NetPulse: Real-Time Network Analysis Dashboard with AI-Driven Insights

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the

HPE CPP-2 Program

Submitted By:

Arpit Khandelwal | Avi Maheshwari | Divesh Jain | Lakshya Gupta | Madhur Kedia

Under the Guidance of:

Mr. Isaac Theogaraj | Ms. Vishakha Hegde

Ms. Ruchika Solanki

December 2025

Acknowledgement

We would like to express our deepest gratitude to the **Hewlett Packard Enterprise (HPE) CPP Program** for providing us with the opportunity to work on this industry-relevant project. The exposure to real-world networking concepts and modern development stacks has been invaluable to our professional growth.

We are extremely grateful to our mentors at **Hewlett Packard Enterprise (HPE)**, **Mr. Isaac Theogaraj** and **Ms. Vishakha Hegde**, for their continuous guidance, technical insights, and encouragement throughout the development phase. Their feedback on architecture and performance optimization helped us bridge the gap between theoretical knowledge and practical implementation.

We express our sincere thanks to **Mr. Arpit Agrawal**, Director of **JECRC**, for providing us such a great infrastructure and environment for our overall development.
We also express sincere thanks to **Dr. V. K. Chandna**, Principal of JECRC, for his kind cooperation and extended support towards this program.
We are also deeply grateful to **Ms. Ruchika Solanki**, our HPE Faculty Mentor for the institute, for guiding us and providing ample support during this period.

Also, our warm thanks to **Jaipur Engineering College and Research Centre**, which provided us this opportunity to carry out this prestigious seminar and enhance our learning in various technical fields.

Abstract

With the exponential growth of digital traffic, ensuring network reliability and security has become paramount. However, traditional network analysis tools, such as Wireshark, often present a steep learning curve due to their complex command-line interfaces and raw packet-level data presentation. This project, **NetPulse**, addresses this challenge by developing a high-performance, browser-based network monitoring dashboard that democratizes access to critical network insights.

The system is architected using a decoupled client-server model. The backend, built with **Python**, leverages the **asyncio** library to perform non-blocking, real-time packet capture via **TShark**, ensuring the system can process high-velocity traffic without latency. Data is streamed instantaneously via **WebSockets** to a **React.js** frontend, which renders dynamic visualizations for key metrics including Throughput, Goodput, Jitter, and Latency. Unlike standard monitoring tools, NetPulse offers advanced features such as interactive Geolocation mapping of destination servers and Sankey diagrams for visualizing top outbound conversations.

A distinguishing innovation of this project is the integration of **Generative AI**. By feeding aggregated session metrics into a Large Language Model (LLM), the system generates automated, human-readable health summaries and anomaly detection reports, significantly reducing the time required for root cause analysis. NetPulse is a robust, scalable solution for real-time network observability, effectively bridging the gap between granular packet inspection and high-level visual monitoring.

Table of Contents

List of Figures

Chapter-1 Introduction

Modern networks generate large volumes of continuous traffic, making real-time performance monitoring essential for understanding throughput, latency trends, protocol usage, and overall network behaviour. Traditional tools such as Wireshark provide deep packet inspection but are not optimized for presenting live, high-level metrics in an accessible dashboard format. This creates a practical gap for users who need immediate visibility and simplified analysis during ongoing network activity.

NetPulse is designed to address this gap by offering a lightweight, real-time network analysis dashboard that converts raw packet data into meaningful performance indicators. The system combines an asynchronous Python backend—responsible for high-speed packet capture and metric computation—with a React-based frontend that visualizes live throughput, jitter, latency, protocol distribution, traffic composition, and packet statistics through an intuitive interface. WebSocket streaming ensures smooth updates, enabling users to monitor changes instantly.

An additional enhancement within NetPulse is the integration of a Generative AI summarizer, which periodically interprets active metrics and provides concise, human-readable insights. This allows users to understand network conditions quickly without manually interpreting raw packet data. This platform also ensures broad relevance across common network scenarios.

Overall, NetPulse delivers a focused and efficient approach to real-time network observability, offering accurate metric computation, responsive visualizations, and AI-driven summaries within a single, browser-based environment.

1.1 Background

In the era of high-speed digital infrastructure, network observability has become a critical requirement for maintaining system reliability and security. Modern networks generate vast amounts of data every second, making manual monitoring increasingly impractical. While packet-level data provides the ultimate source of truth for network diagnostics, the sheer volume and velocity of this traffic often overwhelm traditional analysis methods. Consequently, there is a growing demand for tools that can bridge the gap between low-level data capture and high-level visual intelligence, enabling operators to assess network health instantaneously.

1.2 Problem Statement

Existing packet analysis tools are powerful but often complex, resource-heavy, or unsuitable for real-time visualization. Users manually interprets raw packet data, and traditional dashboards lack the ability to provide continuous, clear summaries of network behaviour. There is a gap between low-level packet capture utilities and easy-to-use, live monitoring platforms that can present throughput, latency, jitter, protocol distribution, and traffic composition in a concise way. Therefore, there is a distinct need for a lightweight, browser-based solution that combines the precision of packet capture with the usability of modern web dashboards.

1.3 Objectives

The primary objective of NetPulse is to develop a real-time network analysis dashboard that provides accurate performance metrics through a simple browser-based interface. The specific objectives include:

- **Develop a Browser-Based Dashboard:** To create a responsive frontend using React.js that visualizes network metrics dynamically without requiring heavy client-side resources.

- **Implement Efficient Data Processing:** To engineer a Python-based backend capable of asynchronous packet parsing and data retrieval to minimize latency during high-traffic capture sessions.

- **Calculate QoS Metrics:** To mathematically compute Quality of Service (QoS) metrics, specifically Jitter (using RFC 3550 standards) and Latency (weighted moving averages), rather than relying solely on raw packet counts.

- **Integrate Artificial Intelligence:** To embed Generative AI (Google Gemini) for automated root cause analysis, converting complex statistical data into natural language health summaries for the user.

1.4 Scope

NetPulse focuses on real-time network observability by supporting essential protocols including TCP, UDP, QUIC, RTP, DNS, and IGMP protocols, along with general IPv4/IPv6 traffic differentiation. The system provides live metrics, packet tables, geolocation mapping and protocol-level analysis like Inbound/Outbound Throughput, Goodput (effective data rate), Packet Loss, Jitter, and Round-Trip Time (RTT) Latency , through a responsive React dashboard. The project is designed to function as an efficient, lightweight, and accessible tool for both academic and practical network monitoring scenarios.

Chapter 2: Literature Review and Competitive Analysis

2.1 Overview of Existing Tools:

The current landscape of network monitoring is dominated by tools that have specific limitations regarding real-time observability. Wireshark, along with its command-line counterpart TShark, stands as the industry standard for deep packet inspection and protocol dissection. However, it is fundamentally designed as a desktop application for post-event forensic analysis rather than real-time health monitoring, its tabular interface is ill-suited for identifying long-term trends or instantaneous spikes in traffic volume at a glance and it is not optimized for continuous real-time visualization or high-level metric tracking. On the web-based front, ntopng offers robust flow-based analysis and visualization capabilities, yet it often requires complex configuration and significant system resources to manage historical data, rendering it is less ideal for lightweight, on-demand deployment. At the enterprise level, solutions like SolarWinds and Network Miner provide comprehensive feature sets but are often accompanied by prohibitive costs and complex licensing structures, making them inaccessible for smaller development teams or educational contexts.

2.2 Gap Analysis

A review of existing tools highlights several limitations that affect accessibility and real-time usability. Existing browser-based solutions often struggle to balance the computational load of accurate packet capture with smooth User Interface rendering. Most open-source network analyzer tools emphasize deep inspection rather than live performance metrics, making them less suitable for quick monitoring during active traffic. Visualization of protocol distribution, IPv4/IPv6 ratios, and encrypted versus unencrypted traffic is not instantly available in many tools. Additionally, AI-driven interpretative summaries are largely absent from standard monitoring platforms. These gaps underline the need for a system that combines real-time performance metrics with clear, simplified visualization and automated insights.

2.3 Proposed Solution (NetPulse)

NetPulse is designed to address these limitations by delivering a modern, browser-based dashboard capable of real-time metric visualization. Its asynchronous Python backend ensures efficient packet capture, while the React frontend renders live throughput, jitter, latency, protocol statistics, and traffic composition with minimal delay. The integration of a Generative AI summarizer further distinguishes the system by providing periodic, human-readable interpretations of network conditions. Through this combination of speed, clarity, and intelligent reporting, NetPulse offers a practical and accessible approach to real-time network monitoring.

