Native Executable Creation for Windows and macOS

1. Overview

This document describes the approach used to package the **Network Monitoring Dashboard (HPE Project)** into **native executable applications** for **Windows and macOS**.

The objective was to deliver a **single-click, dependency-free application** that:

- Runs directly on the host operating system
- Captures real network traffic using system interfaces
- Does not require Python or Node.js installation on the user machine
- Starts both frontend and backend together

Linux deployment is handled separately using Docker and is **out of scope for this document**.

2. Why Native Executables Were Chosen

2.1 Limitations of Containers on Windows/macOS

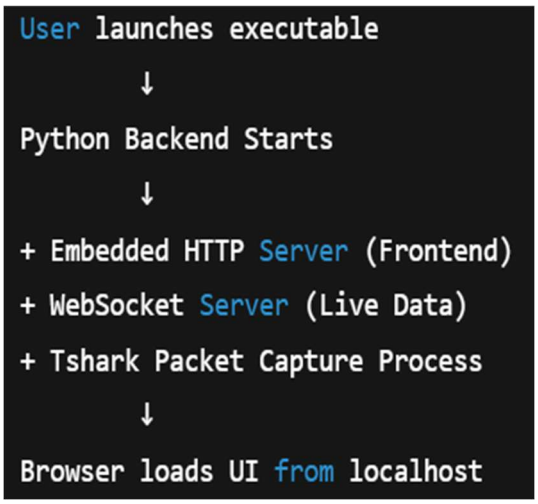
- Docker on Windows/macOS runs inside a **virtual machine**
- VM network adapters cannot access the host's NIC in promiscuous mode
- Packet capture tools (tshark) inside containers only see container traffic
- This makes accurate live packet capture impossible

2.2 Advantages of Native Execution

- Direct access to host network interfaces
- Full compatibility with Npcap (Windows) and OS networking stack
- No virtualization overhead
- Accurate system-wide traffic visibility

Conclusion:
Native executables are mandatory for reliable packet capture on Windows and macOS.

3. High-Level Architecture



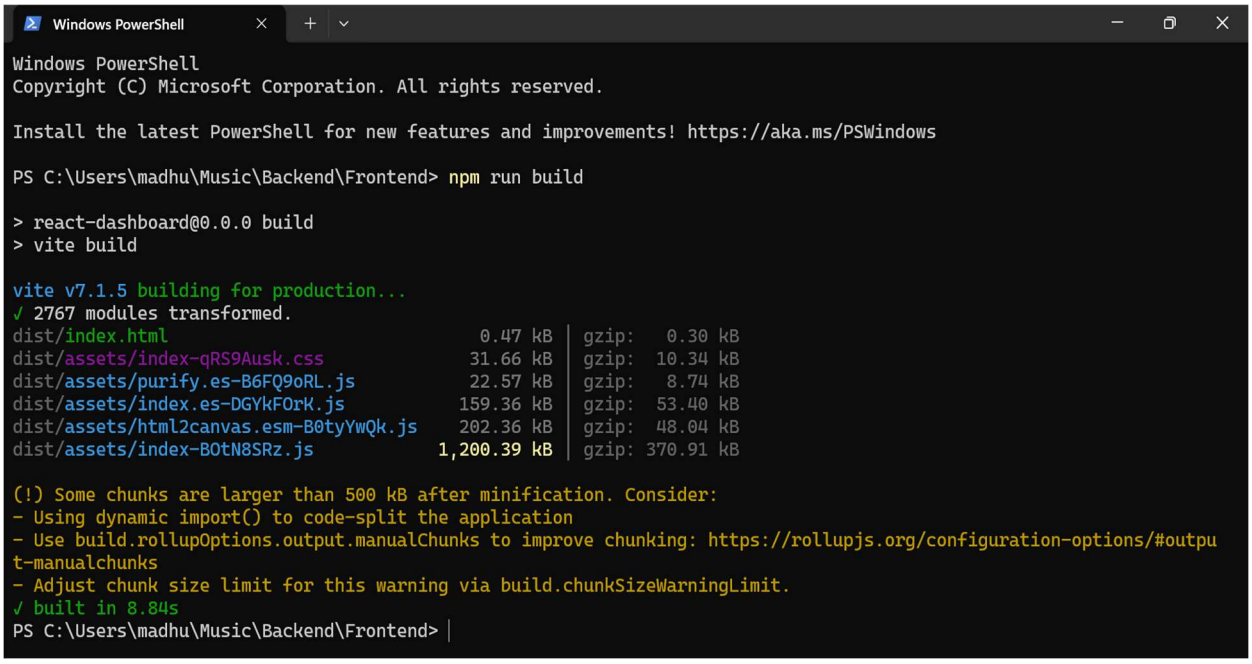
Frontend and backend are bundled and launched together as a single application.

