# Phase 5: Real-Time Detection Framework Architecture on Linux

## 1. Strategic Architecture Overview

The transition of the AICyberDefense project from offline historical analysis (Phases 1-3) to a live operational capability (Phase 5) represents a fundamental shift in architectural requirements. While earlier phases focused on the statistical purity of data and the theoretical efficacy of deep learning models such as the Autoencoder and Isolation Forest, Phase 5 demands a rigorous engineering approach centered on latency minimization, throughput maximization, and fault tolerance. The objective is to deploy a **Real-Time Detection Framework** on a Linux host that ingests raw network traffic and endpoint telemetry, transforms this data into feature vectors compatible with the pre-trained models, and renders verdicts at line speed.

This report provides an exhaustive analysis of the necessary components, leveraging deep research into kernel-space packet processing, user-space feature engineering, and concurrent daemon design. The architecture is predicated on a "Hybrid Multiprocessing" model, necessitated by the specific performance characteristics of Python's Global Interpreter Lock (GIL) and the disparate nature of network (continuous, high-velocity) vs. endpoint (event-driven, bursty) data streams.

The system is designed to operate within the Linux userspace but relies heavily on kernel primitives—specifically AF\_PACKET for network capture and udev/evdev for hardware monitoring—to bypass the latency penalties associated with traditional interpreted middleware. Furthermore, the integration of **NFStream** as the primary flow aggregation engine marks a departure from the legacy CICFlowMeter tool used in dataset generation, requiring a meticulous feature parity analysis to ensure that the live data manifold aligns perfectly with the training distribution derived from CICIDS2017.1

The following sections detail the theoretical underpinnings and implementation strategies for each subsystem, supported by empirical benchmarks and architectural best practices identified in the research corpus.

## 2. High-Performance Network Ingestion Architecture

The ingestion layer serves as the sensory cortex of the detection framework. Its primary mandate is to capture raw Ethernet frames from the Network Interface Card (NIC) and reconstruct them into stateful flows (sessions) without inducing packet loss (drops). In a Gigabit Ethernet environment, this requires processing budgets in the microsecond range per packet.

### 2.1 Kernel-Space Capture Mechanisms: The AF\_PACKET Paradigm

Standard network programming often utilizes the Berkeley Socket API (SOCK\_STREAM or SOCK\_DGRAM) or high-level libraries like libpcap (used by Wireshark/TCPDump). However, for an Intrusion Detection System (IDS) operating on high-throughput links, these abstractions introduce unacceptable overhead due to the frequency of system calls and the cost of copying data between kernel-space and user-space memory.

Research indicates that the optimal mechanism for Python-based packet capture on Linux is the **AF\_PACKET** socket interface, specifically utilizing version 3 of the packet ring buffer (TPACKET\_V3).2

#### 2.1.1 Memory-Mapped I/O (PACKET\_MMAP)

The bottleneck in traditional capture is the recvmsg() system call, which requires a context switch for every packet. AF\_PACKET addresses this via PACKET\_MMAP, a mechanism that creates a shared circular ring buffer mapped into both kernel and user memory spaces. The kernel Direct Memory Access (DMA) engine writes packet data directly into this ring, and the user-space application polls the buffer status bits to consume data. This "zero-copy" approach (from the perspective of the kernel-to-user boundary) significantly reduces CPU cycles spent on interrupt handling and memory marshalling.3

#### 2.1.2 Parallelism via PACKET\_FANOUT

A critical limitation of single-process capture is CPU core saturation. Modern Linux kernels (3.1+) implement PACKET\_FANOUT, a load-balancing mechanism that distributes incoming packets across multiple sockets belonging to the same "fanout group."