Chapter 3: System Design & Architecture

3.1 High-Level Architecture Overview

3.1.1 Data Workflow

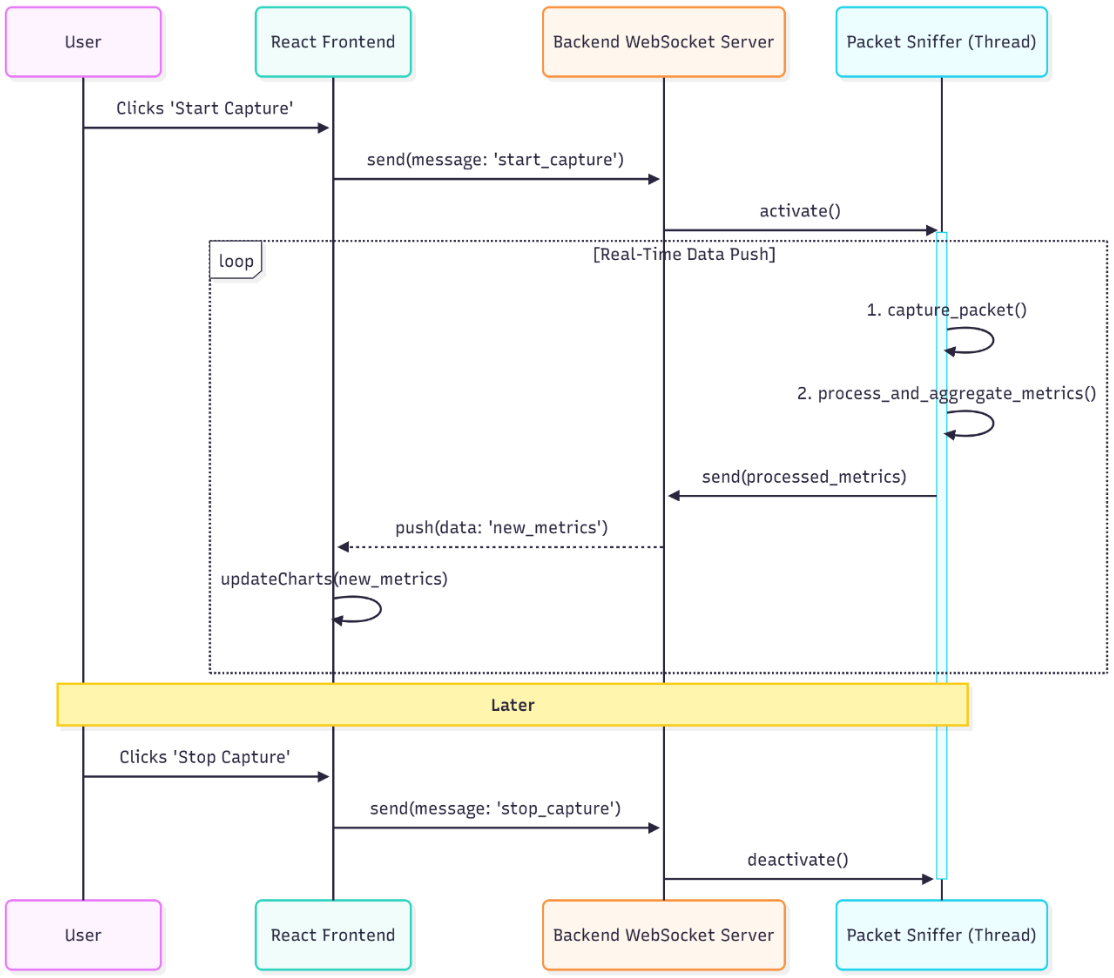


Figure 3.1 High-Level Data Workflow Sequence Diagram

3.1.2 LLM Summary Workflow

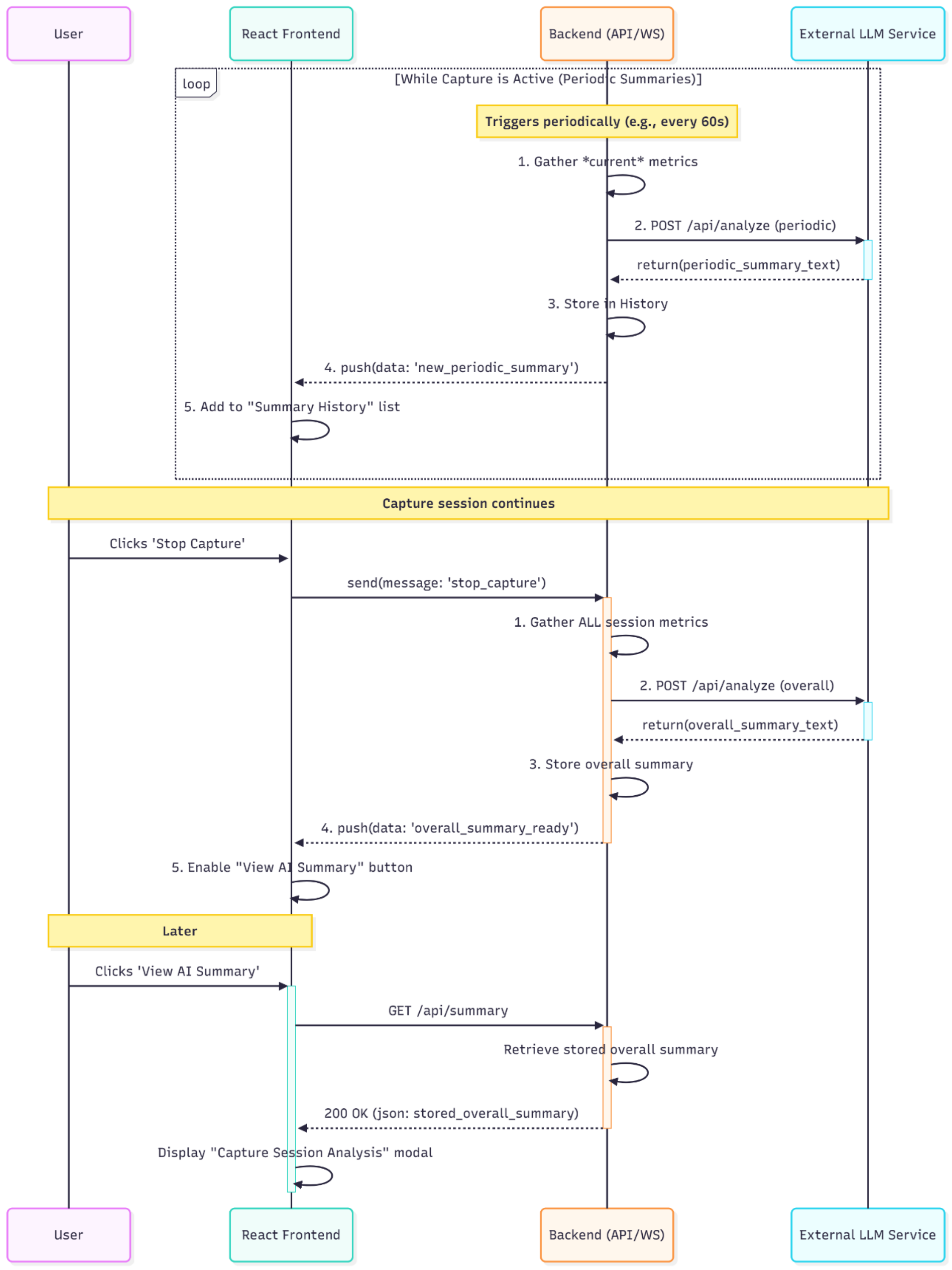
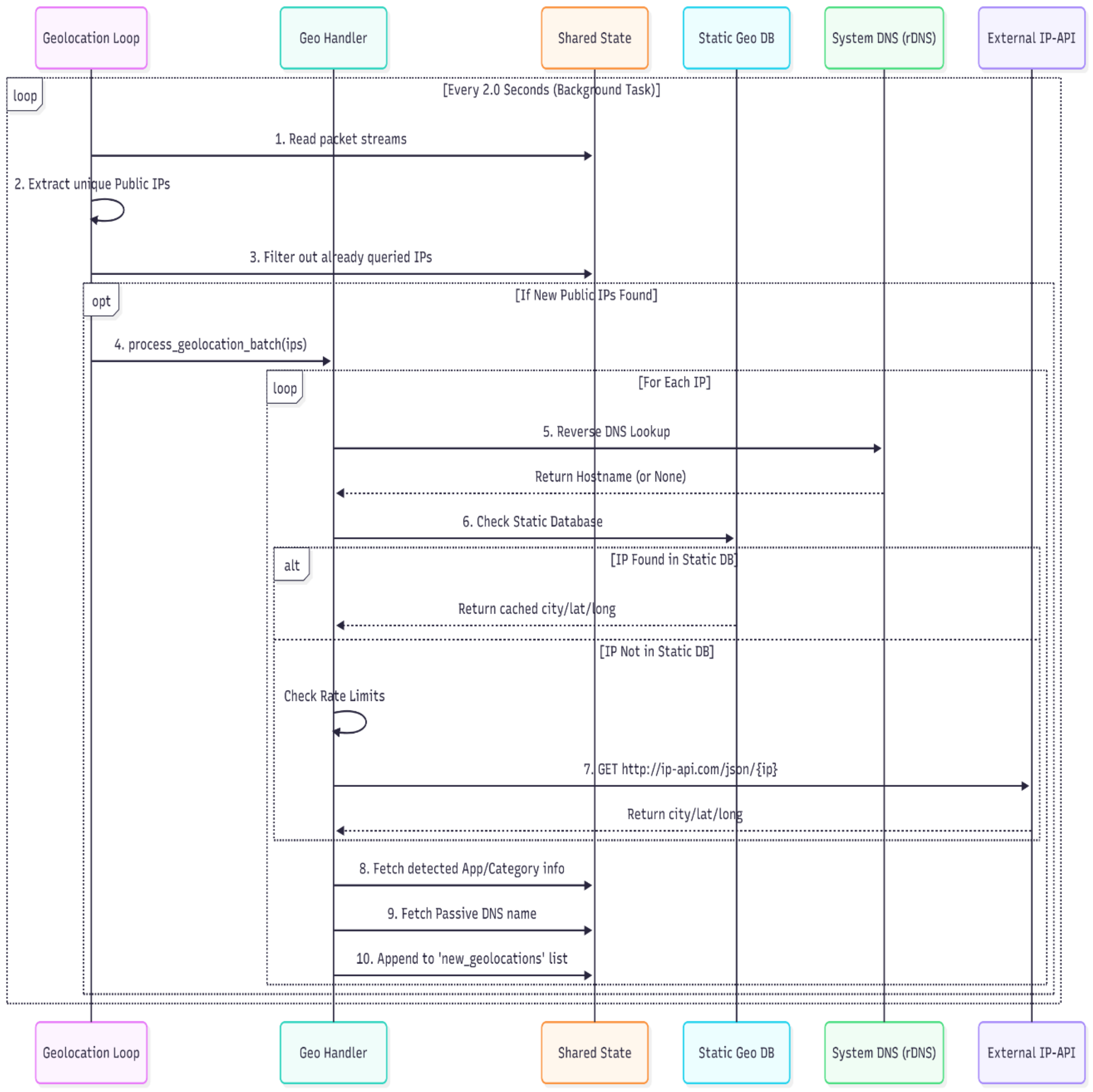

User
React Frontend
Backend (API/WS)
External LLM Service
loop
[While Capture is Active (Periodic Summaries)]
Triggers periodically (e.g., every 60s)
1. Gather *current* metrics
2. POST /api/analyze (periodic)
return(periodic_summary_text)
3. Store in History
4. push(data: 'new_periodic_summary')
5. Add to "Summary History" list
Capture session continues
Clicks 'Stop Capture'
send(message: 'stop_capture')
1. Gather ALL session metrics
2. POST /api/analyze (overall)
return(overall_summary_text)
3. Store overall summary
4. push(data: 'overall_summary_ready')
5. Enable "View AI Summary" button
Later
Clicks 'View AI Summary'
GET /api/summary
Retrieve stored overall summary
200 OK (json: stored_overall_summary)
Display "Capture Session Analysis" modal
User
React Frontend
Backend (API/WS)
External LLM Service

Figure 3.2 LLM Summary Workflow Sequence Diagram

3.1.3 Geolocation Workflow



Geolocation Loop
Geo Handler
Shared State
Static Geo DB
System DNS (rDNS)
External IP-API
loop
[Every 2.0 Seconds (Background Task)]
1. Read packet streams
2. Extract unique Public IPs
3. Filter out already queried IPs
opt
[If New Public IPs Found]
4. process_geolocation_batch(ips)
loop
[For Each IP]
5. Reverse DNS Lookup
Return Hostname (or None)
6. Check Static Database
alt
[IP Found in Static DB]
Return cached city/lat/long
[IP Not in Static DB]
Check Rate Limits
7. GET http://ip-api.com/json/{ip}
Return city/lat/long
8. Fetch detected App/Category info
9. Fetch Passive DNS name
10. Append to 'new_geolocations' list

Figure 3.3 Geolocation Workflow Sequence Diagram

The system is organized as a two-tier architecture with a backend for data capture and processing and a frontend for visualization. The backend is implemented in Python (v3.13) and runs on the monitoring host, containing the Capture Engine and various processing and analytics modules. The frontend is a web-based interface built using React (v19) and served via a local development server powered by Vite. Communication between the backend and frontend occurs through a persistent WebSocket channel, enabling real-time, bidirectional data exchange without requiring explicit polling. The backend streams updates as soon as they are computed, and the React client immediately reflects these changes through component state updates triggered by WebSocket messages. WebSockets provide a full-duplex

communication channel well-suited for pushing fresh data from server to client as soon as it is available, which is essential for maintaining a responsive and low-latency monitoring experience.
The backend is constructed using an asynchronous architecture, ensuring that no single operation blocks the event loop. Each major subsystem executes independently using non-blocking coroutines, allowing concurrent capture, computation, and communication. The system follows a modular, microservice-inspired design, where:

- The Capture Engine performs packet acquisition and parsing,
- The WebSocket Server manages client connections and state dissemination, and
- Shared State Memory acts as the canonical repository of all metrics produced during runtime.

Crucially, three key backend services operate as independent asynchronous tasks, running continuously and in parallel:

1. **Data Collection Loop** – consumes packets captured by Tshark, updates protocol-level and flow-level metrics, and broadcasts incremental deltas to all active clients.
2. **Geolocation Handler** – periodically extracts unseen public IPs from active traffic and resolves their geographic positions using static and API-backed geolocation lookup strategies.
3. **Periodic LLM Summary Loop** – triggers the AI summary generation workflow at fixed intervals, preparing human-readable insights and trend analyses from accumulated network metrics.

By decoupling these background services, the architecture ensures scalability and prevents bottlenecks—packet capture, geolocation enrichment, and AI-driven summarization progress concurrently without interfering with UI responsiveness or transport-layer operations.

3.1.4 Architectural Principles

The architecture adheres to the following core principles:

- Asynchronous Design: All I/O operations are non-blocking, ensuring the event loop remains responsive to WebSocket events and real time client requests, while packet capture and metrics processing runs concurrently.
- Stateless Processing Pipeline: Data flows through multiple processing stages (capture → metrics calculation → serialization) without maintaining intermediate state across modules, reducing coupling and improving maintainability.

- Shared State Pattern with Centralized Repository: Global application state is maintained in a dedicated shared_state module, acting as the single source of truth for all metrics, configuration, and runtime data across distributed components.
- Event-Driven Communication: Components communicate via event-driven patterns, where state changes trigger WebSocket broadcasts to connected clients, implementing a push-based update model rather than polling.