4. Frontend Packaging Strategy

4.1 Why Dynamic Frontend Serving Was Avoided

Dynamic frontend servers (Vite / Node.js) are designed for development and introduce:

- Runtime dependency on Node.js
- Additional background processes
- Complexity in executable packaging
- Increased memory usage



4.2 Static Build Approach

The frontend is built using:

npm run build

This produces a dist/ directory containing:

- index.html
- Minified JavaScript bundles
- Optimized CSS
- Static assets (images, icons)

After build:

- The frontend no longer requires Node.js
- React runs entirely in the browser

- Files can be served by any HTTP server

4.3 Embedding Frontend into Executable

- The dist/ folder is bundled inside the executable
- A Python HTTP server serves these files locally at:
- http://localhost:8000
- The frontend behaves like a hosted web app, but runs fully offline

5. Backend Packaging Strategy

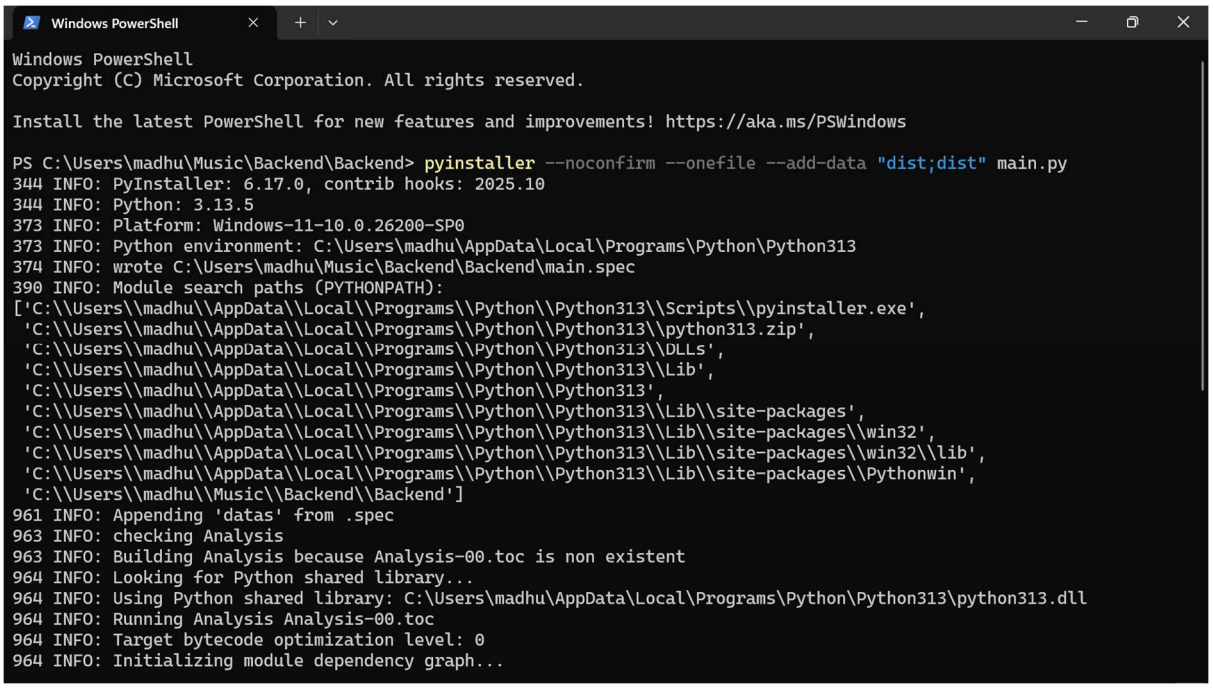
5.1 Tool Used: PyInstaller

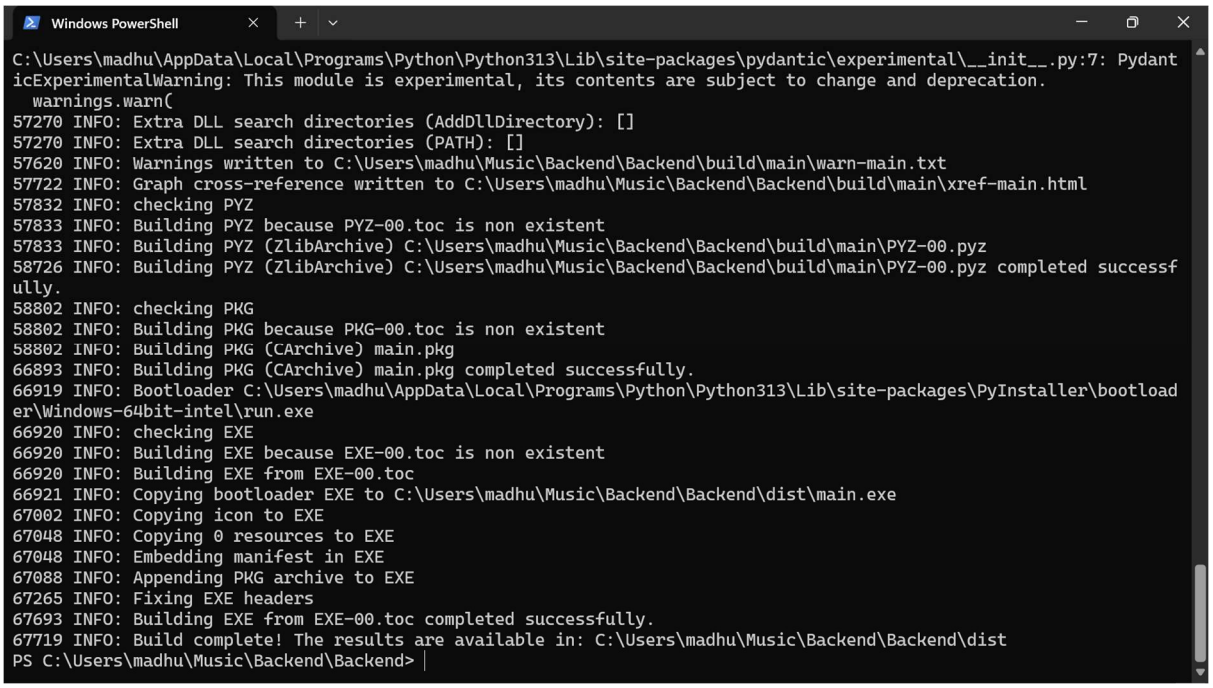
PyInstaller is used to package the Python backend into a native executable.

Key capabilities:

- Bundles Python interpreter
- Includes all required Python libraries
- Packages application code and static assets
- Produces a single native binary:
    - .exe on Windows
    - .app on macOS

The user does **not** need Python installed.





6. Static Asset Bundling

6.1 Frontend Assets

- React build output (dist/) is included using PyInstaller's --add-data
- Assets are extracted at runtime into a temporary directory
- The backend resolves paths dynamically to serve files correctly

6.2 Runtime Static Server

- A lightweight Python HTTP server serves:
    - / → index.html
    - /assets/* → frontend assets
    - /leaflet/* → map icons and static resources
- SPA fallback routing is handled for client-side navigation

7. Runtime Dependency Validation

7.1 External Dependencies

The application depends on:

- **Wireshark (TShark)**
- **Npcap** (Windows packet capture driver)
- **A modern browser**

These cannot be bundled due to:

- Kernel-level drivers

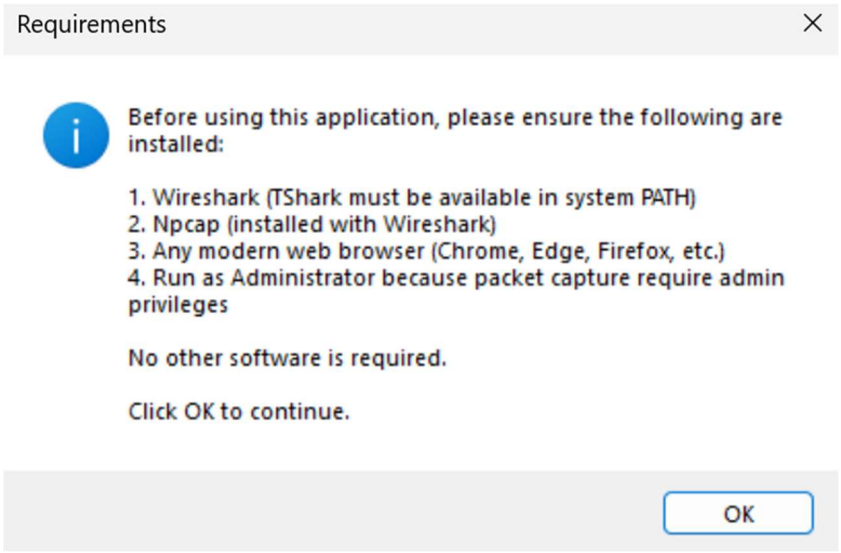
- OS security constraints
- Licensing considerations

8. Startup Validation Flow

8.1 Initial Requirements Notification

On startup, an informational popup explains:

- Required dependencies
- Packet capture constraints
- Browser usage



8.2 Tshark Presence Check

The application checks:

shutil.which("tshark")

This verifies that:

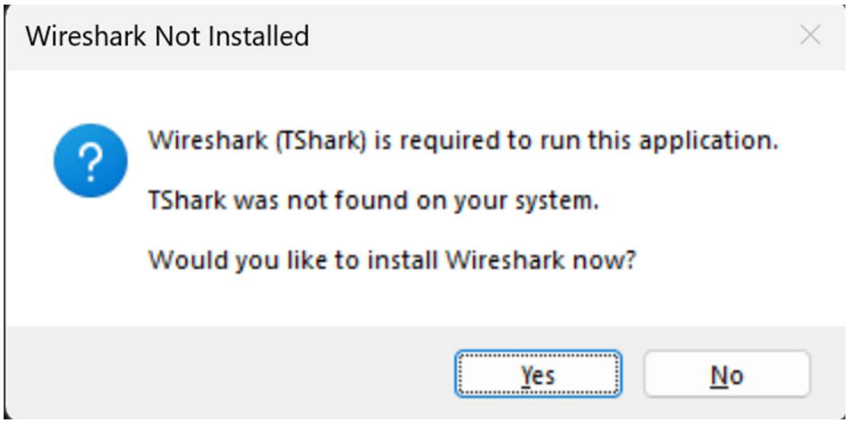
- Wireshark is installed
- Tshark is accessible via system PATH