Crucially for NIDS, this distribution uses a hash of the flow 5-tuple (Source IP, Destination IP, Source Port, Destination Port, Protocol). This ensures **flow affinity**: all packets belonging to a specific TCP session are guaranteed to arrive at the same worker process. This eliminates the need for complex inter-process locking or shared state tables for flow reassembly, allowing the architecture to scale linearly with the number of available CPU cores.5

### 2.2 Framework Selection: Comparative Analysis

The selection of the ingestion framework dictates the system's maximum throughput and development velocity. We analyzed three primary candidates: **Scapy**, **Pyshark**, and **NFStream**.

| **Feature** | **Scapy** | **Pyshark** | **NFStream** |
| --- | --- | --- | --- |
| **Underlying Mechanism** | Raw Sockets / Libpcap | TShark (Wireshark) Wrapper | C-Engine via CFFI (AF\_PACKET) |
| **Parsing Speed** | Extremely Slow (Python Objects) | Slow (XML/JSON Parsing) | Near-Native (C Structures) |
| **State Management** | Manual Implementation | Inherited from Wireshark | Built-in Flow Cache |
| **Concurrency** | Difficult (Not Thread Safe) | Process-based (Heavy) | Native Multi-threading |
| **Use Case** | Prototyping / Packet Crafting | Forensics / Deep Analysis | **High-Speed NIDS** |

**Analysis:**

* **Scapy:** While Scapy is the industry standard for packet manipulation and crafting, its performance as a sniffer is prohibitive for real-time NIDS. It instantiates a full Python object for every packet layer, resulting in massive memory overhead and garbage collection pauses that lead to packet drops at speeds as low as a few Mbps.4
* **Pyshark:** Pyshark spawns tshark subprocesses and parses their textual output. This serialization/deserialization penalty makes it unsuitable for live traffic inspection where millisecond latency is required.4
* **NFStream:** NFStream is architected specifically for this use case. It separates the "fast path" (packet parsing, hashing, and aggregation) into a C-based engine, while exposing the "slow path" (feature logic and export) to Python via the C Foreign Function Interface (CFFI). Benchmarks demonstrate that NFStream outperforms both Scapy and pure Libpcap wrappers, achieving performance metrics comparable to compiled C++ applications while maintaining Python's flexibility.2

**Decision:** The framework will utilize **NFStream** as the core ingestion engine. Its native support for AF\_PACKET fanout, N-tuple hashing, and efficient flow expiration management aligns perfectly with the project's performance requirements.

### 2.3 Optimization of Flow Expiration Timers

In offline PCAP processing, flows are aggregated until the file ends or a TCP FIN/RST is seen. In a real-time system, waiting for connection termination is dangerous; a long-duration DDoS attack or slow data exfiltration might remain undetected for hours.

We must configure the NFStreamer with aggressive timeout policies to force "snapshotting" of active flows:

* **Active Timeout (active\_timeout):** Set to **60 seconds**. This ensures that long-lived connections (e.g., SSH tunnels, VPNs) are sliced into minute-long feature vectors. The model will analyze these slices independently, allowing the detection of anomalies within ongoing sessions.9
* **Idle Timeout (idle\_timeout):** Set to **15 seconds**. Flows that have ceased transmission are flushed quickly to free up memory in the flow cache and reduce detection latency.9

These settings trade a slight increase in flow fragmentation for significantly improved responsiveness to active threats.

## 3. Real-Time Feature Engineering (Network)

The AICyberDefense detection logic (Phase 2B Autoencoder) relies on a specific feature set derived from the **CICIDS2017** dataset. This dataset was originally generated using **CICFlowMeter**, a Java-based tool. To utilize the trained models, the Linux framework must generate feature vectors that are statistically identical (feature parity) to CICFlowMeter's output.

### 3.1 The Feature Parity Challenge

CICFlowMeter extracts over 80 features, including statistical distributions of packet lengths, inter-arrival times (IAT), and flag counts. NFStream, out of the box, provides a subset of these features (approx. 40).

**Gap Analysis:**

* **Available in NFStream:** src\_ip, dst\_port, bidirectional\_packets, bidirectional\_bytes, duration\_ms.
* **Missing / Requiring Custom Logic:** Flow IAT Std, Flow IAT Max, Fwd Packet Length Variance, Bwd Packet Length Mean, Flow Bytes/s (calculated continuously), and detailed TCP flag counters (e.g., PSH Flag Count, URG Flag Count).5

To bridge this gap, we must implement a custom **NFPlugin**. This plugin system allows us to inject Python logic into the C-based flow processing loop, updating custom state variables for every packet processed.