3.2 Data Flow Architecture

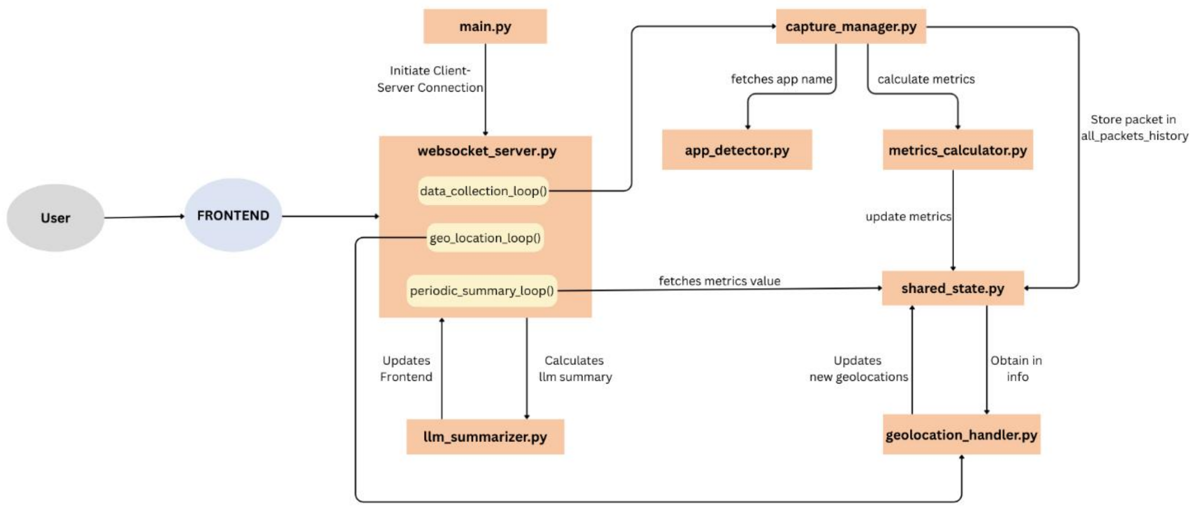


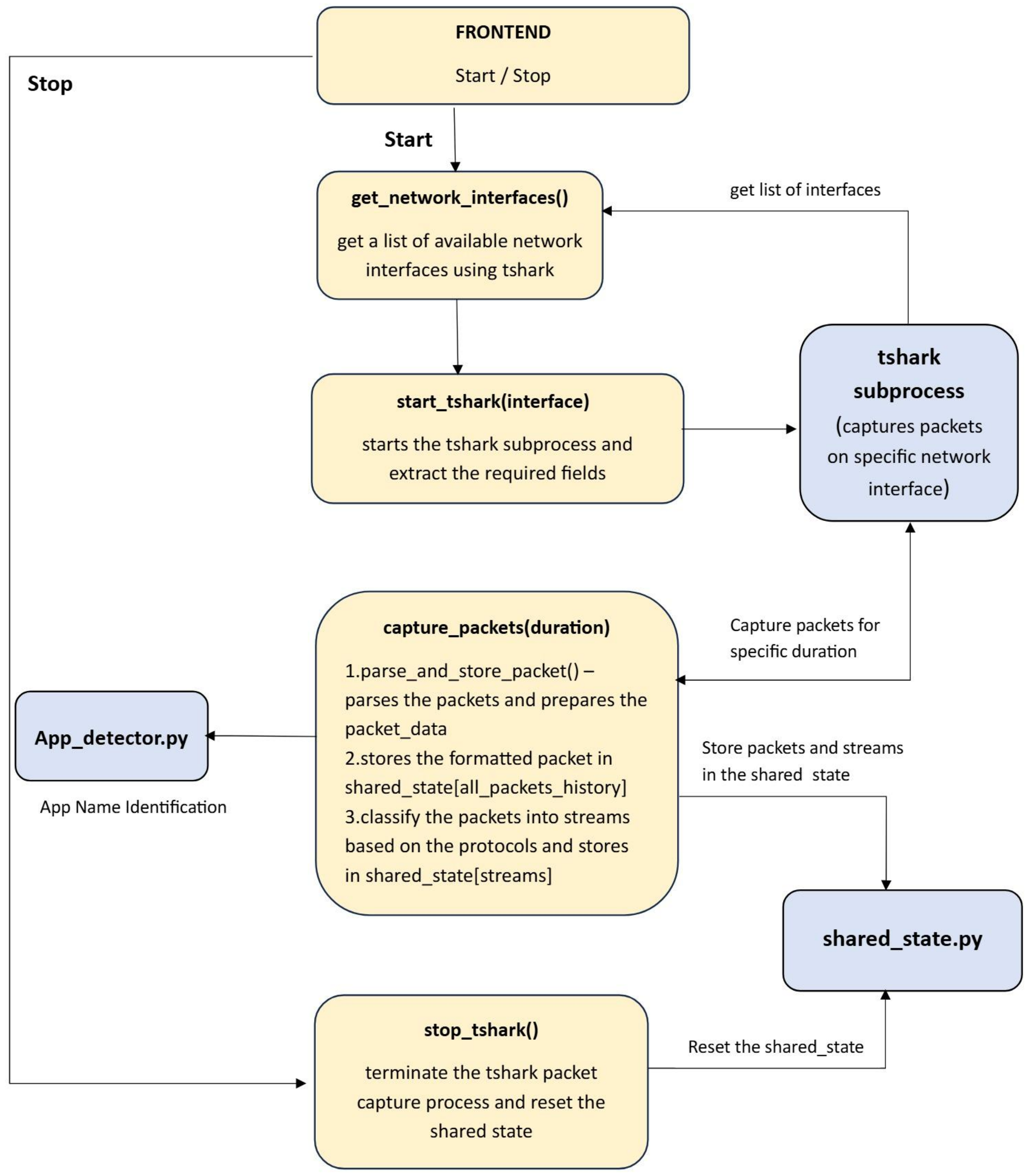
Figure 3.4 Data Flow Architecture

3.2.1 Packet Capture and Processing Pipeline

The packet capture pipeline operates as a continuous, non-blocking asynchronous loop that processes network traffic in real-time:

1. Interface Selection and Packet Acquisition (capture_manager.py)

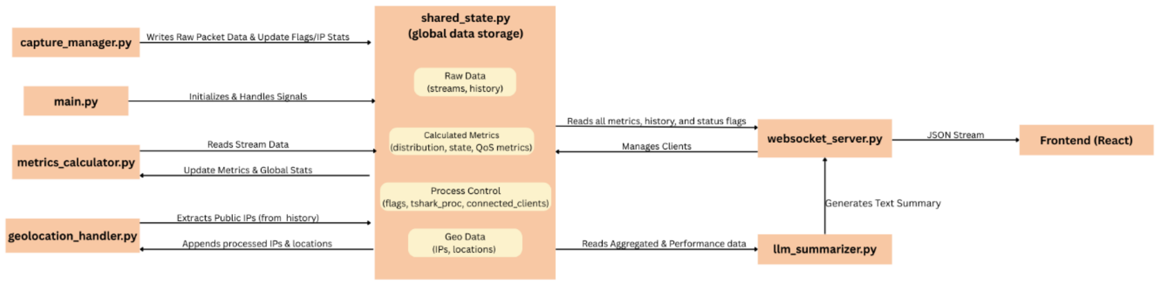
At startup, the capture manager enumerates available NICs using system utilities and selects one for monitoring. The chosen interface is configured and stored in the system state. The backend launches Tshark in a subprocess bound to the selected interface. Tshark is configured to output relevant packet fields (timestamps, IPs/ports, DNS queries/responses, TLS SNI, frame lengths, etc.) with line-buffering enabled . As packets arrive, Tshark writes each packet's data line-by-line to stdout. The Python code reads this output asynchronously.



FRONTEND
Start / Stop
Stop
Start
get_network_interfaces()
get a list of available network interfaces using tshark
get list of interfaces
start_tshark(interface)
starts the tshark subprocess and extract the required fields
tshark subprocess
(captures packets on specific network interface)
capture_packets(duration)
1.parse_and_store_packet() – parses the packets and prepares the packet_data
2.stores the formatted packet in shared_state[all_packets_history]
3.classify the packets into streams based on the protocols and stores in shared_state[streams]
Capture packets for specific duration
App_detector.py
App Name Identification
Store packets and streams in the shared state
shared_state.py
stop_tshark()
terminate the tshark packet capture process and reset the shared state
Reset the shared_state

Figure 3.5 Packet Capture Process Flow

2. Raw Packet Parsing and Stream Organization(capture_manager.py)

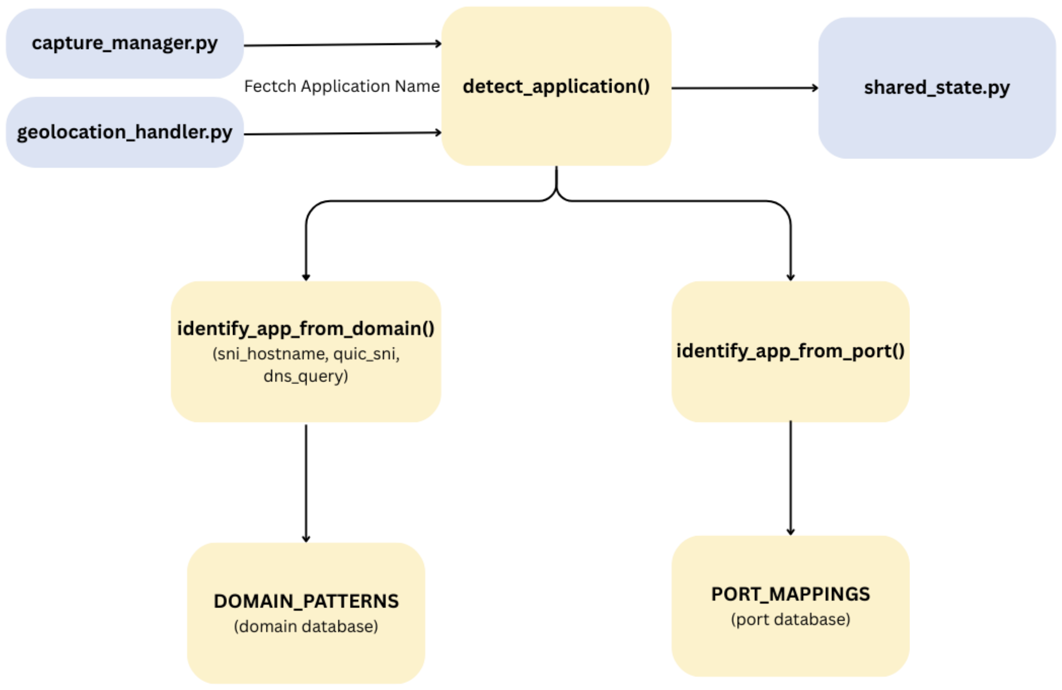


capture_manager.py
Writes Raw Packet Data & Update Flags/IP Stats
main.py
Initializes & Handles Signals
metrics_calculator.py
Reads Stream Data
Update Metrics & Global Stats
geolocation_handler.py
Extracts Public IPs (from history)
Appends processed IPs & locations
shared_state.py (global data storage)
Raw Data (streams, history)
Calculated Metrics (distribution, state, QoS metrics)
Process Control (flags, tshark_proc, connected_clients)
Geo Data (IPs, locations)
Reads all metrics, history, and status flags
Manages Clients
websocket_server.py
JSON Stream
Frontend (React)
Generates Text Summary
Reads Aggregated & Performance data
llm_summarizer.py

Figure 3.6 Raw Data Parsing Logic

The Capture Manager ingests the raw data stream from Tshark and normalizes the packet records into structured objects. It organizes these packets into logical streams (e.g., TCP or UDP flows) based on unique stream identifiers. These structured objects are then committed to the centralized Shared State memory, making them immedziately available for downstream analysis by the Metrics Calculator and WebSocket Server.

3. Application Detection(app_detector.py)



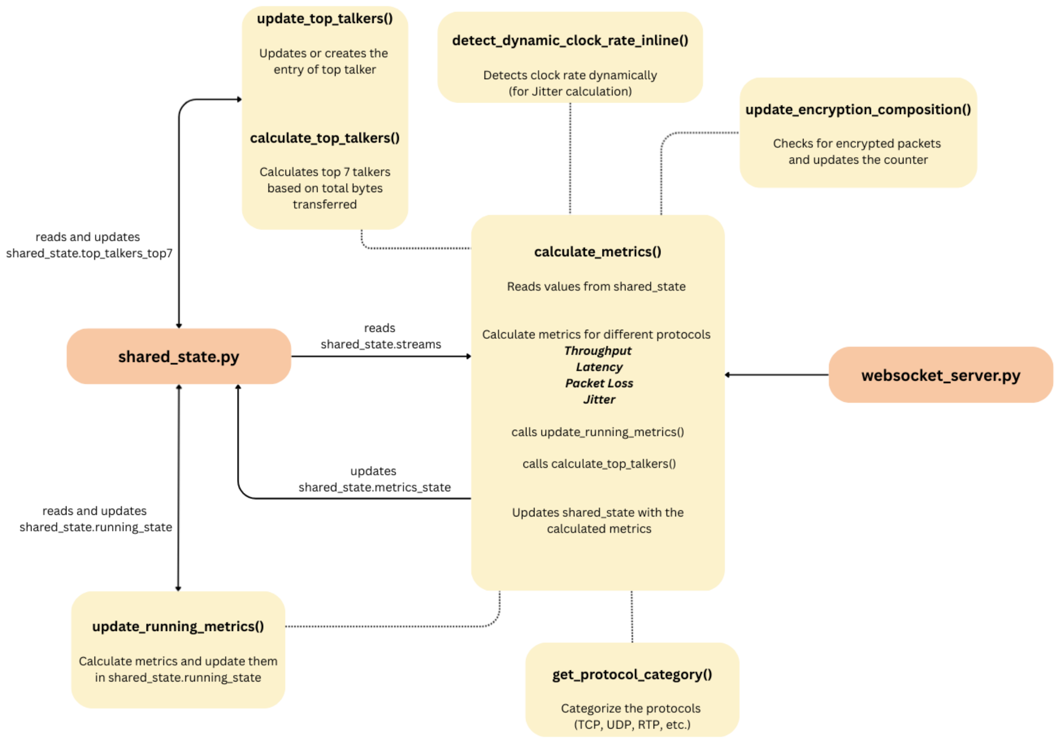
capture_manager.py
Fectch Application Name
geolocation_handler.py
detect_application()
shared_state.py
identify_app_from_domain() (sni_hostname, quic_sni, dns_query)
identify_app_from_port()
DOMAIN_PATTERNS (domain database)
PORT_MAPPINGS (port database)

Figure 3.7 Application Detection Logic

The Application Detection module operates as a classification layer within the processing pipeline. It inspects protocol metadata to tag traffic flows with human-readable application categories (e.g., 'Video Streaming' or 'Social Media'). These category tags are enriched directly into the flow data within the

Shared State, enabling the frontend to visualize traffic composition by application type rather than just raw port numbers.

4: Metrics Calculation and Running Averages(metrics_calculator.py)



update_top_talkers()
Updates or creates the entry of top talker
calculate_top_talkers()
Calculates top 7 talkers based on total bytes transferred
detect_dynamic_clock_rate_inline()
Detects clock rate dynamically (for Jitter calculation)
update_encryption_composition()
Checks for encrypted packets and updates the counter
reads and updates shared_state.top_talkers_top7
calculate_metrics()
Reads values from shared_state
Calculate metrics for different protocols
Throughput
Latency
Packet Loss
Jitter
calls update_running_metrics()
calls calculate_top_talkers()
Updates shared_state with the calculated metrics
shared_state.py
reads shared_state.streams
websocket_server.py
updates shared_state.metrics_state
reads and updates shared_state.running_state
update_running_metrics()
Calculate metrics and update them in shared_state.running_state
get_protocol_category()
Categorize the protocols (TCP, UDP, RTP, etc.)