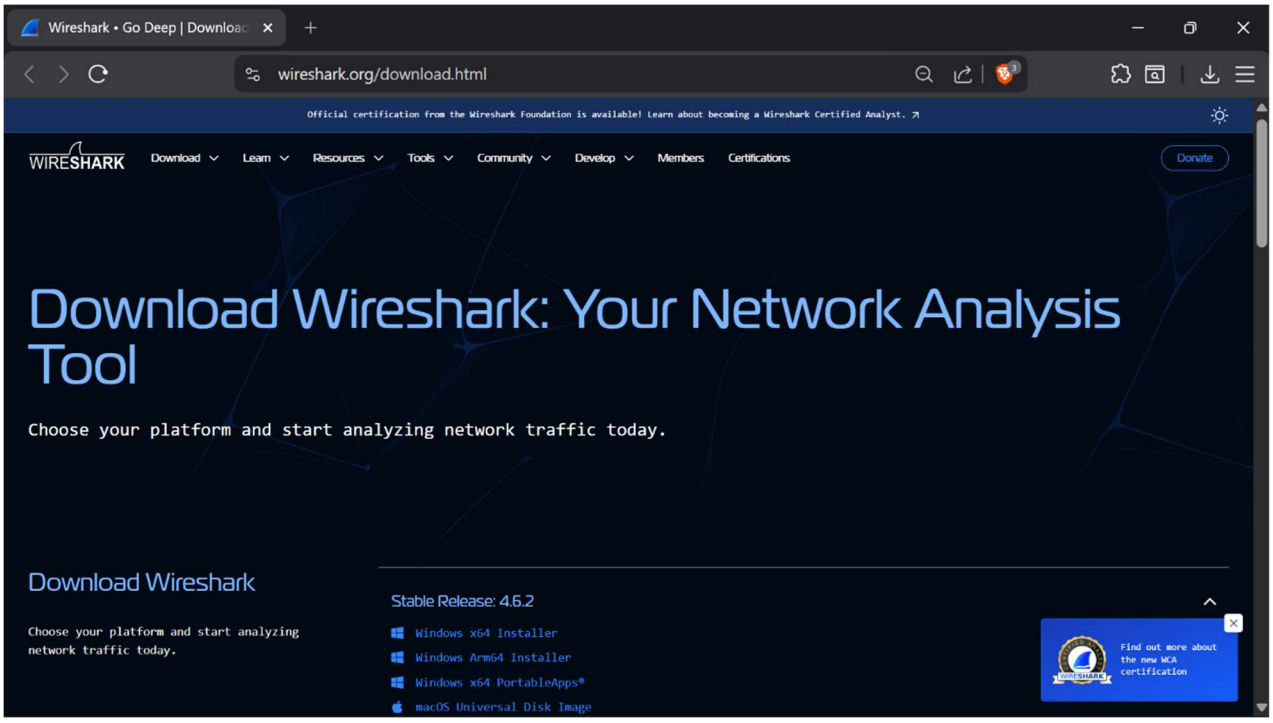
8.3 Missing Dependency Handling

If tshark is not found:

- A custom popup is shown with two options:
    - **Install Wireshark**
    - **Exit**
- Choosing **Install** redirects the user to the official Wireshark installer
- Application terminates cleanly

This prevents partial startup or silent failures.





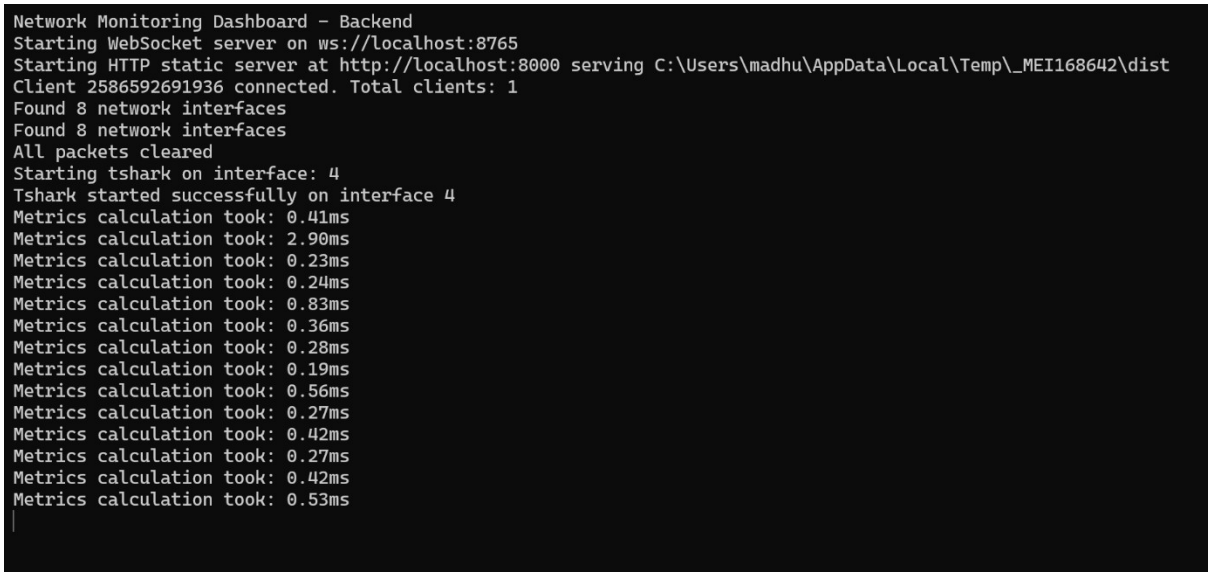
9. Terminal-Based Execution and Log Visibility

