numeric_df = numeric_df.loc[:, numeric_df.dtypes != bool]
```

5. ⚠ You Mutate `numeric_df` After Scaling

You add:

```
numeric_df["anomaly"] = (preds == -1)
```

But `numeric_df` is **unscaled**, while `preds` are from `scaled`.

That's fine for a dashboard, but it should be **explicitly named** to avoid confusion.

Rename for clarity:

- `features_df`
- `dashboard_df`

6. ✓ Logging > `print()`

For ML pipelines, logging is preferred.

At minimum:

```
import logging
logging.basicConfig(level=logging.INFO)
```

Then replace `print()` with `logging.info() / warning()`.

7. ⚠ Hard-coded Thresholds (`len(df) < 5`)

This is fine for now, but should be a named constant.

```
MIN_REQUIRED_FLOWS = 5
```

Improved & Refactored Version (Recommended)

This version:

- Keeps your logic
- Improves safety, clarity, and ML hygiene
- Is still **easy to understand**

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import IsolationForest

MIN_REQUIRED_FLOWS = 5
ZERO_VAR_TOL = 1e-12

def build_dataset(df: pd.DataFrame, pcap_basename: str):
    print("Building ML dataset")

    # 1. Keep only valid rows
    df = df[df["is_valid"] is True].copy()
    print(f"Valid rows: {len(df)}")

    if len(df) < MIN_REQUIRED_FLOWS:
        print("WARNING: Very few valid flows - ML may be unreliable")

    # 2. Drop non-ML columns
    drop_cols = [
        "src_ip",
        "dst_ip",
        "protocol_name",
        "error_reason",
        "is_valid",
    ]
    df = df.drop(columns=drop_cols, errors="ignore")

    # 3. Select numeric (non-boolean) features
    numeric_df = df.select_dtypes(include=[np.number]).copy()
```

```

        numeric_df = numeric_df.loc[:, numeric_df.dtypes != bool]

        print(f"Numeric features ({numeric_df.shape[1]}):")
        print(list(numeric_df.columns))

        # 4. Handle NaNs
        if numeric_df.isna().any().any():
            print("WARNING: NaNs detected - filling with column
medians")
            numeric_df = numeric_df.fillna(numeric_df.median())

        # 5. Remove zero-variance features
        stds = numeric_df.std()
        zero_var_cols = stds[stds < ZERO_VAR_TOL].index

        if len(zero_var_cols) > 0:
            print("Removing zero-variance features:")
            print(list(zero_var_cols))
            numeric_df = numeric_df.drop(columns=zero_var_cols)

        # 6. Scale features
        scaler = StandardScaler()
        scaled = scaler.fit_transform(numeric_df)

        print(f"Scaled dataset shape: {scaled.shape}")

        # 7. Anomaly preview (NOT final model)
        preview_model = IsolationForest(
            n_estimators=100,
            contamination="auto",
            random_state=42,
        )
        preds = preview_model.fit_predict(scaled)

        anomalies = np.sum(preds == -1)
        total = len(preds)

        print(f"Anomaly preview: {anomalies}/{total} flows
flagged")

        if anomalies == 0:
            print("No anomalies detected - dataset may be too
uniform")
        elif anomalies > total * 0.5:
            print("High anomaly rate - dataset may be unusual")
        else:
            print("Anomaly level within expected range")

        # 8. Attach anomaly labels for dashboard
        dashboard_df = numeric_df.copy()
        dashboard_df["anomaly"] = (preds == -1)

        anomaly_info = {
            "anomaly_count": int(anomalies),
            "total_flows": int(total),
            "anomaly_percentage": float(anomalies / total * 100),
        }

        print("Dataset ready for ML")

        return dashboard_df, anomaly_info
    
```

	<p>Final Verdict</p> <p>Score: 8.5 / 10</p> <p>You are:</p> <ul style="list-style-type: none"> • Thinking like an ML engineer ✓ • Cleaning data properly ✓ • Adding sanity checks ✓ <p>To reach professional-grade:</p> <ul style="list-style-type: none"> • Separate preview logic from training logic • Add logging • Persist scaler + feature list for inference • Formalize thresholds & configuration <p>If you want, I can next:</p> <ul style="list-style-type: none"> • Split this into train / inference pipelines • Add unit tests (pytest) • Add feature persistence for Streamlit deployment
Implemented:	Added parameters to IsolationForest, added NaN handler, added tolerance to zero-variance detection