Figure 3.8 Metrics Calculator Logic

The metrics_calculator.py module consumes parsed packets and aggregates from shared_state and updates performance metrics in real time. For each relevant flow or endpoint, it computes throughput, latency, jitter and packet loss. Based on the data calculated for each packet the average and cumulative data for various metrics, protocol distribution, ip composition, encryption composition, protocol specific data and cumulative metrics data is calculated. These metrics are updated on each packet arrival over sliding windows and written back into shared_state for display and periodic llm calculation.

5: WebSocket Broadcast(websocket_server.py)

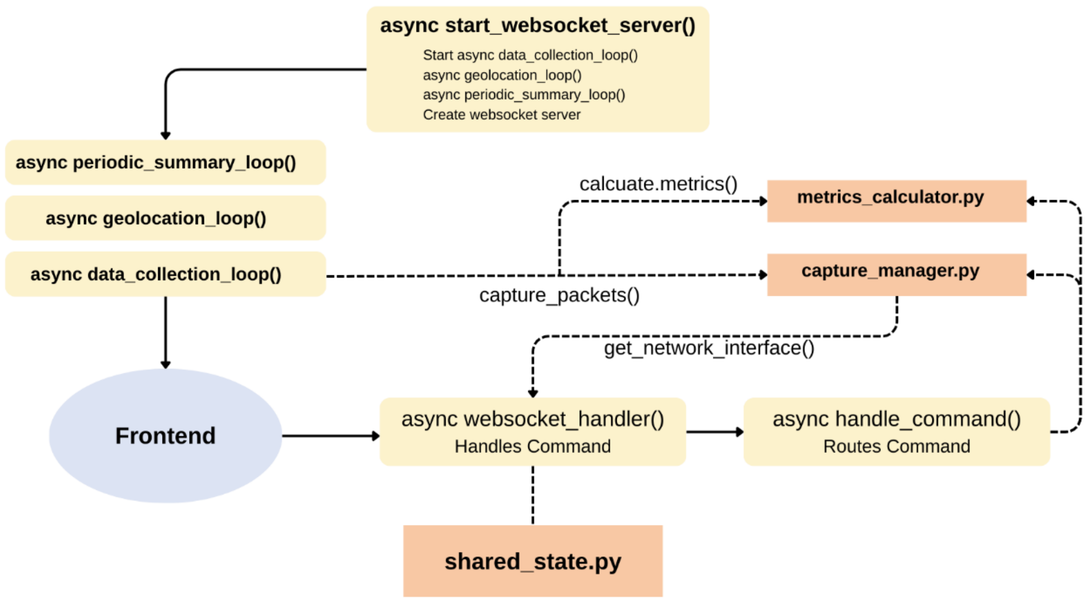


Figure 3.9 WebSocket Broadcast Architecture

Finally, the websocket_server.py module maintains the live channel to the React UI. As the shared_state is updated with new packets, computed metrics, geolocation info, or an AI summary, websocket_server.py serializes the relevant data and pushes it over the WebSocket to connected clients. The persistent WebSocket connection means the server can push updates instantly without waiting for client requests. On the frontend, React components (using a WebSocket hook) listen for incoming messages, parse the object, and update the charts/maps/display accordingly. This streaming architecture ensures the dashboard reflects network state in real time.

6. Geolocation Resolution (geolocation_handler.py):

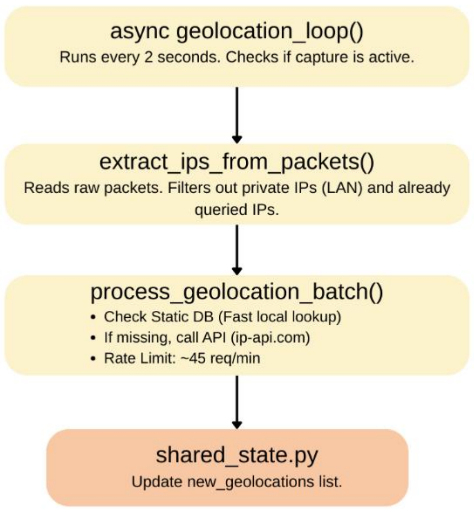

async geolocation_loop()
Runs every 2 seconds. Checks if capture is active.
extract_ips_from_packets()
Reads raw packets. Filters out private IPs (LAN) and already queried IPs.
process_geolocation_batch()
• Check Static DB (Fast local lookup)
• If missing, call API (ip-api.com)
• Rate Limit: ~45 req/min
shared_state.py
Update new_geolocations list.

Figure 3.10 Geolocation Resolution Flow

The Geolocation Handler functions as an independent background service that monitors the Shared State for newly discovered public IP addresses. It asynchronously resolves these IPs to physical coordinates (latitude/longitude) and enriches the host data. This decoupled approach ensures that external lookup latency does not impact the real-time packet processing loop, while still providing the frontend with accurate location data for the interactive map.

7. Periodic AI Summary Generation (llm_summarizer.py):

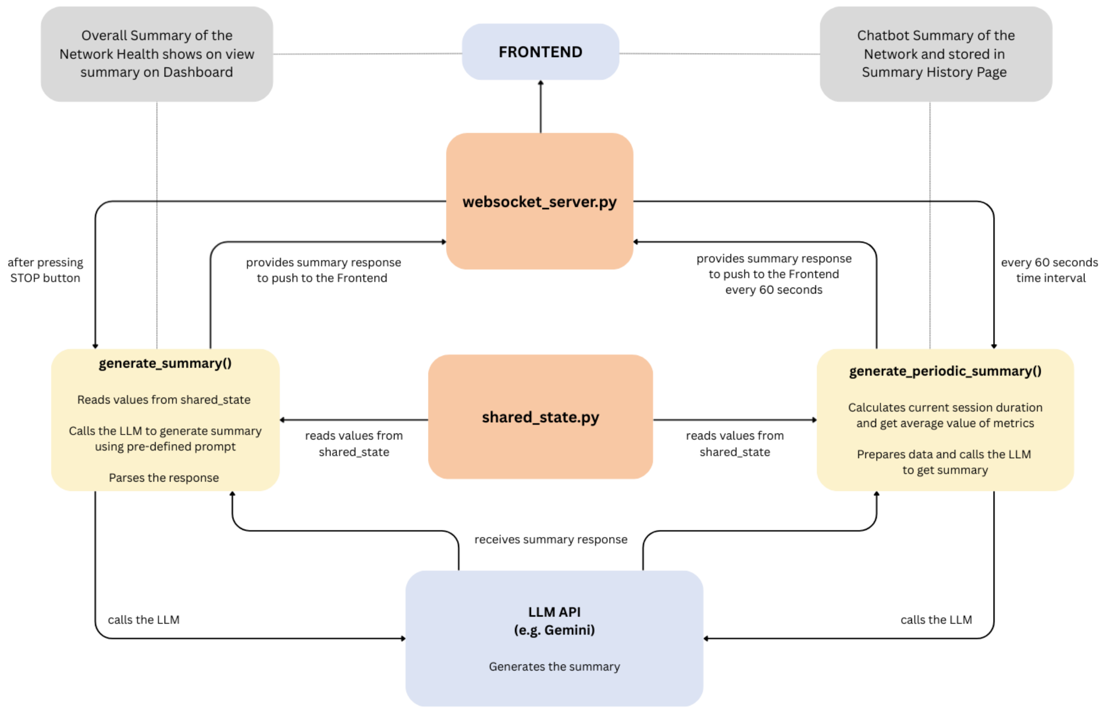

Overall Summary of the Network Health shows on view summary on Dashboard
FRONTEND
Chatbot Summary of the Network and stored in Summary History Page
websocket_server.py
after pressing STOP button
provides summary response to push to the Frontend
provides summary response to push to the Frontend every 60 seconds
every 60 seconds time interval
generate_summary()
Reads values from shared_state
Calls the LLM to generate summary using pre-defined prompt
Parses the response
shared_state.py
reads values from shared_state
reads values from shared_state
generate_periodic_summary()
Calculates current session duration and get average value of metrics
Prepares data and calls the LLM to get summary
receives summary response
calls the LLM
LLM API (e.g. Gemini)
Generates the summary
calls the LLM

Figure 3.11 LLM Summarizer Logic

The Periodic AI Summary service runs on a scheduled interval (e.g., every 60 seconds). It aggregates statistical network data from the Shared State and interfaces with an external Generative AI service to produce a natural language report. This narrative summary is then injected back into the system's event stream, where the WebSocket Server broadcasts it to connected clients, providing users with automated, high-level insights into network health.

Chapter 4: Methodology & Implementation Details

This chapter details the technical implementation of the NetPulse system, focusing on the algorithms used for packet processing, metric derivation, and the architectural patterns that enable real-time performance.

4.1 Backend Implementation

The backend is built using Python 3.13 and leverages the asyncio library to handle concurrent tasks like packet capture, metric calculation, and WebSocket streaming without blocking the main execution thread.

4.1.1 Asynchronous Packet Capture:

Traditional packet capture often relies on blocking I/O, which causes latency in high-traffic networks. NetPulse utilizes a non-blocking Producer-Consumer pattern.

- **Process Management:** The capture_manager.py module initiates tshark (the CLI version of Wireshark) as a subprocess using asyncio.create_subprocess_exec. This allows the system to read the standard output (stdout) stream asynchronously.
- **Field Extraction:** Instead of capturing full packet payloads, which is resource-intensive, the system instructs tshark to output specific fields (e.g., frame.time_epoch, ip.src, tcp.analysis.ack_rtt, rtp.ssrc) in a pipe-separated format (-T fields -E separator=|). This minimizes parsing overhead.
- **Efficient Storage:** Parsed packets are stored in a circular buffer-like structure (all_packets_history), and slicing is used (list[-display_count:]) to retrieve recent packets with O(1) complexity for the frontend display.

4.1.2 Network Metric Calculations:

The metrics_calculator.py module computes QoS (Quality of Service) metrics mathematically rather than relying solely on pre-computed values.

- **Throughput & Goodput:** Throughput is calculated by summing the total packet length (header + payload) over a specific time delta. Goodput, representing the application-level data rate, is derived by subtracting protocol headers (Ethernet, IP, TCP/UDP/RTP) from the total length.

$$\text{Throughput (bps)} = \frac{\sum \text{Packet Bits}}{\text{Duration}}$$

- **Jitter (RFC 3550 Implementation):** For Real-time Transport Protocol (RTP) streams, Jitter is calculated using the smoothing formula defined in IETF RFC 3550. The system dynamically detects the clock rate (e.g., 8000Hz for Audio, 90000Hz for Video) to normalize timestamps.

$$J(i) = J(i-1) + \frac{|D(i-1,i)| - J(i-1)}{16}$$

Where **D** is the difference in transit times between consecutive packets. This weighted moving average smoothens short-term fluctuations.

- **Latency (Weighted Average):**

TCP Round-Trip Time (RTT) is extracted from the tcp.analysis.ack_rtt field. To prevent inactive streams from skewing the global average, the system calculates a weighted average based on the packet count of each stream.

4.1.3 Heuristic Application Detection:

To identify the application layer (e.g., "YouTube", "Zoom") from raw packets, app_detector.py employs a prioritized heuristic engine:

1. TLS SNI (Server Name Indication): Extracts the hostname from the TLS handshake for encrypted traffic.
2. QUIC SNI: specific parsing for QUIC/HTTP3 packets.
3. DNS Queries: Maps IP addresses to recently resolved DNS names.
4. IP Cache: Checks a local cache of previously identified IPs.
5. Port Mapping: Falls back to standard ports (e.g., 443 for HTTPS, 53 for DNS) if deep inspection fails. This multi-layered approach ensures high classification rates even for encrypted traffic.

4.1.4 Hybrid Geolocation Strategy:

Geolocation lookups can be latency-expensive and rate-limited. NetPulse implements a hybrid strategy in geolocation_handler.py:

1. Static Database: Common public IPs (Google, Cloudflare, AWS) are looked up in a local dictionary (STATIC_GEOLOCATION_DB) for instant O(1) retrieval.
2. Async API Fallback: If the IP is not found locally, an asynchronous request is made to an external Geolocation API. These requests are batched and rate-limited to avoid API throttling.

4.2 Frontend Implementation

The frontend is a Single Page Application (SPA) built with React 19 and Vite, optimized for high-frequency data updates.

4.2.1 Real-Time State Management:

The custom hook useWebSocket.js manages the WebSocket lifecycle. It maintains separate state variables for protocol-specific metrics (TCP, UDP, IPv4, etc.) and history arrays for charting.

- **Delta Updates:** The hook processes update messages from the backend, appending new data points to historical arrays (metricsHistory) while maintaining a fixed window size (MAX_HISTORY_LENGTH = 30) to prevent memory leaks during long captures.

4.2.2 High-Performance Visualization:

- **Dynamic Scaling Charts:** The MetricChart.jsx component includes a utility getOptimalBitrateScale that automatically adjusts the Y-axis units (bps, Kbps, Mbps, Gbps) based on the current traffic volume, ensuring readability across different scales.
- **Sankey Diagrams:** Top talkers are visualized using TopTalkersSankey.jsx, which maps source IPs to destination IPs, with flow width proportional to data volume. This utilizes the react-google-charts library.

4.2.3 Virtualized Data Tables:

To render thousands of captured packets without freezing the browser, **PacketTable.jsx** implements **Virtualization** using @tanstack/react-virtual. This technique only renders the DOM elements currently visible in the viewport, significantly reducing the rendering workload and allowing for smooth scrolling even with datasets exceeding 10,000 packets.

4.3 AI Integration

The system integrates Generative AI (Google Gemini) to provide automated root cause analysis and health summaries.