### 3.2 Advanced Statistical Accumulation: Welford’s Algorithm

A naive implementation of standard deviation requires storing all data points (e.g., all packet sizes in a flow) and iterating over them twice (once for the mean, once for variance). This is $O(N)$ in memory and time, which is unacceptable for streaming data.

To maintain $O(1)$ memory usage (constant state per flow), we implement **Welford’s Online Algorithm** for calculating variance and standard deviation.13

**Mathematical Formulation:**

For a sequence of values $x\_1, x\_2, \dots, x\_n$ (e.g., Packet Inter-Arrival Times):

1. Initialize $count = 0$, $mean = 0$, $M2 = 0$.
2. For each new value $x$:
   * $count \leftarrow count + 1$
   * $delta \leftarrow x - mean$
   * $mean \leftarrow mean + \frac{delta}{count}$
   * $delta2 \leftarrow x - mean$
   * $M2 \leftarrow M2 + delta \times delta2$
3. At any point, Variance ($s^2$) = $\frac{M2}{count - 1}$ (sample variance).
4. Standard Deviation = $\sqrt{s^2}$.

This algorithm is numerically stable and allows us to compute Flow IAT Std and Packet Length Std incrementally without storing the history of every packet.15

### 3.3 NFPlugin Implementation Architecture

The custom plugin CICFeaturePlugin will hook into three lifecycle events provided by NFStream 5:

#### 3.3.1 Initialization (on\_init)

When a new flow is detected (first packet), we initialize the custom storage area (flow.udps). This includes:

* Welford accumulators for Packet Length (Bidirectional, Forward, Backward).
* Welford accumulators for Inter-Arrival Time (IAT).
* Counters for specific TCP flags (SYN, ACK, PSH, URG, FIN, RST).
* Timestamps for the last packet seen (to calculate deltas for IAT).

#### 3.3.2 Update Loop (on\_update)

For every subsequent packet in the flow:

1. **Directionality:** Determine if the packet is Forward (src -> dst) or Backward (dst -> src) using packet.direction.
2. **Timings:** Calculate delta\_time = packet.time - flow.udps.last\_time. Update the IAT Welford accumulator.
3. **Size:** Update the Packet Length Welford accumulator with packet.raw\_size.
4. **Flags:** Bitwise check packet.tcp\_flags and increment relevant counters (e.g., if flags & 0x02: syn\_count += 1).
5. **Window Management:** If calculating "Sub-flow" features (specific to CICIDS2017), maintain a sliding window count if necessary, although strict CIC parity usually aggregates over the full flow duration.

#### 3.3.3 Expiration & Export (on\_expire)

When the flow expires (idle or active timeout):

1. Finalize Welford calculations: Divide $M2$ by $(count-1)$ and take the square root to get Standard Deviation.
2. Compute Rates: Flow Bytes/s = total\_bytes / (duration\_ms / 1000.0).
3. Format the feature vector: Construct a dictionary or Numpy array mapping exactly to the 80 columns expected by the Phase 2B model (e.g., ensuring Fwd Packet Length Std is at index 14, etc.).17

### 3.4 Online Normalization Strategy (Z-Score)

The deep learning models (Autoencoders) are trained on standardized data (Z-Score Normalization).

$$z = \frac{x - \mu}{\sigma}$$

In a production environment, we cannot calculate $\mu$ (global mean) and $\sigma$ (global std dev) on the live stream because we do not have the future data.

**Frozen Scaler Approach:**

We must utilize the **Pre-fitted Scaler** saved during Phase 1. The mean\_ and scale\_ attributes from the training phase StandardScaler are loaded into memory at daemon startup.