Tomiris

Context:	Sub data packet output for flows
Team Member:	Tim: <input type="checkbox"/> Tomiris: ☒ Amelia: <input type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	Right now I'm using a dropdown to show the sub-packet details for each flow, but I'm not fully satisfied with it. What other UI components or approaches could I use to display this information more clearly or efficiently?
Response:	<p>Alternative ways to show packet-level details without a dropdown</p> <p>1. Expandable table rows (accordion style)</p> <p>Clicking a flow row expands a packet table directly underneath.</p> <p>Flow Row ↳ Sub-table showing packets</p>

Pros: clean, intuitive
Cons: can get long if many flows expand at once

2. Side panel / drawer

Click a flow → panel slides from the right with packet list.

Pros: modern UI, doesn't disrupt main table
Cons: requires some extra UI work

3. Modal popup

Click a flow → modal opens with packet details.

Pros: simple to implement
Cons: blocks the screen

4. Separate “Packets” section

Click a flow → lower section updates and displays packets.

Pros: clean, no popups
Cons: user scrolls more

5. New page / tab

Select a flow → navigate to new view with packet details.

Pros: handles large data
Cons: not SPA-like unless designed well

Context:	Filtering section
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input checked="" type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	At the moment, my filter only works for protocols. How can I extend the filtering system so that users can also filter flows by source/destination

	IP and by port numbers? I'm not sure how to structure the UI for multiple filters or how to validate these inputs.
Response:	<p>Add Filters for:</p> <ul style="list-style-type: none"> • Source IP • Destination IP • Source Port • Destination Port <p>All added below the protocol filter, and they work together.</p> <p>Updated Filter Section</p> <pre># 2. Filtering section # Allows filtering by protocol, IP address, and port numbers # ----- -----</pre> <p>st.sidebar.header("Filters")</p> <p># Filter by protocol</p> <pre>protocols = st.sidebar.multiselect("Protocol", df["protocol_name"].unique())</pre> <p># Filter by source IP</p> <pre>src_ips = st.sidebar.multiselect("Source IP", df["src_ip"].unique())</pre> <p># Filter by destination IP</p> <pre>dst_ips = st.sidebar.multiselect("Destination IP", df["dst_ip"].unique())</pre> <p># Filter by source port</p> <pre>src_ports = st.sidebar.multiselect(</pre>

```
        "Source Port",
        sorted(df["src_port"].dropna().unique())
    )

# Filter by destination port
dst_ports = st.sidebar.multiselect(
    "Destination Port",
    sorted(df["dst_port"].dropna().unique())
)

# Start with full dataset
filtered = df.copy()

# Apply filters conditionally
if protocols:
    filtered =
filtered[filtered["protocol_name"].isin(protocols)]

if src_ips:
    filtered =
filtered[filtered["src_ip"].isin(src_ips)]

if dst_ips:
    filtered =
filtered[filtered["dst_ip"].isin(dst_ips)]

if src_ports:
    filtered =
filtered[filtered["src_port"].isin(src_ports)]

if dst_ports:
    filtered =
filtered[filtered["dst_port"].isin(dst_ports)]
```

What this adds

Your dashboard now supports:

	<ul style="list-style-type: none"> ✓ Filtering like a real network analysis tool ✓ Multi-filtering (combine protocol + IP + ports) ✓ Sidebar layout (cleaner UI) ✓ Works with the rest of your charts without breaking anything
--	---

Context:	Unit tests
Team Member:	Tim: <input type="checkbox"/> Tomiris: <input checked="" type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	For unit testing the extract_features function, is it more appropriate to use real PCAP files as test inputs or to rely on mock data? What are the advantages and limitations of each approach?
Response:	<p>Option 1 — You can test with real PCAP files (integration tests)</p> <p>If your project includes a script that:</p> <ol style="list-style-type: none"> 1. Reads a PCAP file 2. Extracts flow/session data 3. Outputs a CSV (flows.csv, test_flows_validated.csv, etc.) <p>...then you <i>can absolutely</i> write a test that feeds a real PCAP file and checks the correctness of the output.</p> <p>⚠ BUT PCAP tests are considered <i>integration tests</i>, not unit tests. So keep them separately in a folder like:</p> <pre>tests/integration/test_real_pcap.py tests/data/example1.pcap tests/data/example2.pcap</pre>