4.3.1 Prompt Engineering & Data Formatting:

The llm_summarizer.py module acts as an adapter between the raw metric data and the LLM.

- **Data Aggregation:** It pre-calculates summary statistics (averages, peaks, totals) to create a concise JSON payload.
- **Structured Prompting:** The prompt explicitly instructs the AI to return a valid JSON object with specific keys (summary, breakdown, observations). This ensures the response can be programmatically parsed and rendered by the frontend, rather than just displaying unstructured text.

4.3.2 Periodic Summary Bot:

A dedicated background task, periodic_summary_loop in websocket_server.py, triggers the AI analysis every 60 seconds. It checks if the capture is active and if sufficient data has accumulated. The generated summary is then broadcast to all connected clients via a specific WebSocket event (type: "periodic_summary"), allowing the frontend "Bot" component to update users without manual intervention.

Chapter 5: Features & Modules

This project brings together a wide range of features designed to give users a clear and meaningful view of their network's behavior. It provides real-time dashboards, visual charts, and detailed packet information that make it easier to understand traffic patterns, spot unusual activity, and troubleshoot performance issues. The platform also includes interactive maps, protocol insights, and AI-generated summaries, helping users quickly identify trends and get practical recommendations. With flexible filtering and customizable views, it allows each user to focus on the information that matters most to them. Overall, the project offers an accessible yet powerful way to monitor, analyze, and optimize network performance.

5.1 Dashboard

5.1.1 Throughput, Goodput and Packets per Second (PPS) Monitoring

The Dashboard provides a real-time, high-level overview of network activity through intuitive charts and counters. It displays key metrics such as throughput, packet rates, and overall traffic trends, enabling users to quickly assess network health at a glance. With live updates and visual indicators, it helps identify spikes, drops, or abnormal patterns instantly. This module serves as the primary entry point for monitoring and ensures that critical insights are always visible and easy to interpret.

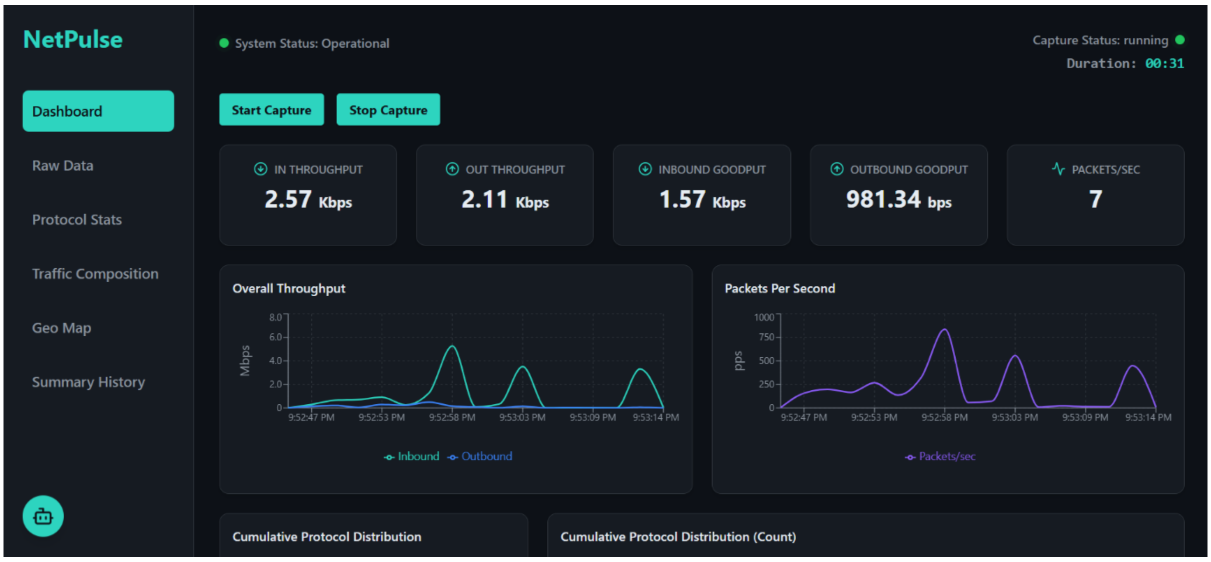


Figure 5.1 Real-Time Dashboard Interface

5.1.2 Protocol Distribution

Traffic across major protocols like TCP, UDP, RTP, DNS, and QUIC is represented through pie and bar charts, showing which protocols dominate network traffic. Simplifies multi-protocol analysis with visual clarity over tabular data.

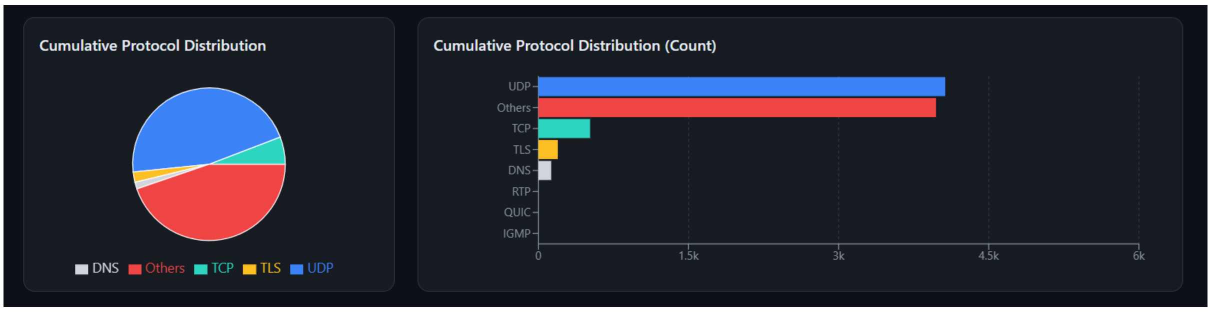


Figure 5.2 Protocol Distribution Charts

5.1.3 Top 7 Outbound Conversations (by Cumulative Volume)

This module visualizes traffic flows between source and destination endpoints, presenting a clear picture of how data moves across the network. The width of each flow indicates the relative traffic volume, helping users instantly identify heavy or dominant connections. By converting raw "top talkers" data into an intuitive visual map, it makes understanding network communications easier and more insightful.

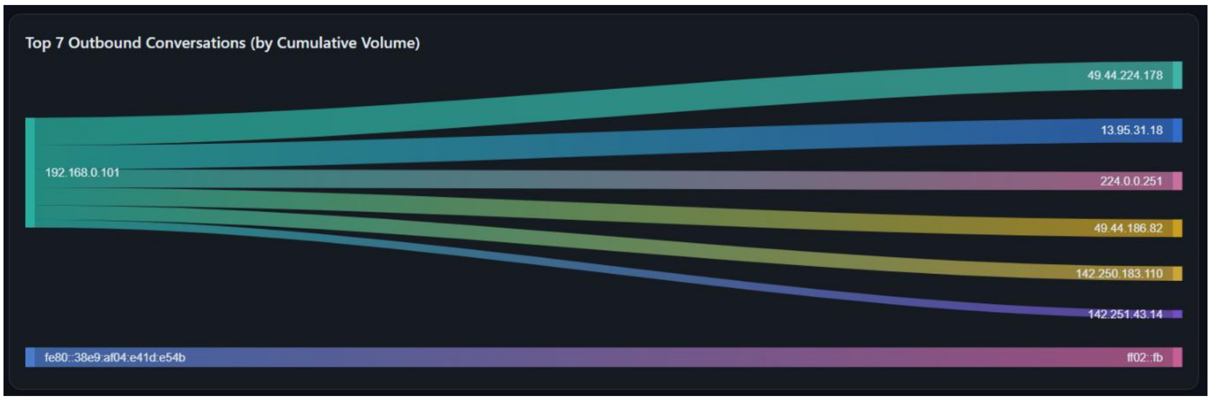
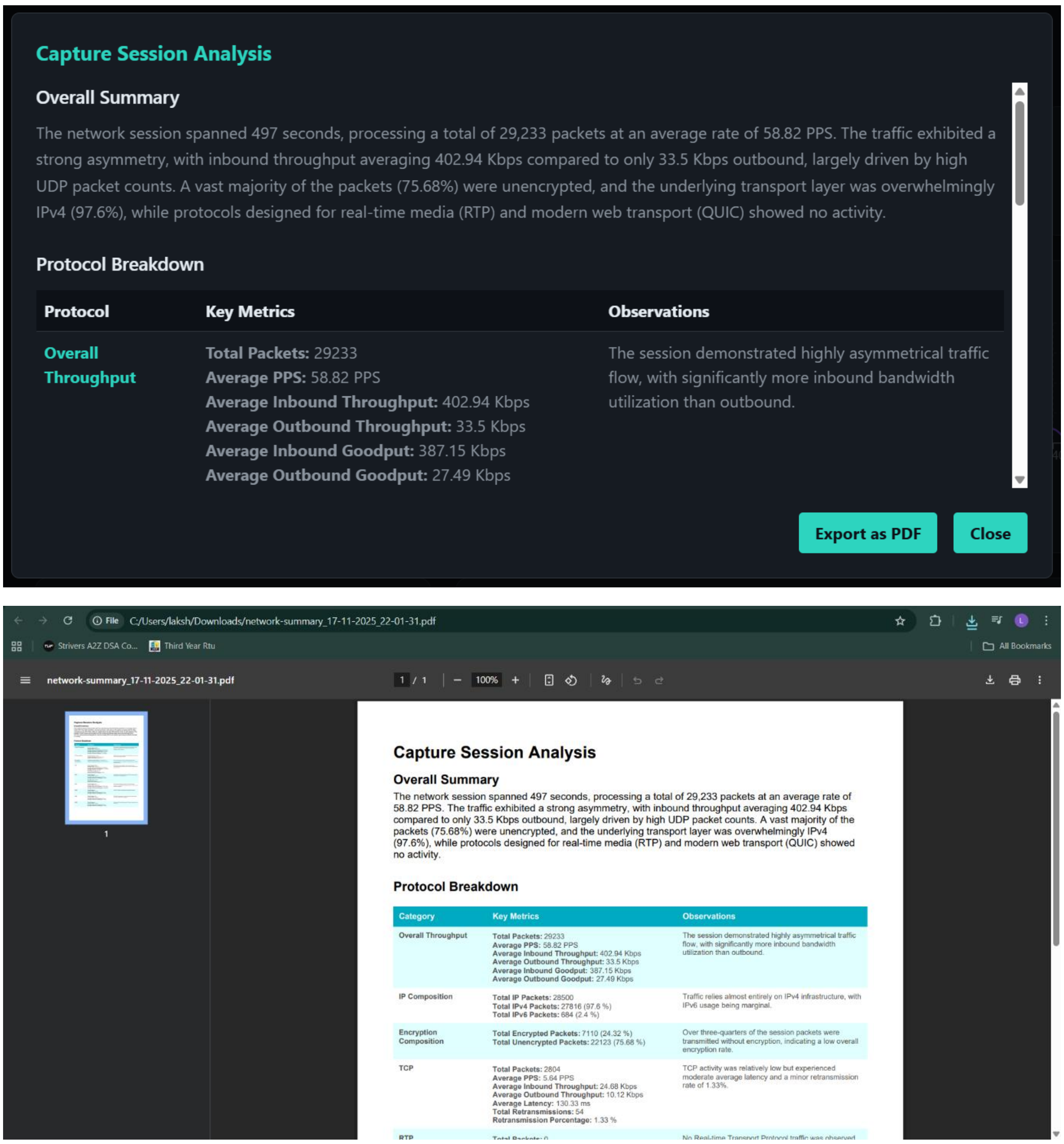


Figure 5.3 Top 7 Outbound Conversations

5.1.4 AI-Powered Network Health Summary

The AI-Powered Network Health Summary continuously analyzes live network metrics and generates a clear, human-readable overview of overall network health. It identifies trends, highlights anomalies, and provides actionable recommendations to improve performance. Additionally, users can export the AI-generated summary as a PDF, making it a convenient tool for reporting and documentation.



Capture Session Analysis

Overall Summary

The network session spanned 497 seconds, processing a total of 29,233 packets at an average rate of 58.82 PPS. The traffic exhibited a strong asymmetry, with inbound throughput averaging 402.94 Kbps compared to only 33.5 Kbps outbound, largely driven by high UDP packet counts. A vast majority of the packets (75.68%) were unencrypted, and the underlying transport layer was overwhelmingly IPv4 (97.6%), while protocols designed for real-time media (RTP) and modern web transport (QUIC) showed no activity.

Protocol Breakdown

Protocol	Key Metrics	Observations
Overall Throughput	**Total Packets:** 29233 **Average PPS:** 58.82 PPS **Average Inbound Throughput:** 402.94 Kbps **Average Outbound Throughput:** 33.5 Kbps **Average Inbound Goodput:** 387.15 Kbps **Average Outbound Goodput:** 27.49 Kbps	The session demonstrated highly asymmetrical traffic flow, with significantly more inbound bandwidth utilization than outbound.

Export as PDF
Close
File
C:/Users/laksh/Downloads/network-summary_17-11-2025_22-01-31.pdf
Strivers A2Z DSA Co...
Third Year Rtu
All Bookmarks
network-summary_17-11-2025_22-01-31.pdf
1 / 1
100%
1

Capture Session Analysis

Overall Summary

The network session spanned 497 seconds, processing a total of 29,233 packets at an average rate of 58.82 PPS. The traffic exhibited a strong asymmetry, with inbound throughput averaging 402.94 Kbps compared to only 33.5 Kbps outbound, largely driven by high UDP packet counts. A vast majority of the packets (75.68%) were unencrypted, and the underlying transport layer was overwhelmingly IPv4 (97.6%), while protocols designed for real-time media (RTP) and modern web transport (QUIC) showed no activity.