* **Process:**
  1. Extract raw features via NFStream (e.g., Flow Duration = 500ms).
  2. Apply the transformation: $Feature\_{norm} = (Feature\_{raw} - \mu\_{train}) / \sigma\_{train}$.
  3. Pass $Feature\_{norm}$ to the inference engine.
* **Implication:** This enforces the definition of "normality" as seen during training. If live traffic patterns shift significantly (Concept Drift), the reconstruction error will rise, properly signaling an anomaly (or the need for retraining).18

## 4. Endpoint Telemetry: USB Monitoring Subsystem

While NIDS detects network-level threats, it is blind to physical access attacks. The "BadUSB" class of attacks, exemplified by the **USB Rubber Ducky**, emulates a Human Interface Device (HID) to inject keystrokes at superhuman speeds, executing scripts to open reverse shells or exfiltrate data before network defenses can react.20

### 4.1 Threat Mechanics and Detection Theory

Operating systems implicitly trust devices claiming to be keyboards (USB Class 03). The Rubber Ducky exploits this by acting as a keyboard that "types" a malicious script.

* **Distinguishing Feature:** **Typing Speed**.
  + **Human:** Variable IAT, typically 150ms–300ms. Burst speeds rarely exceed 100 WPM (Words Per Minute).
  + **HID Injection:** Consistent, extremely low IAT (e.g., 10-20ms). Theoretical speeds exceed 1000 WPM.22

Therefore, the detection logic relies on **Keystroke Dynamics Analysis** rather than simple device identifiers (VID/PID), which are easily spoofed.23

### 4.2 Layer 1: Device Enumeration via udev

The Linux udev subsystem manages device events. We employ the pyudev library to create an asynchronous monitor for the usb and input subsystems.

* **Trigger:** The monitor listens for ACTION=="add".
* **Heuristics:**
  1. **Composite Device Check:** A legitimate "Flash Drive" should rarely declare a HID Keyboard interface. If a device presents bInterfaceClass=08 (Storage) AND bInterfaceClass=03 (HID), it is highly suspicious (likely a Bash Bunny or malicious composite device).24
  2. **Allow-listing:** While spoofable, checking ID\_VENDOR against a whitelist of approved peripherals in a secure environment provides a first layer of defense.26

### 4.3 Layer 2: Behavioral Analysis via evdev

Upon detecting a new HID device, the system must actively monitor its input. The evdev library provides an interface to the Linux input subsystem (/dev/input/eventX).

#### 4.3.1 The Sliding Window Analyzer

We implement a dedicated USBMonitor class running in an asynchronous loop.

1. **Binding:** When udev reports a new keyboard at /dev/input/eventX, the monitor attaches an evdev.InputDevice reader.
2. **Data Collection:** It reads EV\_KEY events (Key Down / Key Up) and timestamps them.
3. **Metric Calculation:** It calculates the delta between consecutive Key Down events (Inter-Arrival Time).
4. **Sliding Window:** A FIFO buffer stores the last $N$ (e.g., 50) IAT values.
   * Calculate **Window Mean** ($ \mu\_{window} $) and \*\*Window Variance\*\* ($ \sigma^2\_{window} $).
5. **Detection Thresholds:**
   * **Attack Condition:** If $\mu\_{window} < 25ms$ (approx 400 WPM equivalent) AND $\sigma^2\_{window} < Threshold\_{variance}$ (Machine-like consistency).
   * The low variance check is crucial to distinguish a machine from a user "mashing" keys randomly.27

#### 4.3.2 Mitigation: The "Unbind" Response

If an attack is detected, simply alerting is often too slow. The system must sever the connection.

* **Mechanism:** Write the device's bus ID to the sysfs unbind file.
  + Command: echo '1-1.2' > /sys/bus/usb/drivers/usb/unbind
  + This logically disconnects the device at the kernel driver level, stopping the keystroke injection immediately.29

## 5. Daemon Process Architecture

The system must handle high-bandwidth network capture (CPU intensive), USB monitoring (I/O and Event driven), and Deep Learning inference (Matrix operations) simultaneously. A single-threaded Python script is insufficient due to the GIL, which serializes CPU-bound operations.