✓ Example of an integration test using a real PCAP file

```
import os
import pandas as pd
from process_dataset import process_pcap # example
function name

def test_real_pcap_processing(tmp_path):
    pcap_path = "tests/data/example1.pcap"

        # Output CSV path
    output_csv = tmp_path / "output.csv"

        # Run your actual pipeline
    process_pcap(pcap_path, output_csv)

        # Check file was created
    assert output_csv.exists()

        # Load and check dataset looks reasonable
    df = pd.read_csv(output_csv)

        assert len(df) > 0                      # Should
extract at least 1 flow
        assert "src_ip" in df.columns
        assert "dst_ip" in df.columns
        assert "protocol_name" in df.columns
```

This test ensures:

- the PCAP file can be parsed
- your pipeline generates output
- the output format matches what your dashboard expects

Option 2 — Unit tests (what you are doing now) DO NOT need real PCAPs

Unit tests only test **your internal functions**, like:

- `validate_ip()`
- `validate_row()`
- duplicate detection
- dataframe cleaning
- etc.

These should be tested using **small, synthetic, in-memory test data**, like you already have.

Why not use PCAPs for unit tests?

- They are large
- They make tests slow
- PCAP structure is not stable
- Unit tests must be fast, isolated, repeatable

So the correct approach:

Test Type	Uses Real PCAPs?	Purpose
Unit tests	✗ No	Test small functions only

	Integration tests	✓ Yes	Test entire pipeline end-to-end with real PCAP files
	Dashboard tests	✗ Probably not	Dashboard works on CSVs, not PCAPs

Context:	Top conversations table
Team Member:	Tim: <input checked="" type="checkbox"/> Tomiris: <input type="checkbox"/> Amelia: <input type="checkbox"/> Sophia: <input type="checkbox"/>
Prompt:	Given a Pandas dataframe 'df' that contains a collection of network flows, produce code to create a table of top conversations that could be displayed in Streamlit.

Response:	<p>Below is a clean, Streamlit-ready example that takes a flow-level Pandas DataFrame (df) like yours and produces a “Top Conversations” table.</p> <p>This assumes your dataframe looks similar to what you’ve been generating:</p> <p>src_ip</p> <p>dst_ip</p> <p>protocol_name (or protocol)</p> <p>packet_count</p> <p>byte_count</p> <p>duration</p> <p> What “Top Conversations” means here</p> <p>A conversation = communication between two IPs, regardless of direction.</p> <p>So:</p> <p>$A \rightarrow B + B \rightarrow A = \text{one conversation}$</p> <p>We’ll:</p> <p>Normalize IP pairs (so order doesn’t matter)</p> <p>Aggregate traffic stats</p> <p>Sort by total bytes</p> <p>Display in Streamlit</p> <p>  Create conversation keys (direction-agnostic)</p>
-----------	--

```
import pandas as pd

df["conversation"] = df.apply(
    lambda r: tuple(sorted([r["src_ip"], r["dst_ip"]])), 
    axis=1
)
```

This ensures:

10.0.0.1 ↔ 8.8.8.8

is always represented the same way.

 **2** Aggregate conversations

```
top_conversations = (
    df.groupby("conversation")
    .agg(
        total_bytes=("byte_count", "sum"),
        total_packets=("packet_count", "sum"),
        avg_packet_size=("avg_packet_size", "mean"),
        flow_count=("conversation", "count"),
    )
    .sort_values(by="total_bytes", ascending=False)
    .reset_index()
)
```

You can also rank by:

total_packets

total_flows

avg_flow_duration

 **4** Display in Streamlit

```
import streamlit as st
```

```
st.subheader("Top Network Conversations")
```

```
st.dataframe(  
    top_conversations,  
    use_container_width=True  
)
```