Protocol Breakdown

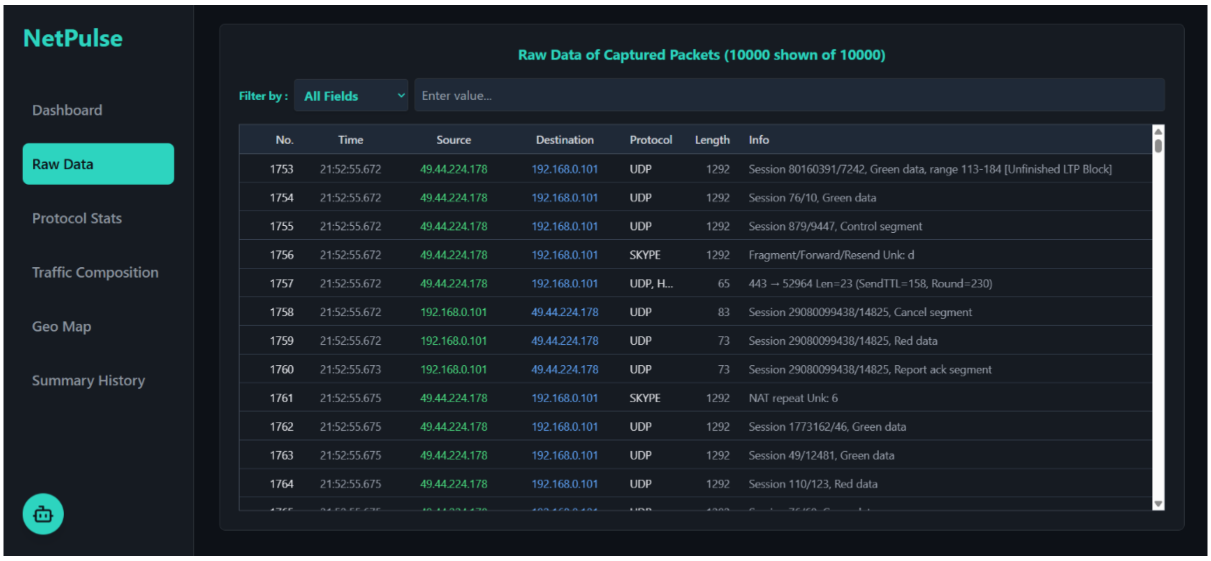
Category	Key Metrics	Observations
Overall Throughput	**Total Packets:** 29233 **Average PPS:** 58.82 PPS **Average Inbound Throughput:** 402.94 Kbps **Average Outbound Throughput:** 33.5 Kbps **Average Inbound Goodput:** 387.15 Kbps **Average Outbound Goodput:** 27.49 Kbps	The session demonstrated highly asymmetrical traffic flow, with significantly more inbound bandwidth utilization than outbound.
IP Composition	**Total IP Packets:** 28500 **Total IPv4 Packets:** 27816 (97.6 %) **Total IPv6 Packets:** 684 (2.4 %)	Traffic relies almost entirely on IPv4 infrastructure, with IPv6 usage being marginal.
Encryption Composition	**Total Encrypted Packets:** 7110 (24.32 %) **Total Unencrypted Packets:** 22123 (75.68 %)	Over three-quarters of the session packets were transmitted without encryption, indicating a low overall encryption rate.
TCP	**Total Packets:** 2804 **Average PPS:** 5.64 PPS **Average Inbound Throughput:** 24.68 Kbps **Average Outbound Throughput:** 10.12 Kbps **Average Latency:** 130.33 ms **Total Retransmissions:** 54 **Retransmission Percentage:** 1.33 %	TCP activity was relatively low but experienced moderate average latency and a minor retransmission rate of 1.33%.
RTP	**Total Packets:** 0	No Real-time Transport Protocol traffic was observed

Figure 5.4 AI-Powered Network Health Summary & PDF

5.2 Raw Data

5.2.1 Raw Data Table

This module displays all captured packets in a clear and user-friendly format, presenting essential details such as timestamps, IP addresses, protocols, and packet sizes for easy interpretation. It also includes advanced filtering options that allow users to quickly narrow down results by specific IP addresses or protocols, making it more efficient to analyze targeted traffic and troubleshoot network issues.



Raw Data of Captured Packets (10000 shown of 10000)

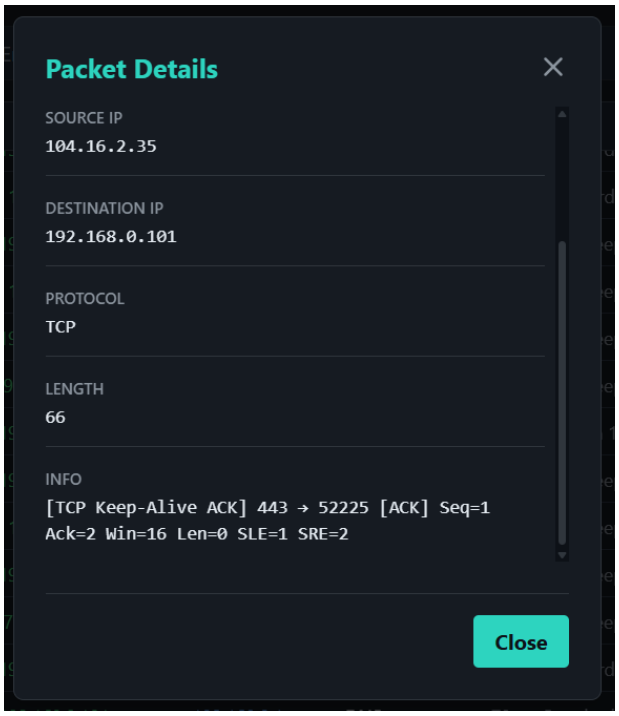
Filter by : All Fields | Enter value...

No.	Time	Source	Destination	Protocol	Length	Info
1753	21:52:55.672	49.44.224.178	192.168.0.101	UDP	1292	Session 80160391/7242, Green data, range 113-184 [Unfinished LTP Block]
1754	21:52:55.672	49.44.224.178	192.168.0.101	UDP	1292	Session 76/10, Green data
1755	21:52:55.672	49.44.224.178	192.168.0.101	UDP	1292	Session 879/9447, Control segment
1756	21:52:55.672	49.44.224.178	192.168.0.101	SKYPE	1292	Fragment/Forward/Resend Unk: d
1757	21:52:55.672	49.44.224.178	192.168.0.101	UDP, H...	65	443 → 52964 Len=23 (SendTTL=158, Round=230)
1758	21:52:55.672	192.168.0.101	49.44.224.178	UDP	83	Session 29080099438/14825, Cancel segment
1759	21:52:55.672	192.168.0.101	49.44.224.178	UDP	73	Session 29080099438/14825, Red data
1760	21:52:55.673	192.168.0.101	49.44.224.178	UDP	73	Session 29080099438/14825, Report ack segment
1761	21:52:55.675	49.44.224.178	192.168.0.101	SKYPE	1292	NAT repeat Unk: 6
1762	21:52:55.675	49.44.224.178	192.168.0.101	UDP	1292	Session 1773162/46, Green data
1763	21:52:55.675	49.44.224.178	192.168.0.101	UDP	1292	Session 49/12481, Green data
1764	21:52:55.675	49.44.224.178	192.168.0.101	UDP	1292	Session 110/123, Red data

Figure 5.5 Raw Data Table Interface

5.2.2 Individual Packet Details

The Individual Packet Display shows detailed information for every captured packet, including source IP, destination IP, protocol, length, and additional packet insights.



Packet Details

SOURCE IP
104.16.2.35

DESTINATION IP
192.168.0.101

PROTOCOL
TCP

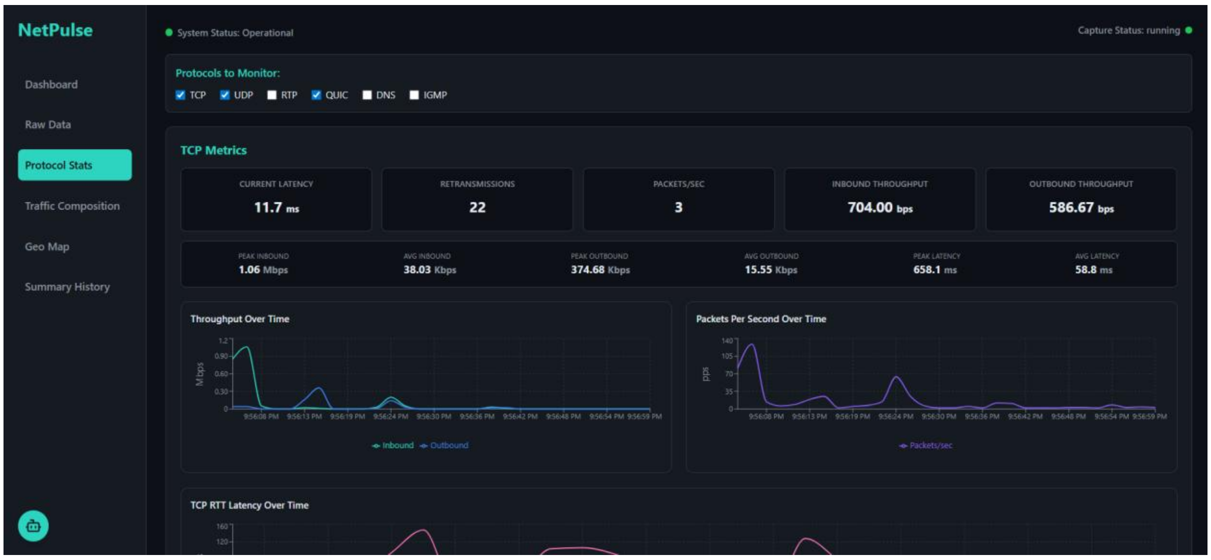
LENGTH
66

INFO
[TCP Keep-Alive ACK] 443 → 52225 [ACK] Seq=1 Ack=2 Win=16 Len=0 SLE=1 SRE=2

Figure 5.6 Individual Packet Details Modal

5.3 Protocol Stats

The Protocol Analysis module breaks down network traffic by protocol, showing how TCP, UDP, RTP, DNS, QUIC, and others contribute to overall usage. Using pie charts and bar graphs, it helps users identify dominant protocols, spot imbalances, and detect unusual behavior. By examining protocol-specific metrics like latency, loss, and throughput, users can efficiently understand performance and troubleshoot issues.



System Status: Operational

Capture Status: running

Protocols to Monitor:

[✓ TCP] [✓ UDP] RTP [✓ QUIC] DNS IGMP

TCP Metrics

CURRENT LATENCY	RETRANSMISSIONS	PACKETS/SEC	INBOUND THROUGHPUT	OUTBOUND THROUGHPUT
11.7 ms	22	3	704.00 bps	586.67 bps

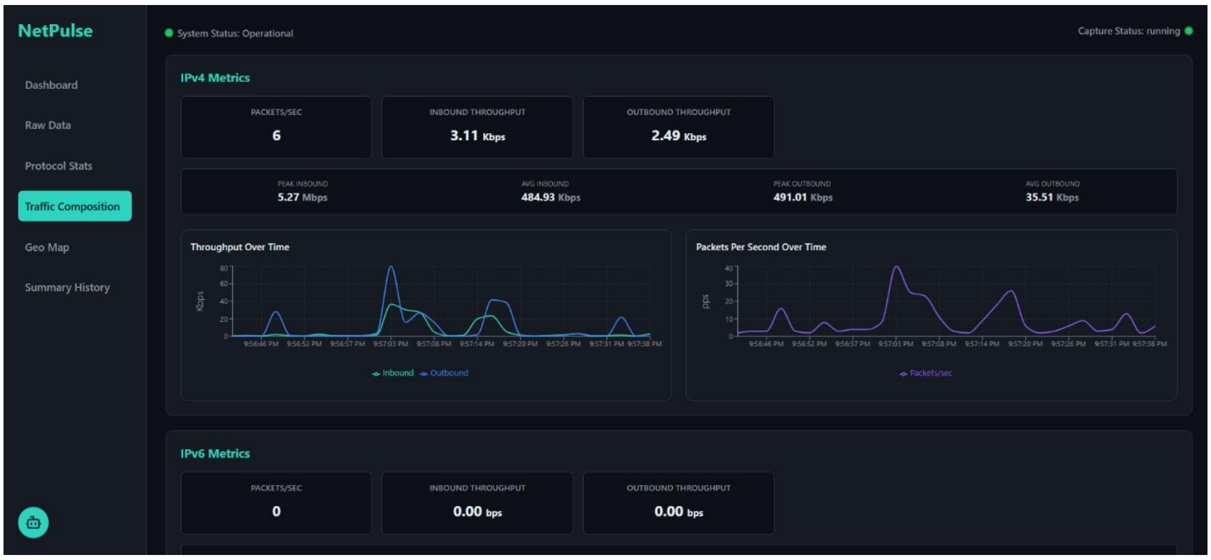
PEAK INBOUND	AVG INBOUND	PEAK OUTBOUND	AVG OUTBOUND	PEAK LATENCY	AVG LATENCY
1.06 Mbps	38.03 Kbps	374.68 Kbps	15.55 Kbps	658.1 ms	58.8 ms

Figure 5.7 Protocol Statistics Interface

5.4 Traffic Composition

5.4.1 IPv4 and IPv6 Composition

IPv4 and IPv6 traffic is separated and displayed through a pie chart, showing the composition of network flows. Line charts present packet rates and throughput for each IP version. This helps monitor traffic patterns, assess readiness for IPv6 adoption, and plan for modern, future-ready network infrastructure.