### 5.1 Hybrid Multiprocessing Architecture

We propose a **Multi-Process Daemon** design, utilizing multiprocessing to isolate CPU-heavy tasks and asyncio for event loops.30

**Process Breakdown:**

1. **Process A: Network Ingestion (NFStream Wrapper)**
   * **Type:** multiprocessing.Process
   * **Role:** Runs the NFStreamer loop. The C-based engine of NFStream releases the GIL during capture, allowing efficient usage of its core.
   * **Data Flow:** Extracts features -> Applies Plugin Logic -> Pushes Feature Vectors to NetworkQueue.
   * **Optimization:** Pinned to specific CPU cores to maximize cache locality.
2. **Process B: Endpoint Monitor (Asyncio Loop)**
   * **Type:** multiprocessing.Process running an asyncio loop.
   * **Role:** Runs pyudev observers and multiple evdev readers as concurrent async tasks. This is ideal for I/O-bound operations where the process spends most time waiting for keypresses.
   * **Data Flow:** Detects Hardware -> Monitors Keystrokes -> Pushes Alerts to LogQueue.
3. **Process C: Inference Engine (PyTorch)**
   * **Type:** multiprocessing.Process
   * **Role:** Loads the heavy PyTorch model (Autoencoder). It runs a continuous loop consuming from NetworkQueue.
   * **Batching:** To improve throughput, it can fetch multiple vectors from the queue to perform batch inference (e.g., batch size 64) rather than predicting one by one.
   * **Data Flow:** Pulls Vectors -> Normalizes (Frozen Scaler) -> Infers -> Pushes Scores to LogQueue.
4. **Process D: Central Logger & Orchestrator**
   * **Type:** Main Process or dedicated Process.
   * **Role:** Consumes from LogQueue. Handles I/O operations to disk (JSONL writing) or external SIEMs (Syslog/Kafka). This decouples blocking disk I/O from the critical path of packet processing.

### 5.2 Inter-Process Communication (IPC)

**Queues:**

We utilize multiprocessing.Queue for IPC. While robust, standard queues rely on pickling, which has overhead.

* **Optimization:** For the NetworkQueue (High Volume), if performance bottlenecks arise, we can transition to multiprocessing.shared\_memory (available in Python 3.8+). This allows the Ingestion process to write feature arrays directly into a shared RAM block, which the Inference process reads without serialization copying, significantly boosting throughput.31

**Signal Handling:**

The architecture must implement a SignalHandler listening for SIGINT and SIGTERM. Upon a stop signal, the parent process must:

1. Set a Event flag to notify child processes.
2. Allow NFStream to finish its current batch.
3. Join all processes to prevent zombie states.33

## 6. Structured Logging & Data Persistence

For Phase 5, logs serve two purposes: real-time alerting and historical training data for Phase 6.

### 6.1 Schema Standard: Elastic Common Schema (ECS)

To ensure interoperability with enterprise tools (ELK Stack, Splunk), we adopt the **Elastic Common Schema (ECS)**.34

**Mapping Example:**

* **Timestamp:** @timestamp (ISO 8601).
* **Network Data:**
  + source.ip, source.port
  + destination.ip, destination.port
  + network.transport (TCP/UDP)
  + network.bytes, network.packets
* **AICyberDefense Specifics:**
  + event.kind: "event" or "alert"
  + event.category: "intrusion\_detection"
  + rule.name: "Autoencoder\_Anomaly" or "BadUSB\_Keystroke"
  + aicd.anomaly\_score: Float value (Reconstruction Error).

### 6.2 Storage Engine: JSONL vs. SQLite

The requirement is high-frequency writing.

* **SQLite:** Offers ACID compliance and powerful querying. However, the Write-Ahead Logging (WAL) and locking overhead can limit write throughput in extremely high-speed logging scenarios. It is better suited for the *analysis* of logs, not the generation.36
* **JSONL (JSON Lines):** A text format where each line is a valid JSON object.
  + **Pros:** Append-only (O(1) complexity), zero locking overhead, human-readable, natively supported by log shippers (Filebeat, Fluentd).
  + **Decision:** We will use **JSONL** for the active log stream. A separate process (or Phase 6 tool) can bulk-load these logs into SQLite or a SIEM for analysis.