System Status: Operational

Capture Status: running

IPv4 Metrics

PACKETS/SEC	INBOUND THROUGHPUT	OUTBOUND THROUGHPUT
6	3.11 Kbps	2.49 Kbps

PEAK INBOUND	AVG INBOUND	PEAK OUTBOUND	AVG OUTBOUND
5.27 Mbps	484.93 Kbps	491.01 Kbps	35.51 Kbps

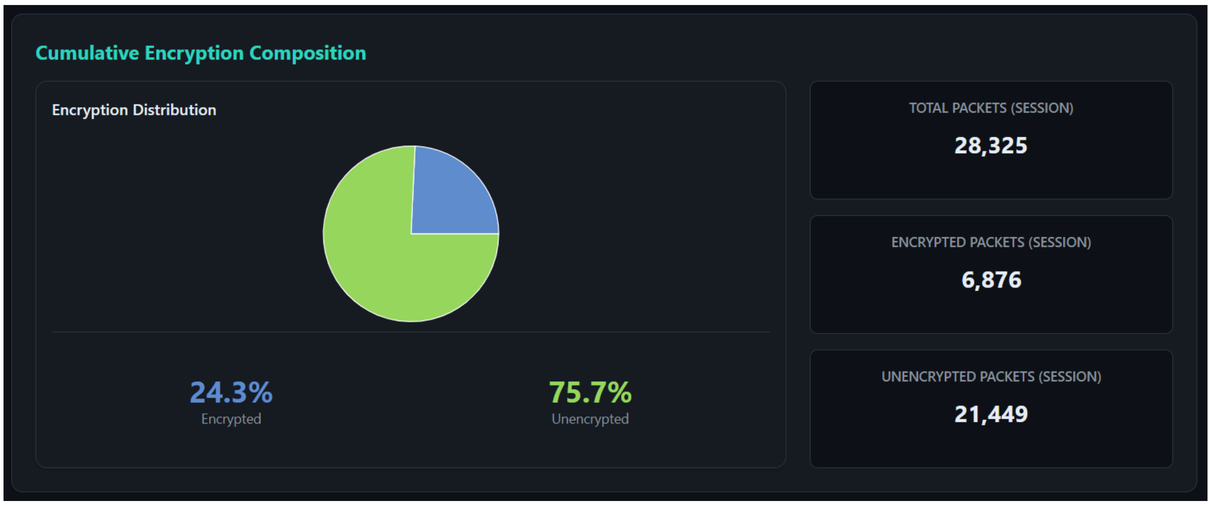
IPv6 Metrics

PACKETS/SEC	INBOUND THROUGHPUT	OUTBOUND THROUGHPUT
0	0.00 bps	0.00 bps

Figure 5.8 Traffic Composition Interface

5.4.2 Encryption Composition

Encryption and Security Analysis visualizes the proportion of encrypted versus unencrypted traffic using a clear pie chart, providing immediate insight into the network's security posture. It highlights potential vulnerabilities in unprotected communications, allowing users to quickly identify areas of concern. This feature offers instant visual cues that help monitor and strengthen overall network security.



Cumulative Encryption Composition

Encryption Distribution

24.3% Encrypted

75.7% Unencrypted

TOTAL PACKETS (SESSION) 28,325

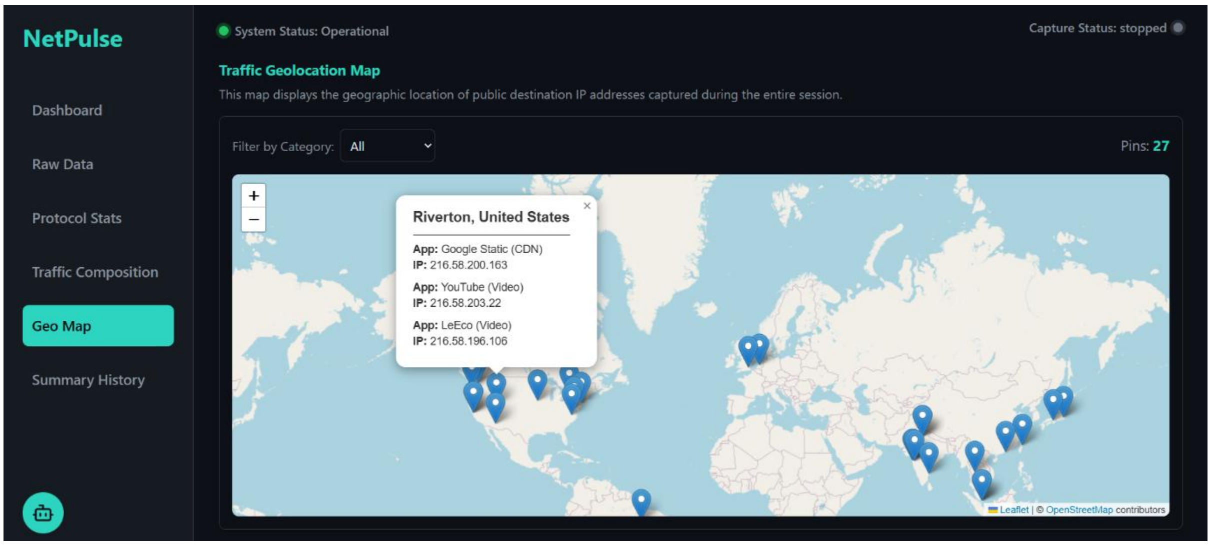
ENCRYPTED PACKETS (SESSION) 6,876

UNENCRYPTED PACKETS (SESSION) 21,449

Figure 5.9 Encryption Composition Pie Chart

5.5 Geo Map

The Geo-Map module visualizes network traffic geographically, mapping source and destination endpoints across the globe. Through an interactive world map, users can track where traffic is coming from, where it is going, and identify unusual or unexpected regions of activity. This feature is especially valuable for detecting suspicious traffic patterns, monitoring global application usage, and enhancing situational awareness of network behavior in a geographic context.



System Status: Operational

Capture Status: stopped

Traffic Geolocation Map

This map displays the geographic location of public destination IP addresses captured during the entire session.

Filter by Category: All

Pins: 27

Riverton, United States

App: Google Static (CDN)
IP: 216.58.200.163

App: YouTube (Video)
IP: 216.58.203.22

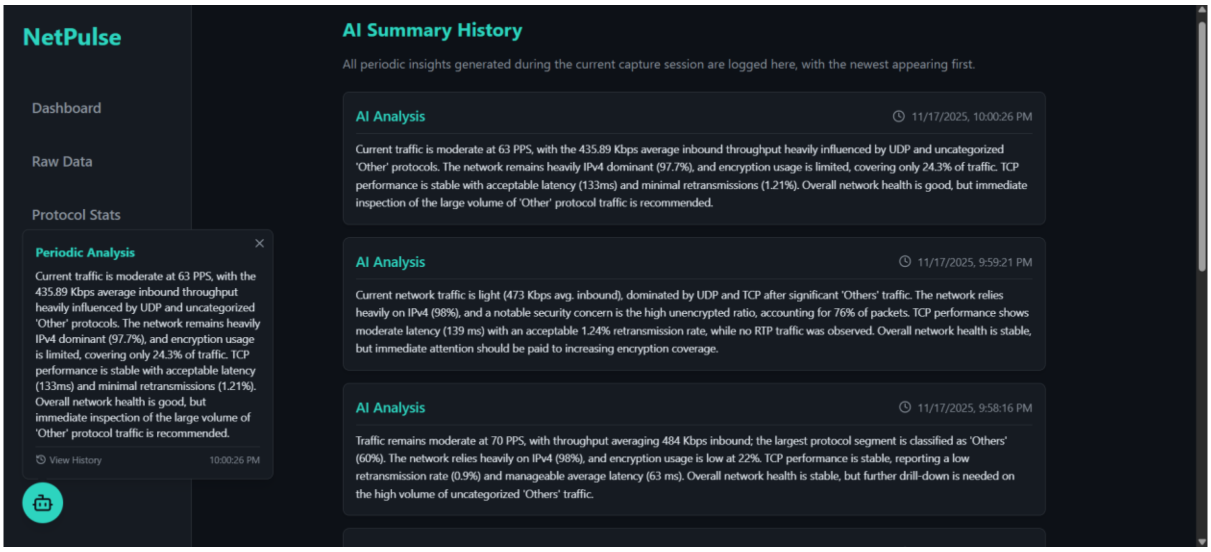
App: LeEco (Video)
IP: 216.58.196.106

Figure 5.10 Traffic Geolocation Map

5.6 AI Summary Bot and Summary History

The AI Summary Bot provides automated insights by analyzing recent network data to generate concise summaries, trend evaluations, and actionable recommendations. It highlights anomalies, performance degradations, and significant traffic shifts, helping users understand complex metrics without deep technical analysis.

The system displays a history of AI-generated network summaries, allowing users to review past reports and analyze network performance at specific timestamps. This enables tracking of trends and identifying patterns over time. It provides a convenient way to monitor network health.



AI Summary History

All periodic insights generated during the current capture session are logged here, with the newest appearing first.

AI Analysis — 11/17/2025, 10:00:26 PM

Current traffic is moderate at 63 PPS, with the 435.89 Kbps average inbound throughput heavily influenced by UDP and uncategorized 'Other' protocols. The network remains heavily IPv4 dominant (97.7%), and encryption usage is limited, covering only 24.3% of traffic. TCP performance is stable with acceptable latency (133ms) and minimal retransmissions (1.21%). Overall network health is good, but immediate inspection of the large volume of 'Other' protocol traffic is recommended.

AI Analysis — 11/17/2025, 9:59:21 PM

Current network traffic is light (473 Kbps avg. inbound), dominated by UDP and TCP after significant 'Others' traffic. The network relies heavily on IPv4 (98%), and a notable security concern is the high unencrypted ratio, accounting for 76% of packets. TCP performance shows moderate latency (139 ms) with an acceptable 1.24% retransmission rate, while no RTP traffic was observed. Overall network health is stable, but immediate attention should be paid to increasing encryption coverage.

AI Analysis — 11/17/2025, 9:58:16 PM

Traffic remains moderate at 70 PPS, with throughput averaging 484 Kbps inbound; the largest protocol segment is classified as 'Others' (60%). The network relies heavily on IPv4 (98%), and encryption usage is low at 22%. TCP performance is stable, reporting a low retransmission rate (0.9%) and manageable average latency (63 ms). Overall network health is stable, but further drill-down is needed on the high volume of uncategorized 'Others' traffic.

Periodic Analysis

Current traffic is moderate at 63 PPS, with the 435.89 Kbps average inbound throughput heavily influenced by UDP and uncategorized 'Other' protocols. The network remains heavily IPv4 dominant (97.7%), and encryption usage is limited, covering only 24.3% of traffic. TCP performance is stable with acceptable latency (133ms) and minimal retransmissions (1.21%). Overall network health is good, but immediate inspection of the large volume of 'Other' protocol traffic is recommended.

10:00:26 PM

Figure 5.11 AI Summary History Log

Chapter- 6 Performance Evaluation (Testing)

This chapter presents the evaluation of NetPulse under different traffic conditions to determine its packet processing limits, throughput handling capability, and overall reliability. Wireshark was used as the baseline reference, as it represents an optimized and industry-standard packet capture tool. All tests were performed to compare NetPulse's backend performance against Wireshark and to identify potential bottlenecks related to stdout buffering and parsing speed.

6.1 Test Environment

All performance evaluations for NetPulse were conducted in a controlled and uniform environment to ensure accuracy and reproducibility. The tests were performed on a system equipped with an Intel Core i5-1135G7 processor, 16 GB of RAM, and a 512 GB SSD, running Windows 11. The backend relied on Python 3.13.5 and TShark 4.4.7 for packet capture, with metric calculations executed at one-second intervals. This environment ensured that hardware limitations did not influence the test outcomes, allowing NetPulse's actual processing capability to be measured with minimal external interference.

6.2 Test Methodology

The primary objective of the testing phase was to determine the maximum processing thresholds before the standard output (stdout) buffer blocked, which would result in packet drops.
Two distinct stress-test scenarios were designed:

- **High Packet Rate (Small Packets):** This scenario focused on stressing the Python parsing logic by generating a high volume of small packets. Traffic was generated by opening 10-15 browser tabs simultaneously, loading high-traffic sites (YouTube, social media), and initiating Google Meet sessions to simulate real-time UDP streams.

- **High Throughput (Large Packets):** This scenario tested the system's ability to handle high data volume (bandwidth). Sustained traffic was generated by initiating multiple simultaneous large file downloads (500MB–1GB each) to saturate the network interface throughput.

These tests aimed to evaluate whether NetPulse could maintain accurate throughput computations and packet consistency under prolonged heavy load. In all cases, NetPulse and Wireshark were started simultaneously, traffic was generated for a controlled duration, and both captures were stopped synchronously to enable precise packet-count comparisons.