## 7. Implementation Roadmap & Code Strategy

### 7.1 Prerequisite Configuration

The Linux host requires specific configuration to allow non-root users to capture packets and monitor input:

Bash

# Allow Python to capture packets without root  
sudo setcap cap\_net\_raw,cap\_net\_admin+eip $(readlink -f $(which python3))  
  
# Allow user access to input devices (for USB monitoring)  
sudo usermod -aG input $USER

### 7.2 Component Implementation Details

**A. The CICFeaturePlugin (Python Code Concept):**

Python

from nfstream import NFPlugin  
import math  
  
class CICFeaturePlugin(NFPlugin):  
 def on\_init(self, packet, flow):  
 flow.udps.last\_ts = packet.time  
 # Welford State: mean, M2 (sum squared diff), count  
 flow.udps.iat\_mean = 0.0  
 flow.udps.iat\_m2 = 0.0  
 flow.udps.iat\_count = 0  
   
 # Flag Counters  
 flow.udps.syn\_cnt = 0  
 flow.udps.fin\_cnt = 0  
 #... other init...  
  
 def on\_update(self, packet, flow):  
 # IAT Calculation  
 if flow.udps.iat\_count > 0:  
 delta = packet.time - flow.udps.last\_ts  
 flow.udps.iat\_count += 1  
 delta\_n = delta - flow.udps.iat\_mean  
 flow.udps.iat\_mean += delta\_n / flow.udps.iat\_count  
 delta\_n2 = delta - flow.udps.iat\_mean  
 flow.udps.iat\_m2 += delta\_n \* delta\_n2  
 else:  
 flow.udps.iat\_count = 1  
   
 flow.udps.last\_ts = packet.time  
   
 # Flag Logic  
 if packet.tcp\_flags & 0x02: flow.udps.syn\_cnt += 1  
 #...  
  
 def on\_expire(self, flow):  
 # Finalize Std Dev  
 if flow.udps.iat\_count > 1:  
 flow.udps.iat\_std = math.sqrt(flow.udps.iat\_m2 / (flow.udps.iat\_count - 1))  
 else:  
 flow.udps.iat\_std = 0.0  
   
 # Calculate Rates  
 duration\_sec = flow.bidirectional\_duration\_ms / 1000.0  
 if duration\_sec > 0:  
 flow.udps.flow\_bytes\_sec = flow.bidirectional\_bytes / duration\_sec  
 else:  
 flow.udps.flow\_bytes\_sec = 0

**B. The USB Monitoring Logic (Python Code Concept):**

Python

async def monitor\_device(device\_path):  
 device = evdev.InputDevice(device\_path)  
 window = collections.deque(maxlen=50)  
 last\_time = time.time()  
   
 async for event in device.async\_read\_loop():  
 if event.type == evdev.ecodes.EV\_KEY and event.value == 1: # Key Down  
 curr\_time = time.time()  
 iat = curr\_time - last\_time  
 last\_time = curr\_time  
 window.append(iat)  
   
 if len(window) == 50:  
 mean\_iat = statistics.mean(window)  
 var\_iat = statistics.variance(window)  
   
 # Heuristic: Fast (<25ms) and Robotic (Variance < 0.001)  
 if mean\_iat < 0.025 and var\_iat < 0.001:  
 alert\_queue.put({  
 "event.kind": "alert",  
 "rule.name": "BadUSB\_Detected",  
 "usb.device": device.name,  
 "metric.iat\_mean": mean\_iat  
 })  
 # Active Response: Unbind driver

## 8. Conclusion

Phase 5 establishes a robust, production-grade foundation for the AICyberDefense system. By rigorously selecting **NFStream** over Scapy/Pyshark, the system achieves the necessary throughput to handle enterprise traffic loads. The custom **NFPlugin** architecture successfully bridges the gap between raw packet capture and the complex statistical features required by the **CICIDS2017**-trained models, ensuring high fidelity in detection.

Simultaneously, the introduction of the **USB Monitoring Subsystem** addresses a critical blind spot in traditional NIDS, utilizing keystroke dynamics to identify physical injection attacks that bypass network perimeters. The **Multi-Process Daemon Architecture** solves the concurrency challenges inherent in Python, providing a scalable and fault-tolerant platform.

This framework not only meets the immediate requirements of real-time detection but provides a flexible substrate for Phase 6 (Visualization) and the future adversarial hardening (Phase 3) where the system must defend against manipulated inputs. The use of standardized logging (ECS/JSONL) ensures that this data is not just transient but forms a valuable historical record for forensic analysis.