6.3 Results

- **High Packet Rate Tests:** NetPulse demonstrated strong accuracy under dense traffic conditions, achieving peak capture rates of 13,633 PPS and 14,877 PPS across the two high-rate evaluations. In Test #1, Wireshark captured 1,663,858 packets compared to NetPulse's 1,663,111, yielding a deviation of just 0.045%. Test #3 showed even closer alignment, where Wireshark recorded 1,160,821 packets and NetPulse captured 1,160,861, reflecting an exceptionally low deviation of 0.003%. Despite Wireshark reporting minor kernel drops during these tests, NetPulse remained fully stable and responsive throughout, confirming its capability to handle high PPS environments with minimal loss.

- **High Throughput Tests:** NetPulse also performed reliably under sustained large-packet data transfers. In Test #2, Wireshark captured 799,979 packets while NetPulse captured 801,997, resulting in a negligible deviation of 0.25%. Test #4 showed near-perfect accuracy, with Wireshark capturing 786,706 packets and NetPulse recording 786,697, a deviation of only 0.001%. Throughput measurements remained consistent with actual network speeds, reaching 37.14 Mbps and 35.4 Mbps, and no packet drops or processing backlogs were observed. These results confirm NetPulse's stability and precision when operating under high-volume, sustained throughput conditions.

6.4 Analysis

The combined results demonstrate that NetPulse delivers highly reliable and data-accurate performance across both high packet rate and high throughput conditions. Packet count deviations across all tests ranged between 0.001% and 0.25%, confirming that NetPulse maintains near-benchmark alignment with Wireshark, even under demanding traffic loads. Throughout the testing process, NetPulse processed over five million packets without experiencing buffering issues, metric calculation delays, or system instability. Its ability to sustain peak performance of up to 14,877 PPS in high packet rate scenarios, along with accurate throughput computation above 35 Mbps in large-packet scenarios, reflects the strength of its asynchronous processing pipeline. These data-driven results demonstrate that NetPulse is sufficiently robust and scalable for real-time network monitoring applications and can reliably interpret traffic patterns with minimal deviation from industry-standard tools.

Chapter 7: User Guide / Setup

This section provides step-by-step instructions to set up and run **NetPulse**, ensuring that users can install dependencies, configure the environment, and launch the application for real-time network monitoring and analysis.

For Windows / MacOS:

7.1 Prerequisites

For Windows:

Before launching the NetPulse executable on Windows, ensure the following external dependencies are installed on your system:

1. **Wireshark (includes TShark):** Mandatory for packet capturing and system-wide traffic visibility. Ensure that the **TShark** command-line utility is available in your system **PATH**.

2. **Npcap (Windows):** The packet capture driver required for Windows systems to access the host's Network Interface Card (NIC) in promiscuous mode.

3. **Modern Web Browser:** Required to render the dashboard UI (Chrome, Edge, Firefox, etc.).

For MacOS:

Before launching the NetPulse executable on macOS, you must ensure your system environment is correctly configured according to these specific requirements:

- **Official Wireshark Installation**: Wireshark must be installed directly from the **official website**

- **Unsupported Versions**: Installations performed via **Homebrew are NOT supported** for this application.

- **ChmodBPF Configuration**: During the Wireshark installation process, you must enable the option **"Install ChmodBPF (Allow non-root users to capture packets)"**. This is a one-time setup that requires your system password and allows the application to capture traffic without requiring root privileges for every session.

- **TShark Availability**: The **TShark** command-line utility must be available. The application automatically searches for TShark in the following default macOS locations if it is not found in your system PATH:
    - /Applications/Wireshark.app/Contents/MacOS/tshark

    - /usr/local/bin/tshark
    - /opt/homebrew/bin/tshark

- **Web Browser**: Any modern web browser is required to view the dashboard, including **Safari, Chrome, Edge, or Firefox**.

7.2 Installation Steps

1. **Download file from GitHub Releases:**
    - Navigate to the **Releases** section of the Network-Analysis-Dashboard-HPE repository.
    - Download the asset corresponding to your Operating System:
        - **Windows:** main.exe.
        - **macOS:** macOS.app bundle.

2. **Placement:** Move the downloaded file to a preferred directory. Ensure you have write permissions, as the application extracts temporary assets at runtime.

7.3 Configuration

- **Administrative Privileges:** You **must** run the executable as an **Administrator** (Windows) or with **root privileges** (macOS). This is mandatory for accessing host network interfaces.

- **System PATH:** Verify that TShark is accessible. The application will perform an automatic validation check (shutil.which("tshark")) upon startup.
- **Dependency Validation:** If TShark is not detected, a custom popup will appear providing an option to redirect to the official Wireshark download page.

7.4 How to Run

1. Launch the Application

Double-click the main.exe or .app file. The application is intentionally terminal-based to provide transparency during execution and allow users to view runtime logs directly.

2. Startup Flow

Upon launch, the application executes the following sequence:

- **Backend Initialization:** The Python backend starts an embedded HTTP server to serve the React frontend and a WebSocket server for live data streaming.

- **Informational Notification:** A startup popup explains the requirements and redirects the user to their browser.

- **Capture Initialization:** The application identifies available network interfaces and initiates the TShark packet capture process.

3. Accessing the Dashboard

The dashboard is automatically served locally:

- **URL:** http://localhost:8000.

- **Real-time Interaction:** The browser loads the UI from localhost, behaving like a hosted web app while maintaining full system-wide traffic visibility.

The terminal window provides immediate feedback on backend status, WebSocket connections, and real-time packet processing metrics.

For Linux Based Systems:

7.5 Prerequisites

Before deploying the NetPulse container, ensure your Linux system has the following installed:

- **Docker Engine:** Required to pull and execute the NetPulse image.

- **Docker Compose:** Necessary to process the docker-compose.yml orchestration file.

- **Sudo/Root Privileges:** Required to allow the containerized **TShark** process to access host network interfaces.

- **Gemini API Key:** Required for the AI-generated network summaries feature.

7.6 Installation Steps

The Docker deployment replaces manual environment setup by using versioned configuration files provided in our GitHub Releases.

1. **Download the Configuration:**
   - Navigate to the **Releases** section of the Network-Analysis-Dashboard-HPE repository.
   - Download the docker-compose.yml file from the latest release.

2. **Pull Images from Docker Hub:**
   - The docker-compose.yml is pre-configured to point to the official images.
   - Open a terminal in the folder containing the file and run:

```
sudo docker-compose pull
```

   - This command will simultaneously pull the NetPulse Backend (Python/TShark engine) and the NetPulse Frontend (React Dashboard) images from Docker Hub

3. **Directory Setup:**
   - Place the docker-compose.yml file in a dedicated folder (e.g., ~/netpulse/).
   - Open a terminal and navigate to that folder: cd ~/netpulse/.

7.7 Configuration

Before running the application, you must configure your environment variables within the docker-compose.yml file:

- **Gemini API Key:** Open the docker-compose.yml file and locate the environment: section. Put GEMINI_API_KEY=<your_api_key> with your actual key to enable AI summaries.
- **Host Networking Mode:** The configuration is set to network_mode: host to ensure the container can see actual host traffic rather than isolated container traffic.
- **Privileged Execution:** The container runs in privileged: true mode to grant the backend the kernel-level permissions needed for raw packet capture.

7.8 How to Run

1. Launch the Stack

Start the NetPulse environment by running:

```
sudo docker-compose up -d
```

The -d flag runs the application in the background. To view real-time logs for debugging or to monitor the packet capture lifecycle, use docker logs -f [container_name].

2. Startup Validation

Once launched, the container performs these internal steps:

- **Server Initialization:** Starts the embedded Python HTTP server and WebSocket server for live data.
- **TShark Hooking:** The backend automatically identifies host interfaces and begins the packet capture process.
- **AI Readiness:** Verifies the Gemini API connection for real-time traffic summaries.

3. Accessing NetPulse

Open your browser and navigate to the following address:

- **Local URL:** http://localhost:3000

Chapter 8: Conclusion & Future Scope

8.1 Conclusion

NetPulse represents a modern, comprehensive approach to network monitoring, integrating detailed packet-level inspection with high-level traffic and protocol visualization into a single, cohesive platform. As a web-based solution, it enables users to monitor, analyze, and understand complex network behavior in real time, all through an intuitive and interactive interface.

By combining live metrics, visual analytics, and AI-powered summaries, **NetPulse** bridges the gap between traditional network analysis tools and the growing need for intelligent, actionable insights. It builds upon the strengths of industry-standard tools like Wireshark, known for precise packet-level analysis, and ntopng, recognized for robust flow visualization, while enhancing usability, interactivity, and automation for modern network management.

NetPulse effectively unifies multiple functionalities into one platform:

- **Packet-Level Inspection**: Displays detailed information on individual packets, including source and destination IPs, protocols, timestamps, and sizes, enabling deep traffic analysis.
- **Protocol and Traffic Analysis**: Breaks down network traffic by protocols and IP versions, visualizing trends, dominant flows, and performance metrics to highlight anomalies or inefficiencies.
- **Flow and Communication Visualization**: Uses interactive Sankey diagrams and geolocation maps to illustrate traffic flows, top talkers, and global communication patterns, offering an intuitive understanding of network behavior.
- **AI-Powered Network Summaries**: Continuously reviews live network metrics to generate human-readable reports, detect anomalies, identify trends, and provide actionable recommendations, supporting proactive network management.
- **Customizable and Interactive Interface**: Offers advanced filtering, sorting, and personalized views, allowing users to focus on the information that matters most to their specific needs.

The key benefits of **NetPulse** include real-time monitoring with minimal latency, a visually rich and highly interactive interface, and the unique capability to produce intelligent AI-driven summaries. It provides administrators with clear insights into network performance, security posture, traffic composition, and protocol behavior, helping in troubleshooting, capacity planning, and future infrastructure readiness.

Overall, **NetPulse** demonstrates how modern network monitoring can be transformed by integrating real-time analytics, interactive visualizations, and AI intelligence. It empowers organizations and network administrators to understand, manage, and optimize complex networks efficiently, offering clarity, precision, and actionable insights—all from a single, accessible platform.

8.2 Limitations

While **NetPulse** provides a comprehensive and modern platform for network monitoring, it does have certain limitations that users should be aware of. These constraints are primarily related to performance, functionality, and deployment scenarios:

- **Dependency on Local Environment**: NetPulse requires Python, Node.js, and Wireshark (tshark) to be installed and properly configured on the user's system. Issues in installation or environment setup can prevent the application from running smoothly.
- **Real-Time Performance Constraints**: Capturing and processing high volumes of network traffic in real time may strain system resources, especially on low-end machines, potentially leading to delayed metrics or incomplete packet captures.
- **Interface-Based Summary Display**: Currently, NetPulse displays AI-generated network summaries and traffic data directly on the interface, without storing long-term historical data in a database. This limits extended trend analysis or reporting.
- **Browser-Based Interface Limitations**: The dashboard relies on a web-based interface, which may have performance issues when handling extremely large datasets or visualizations with thousands of network flows simultaneously.
- **Protocol Support Constraints**: While NetPulse supports major protocols like TCP, UDP, RTP, DNS, and QUIC, support for some emerging or proprietary protocols may be limited.
- **Security and Permissions**: Capturing network traffic requires appropriate permissions or administrative access, which may restrict usability in certain corporate or restricted environments.

Despite these limitations, **NetPulse** remains a powerful tool for network monitoring, analysis, and visualization. Recognizing these constraints helps users make informed decisions and highlights areas for future improvement and feature enhancement.

8.3 Future Scope

NetPulse provides a robust foundation for network monitoring, and there are numerous opportunities to expand its functionality and adapt it to evolving network requirements. The future development of the platform could focus on several key areas:

1. **Advanced Traffic Categorization and Application Intelligence**

   Future versions of NetPulse can incorporate deep application-based traffic classification, enabling the system to identify which services and platforms generate network activity. By mapping flows to specific applications and presenting aggregated usage insights, such as top application contributors and smart structured raw data table, NetPulse can move beyond packet-level visibility and provide meaningful, service-oriented analytics.

2. **AI-Powered Network Assistant**

   An integrated AI chatbot can further enhance usability by allowing users to query the system through natural language. This assistant would answer questions related to network health, traffic anomalies, protocol behaviors, and dashboard navigation. Such conversational intelligence democratizes network analytics, enabling even non-experts to extract insights on demand.

3. **Automated Alerting and Anomaly Detection**

   NetPulse can incorporate **customizable threshold-based alerts** and predefined mitigation strategies triggered by deviations from typical network behavior or when specific user-defined limits are crossed. Administrators could configure thresholds for metrics such as latency, packet loss, throughput, or jitter, enabling the system to generate alerts the moment these values exceed acceptable bounds. Such intelligent alerting can be paired with automated responses that isolate suspicious flows, reroute traffic to balance load, or send notifications to administrators. These capabilities would reduce downtime, minimize manual monitoring efforts, and move the system toward proactive, self-managed network operations.

4. **Historical Data Storage and Trend Analysis**

   Introducing data persistence through time-series databases would allow NetPulse to retain historical metrics, enabling users to track changes over time, compare traffic patterns across periods, validate compliance, and perform deeper forensic analysis, capabilities currently limited to real-time snapshots.

5. **Enhanced Visualization Capabilities**
   Future enhancements may include interactive 3D topology maps, Sankey flow diagrams, and geospatial overlays to present traffic and performance metrics more intuitively. These advanced visualizations can significantly improve network comprehension and fault diagnosis.

By pursuing these advancements, NetPulse can evolve from a real-time monitoring dashboard into a predictive, intelligent, and self-sustaining observability platform capable of understanding network behavior, anticipating issues, and guiding operational decisions.