The application executable is intentionally created as a **terminal-based application** rather than a silent background process.

Purpose:

- To allow users and developers to view runtime logs directly
- To provide immediate feedback on:
    - Backend startup status
    - WebSocket server initialization

- Packet capture lifecycle (start/stop)
- Error and exception messages



Benefits:

- Improves transparency during execution
- Simplifies debugging and troubleshooting
- Prevents silent failures during packet capture
- Enables easier validation in testing and enterprise environments

This design choice ensures that operational behavior of the application remains **observable and verifiable**, which is critical for a network monitoring system.

10. Windows Executable Creation Process

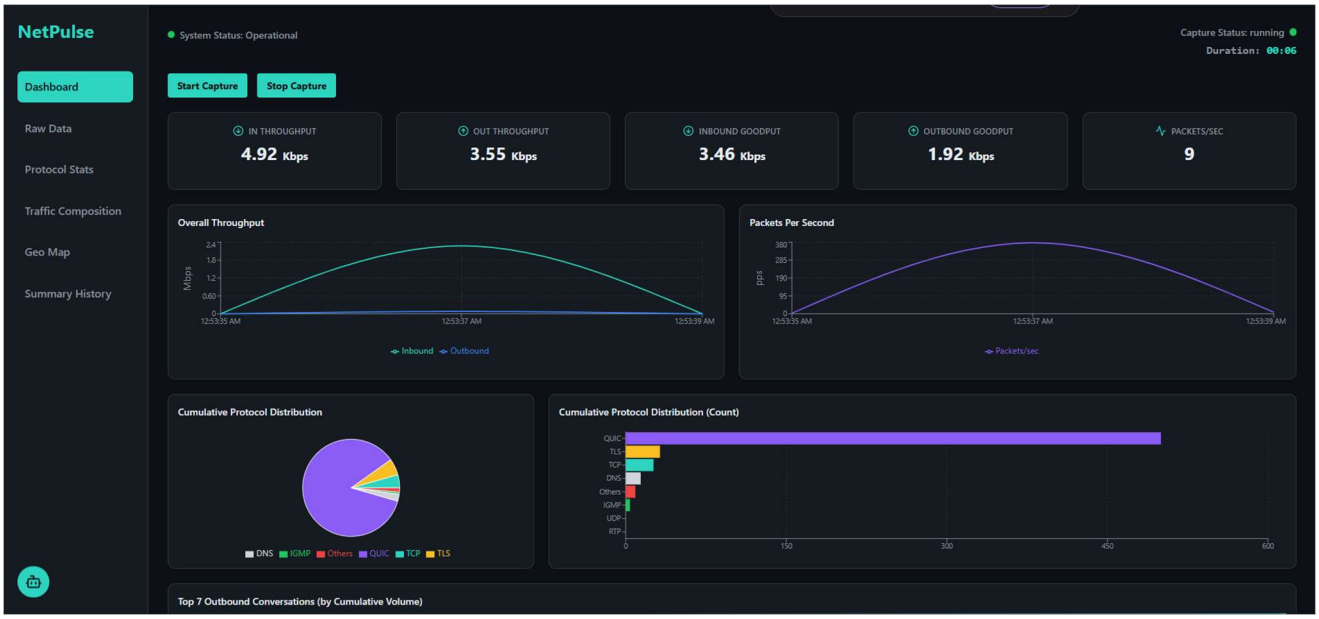
10.1 Steps

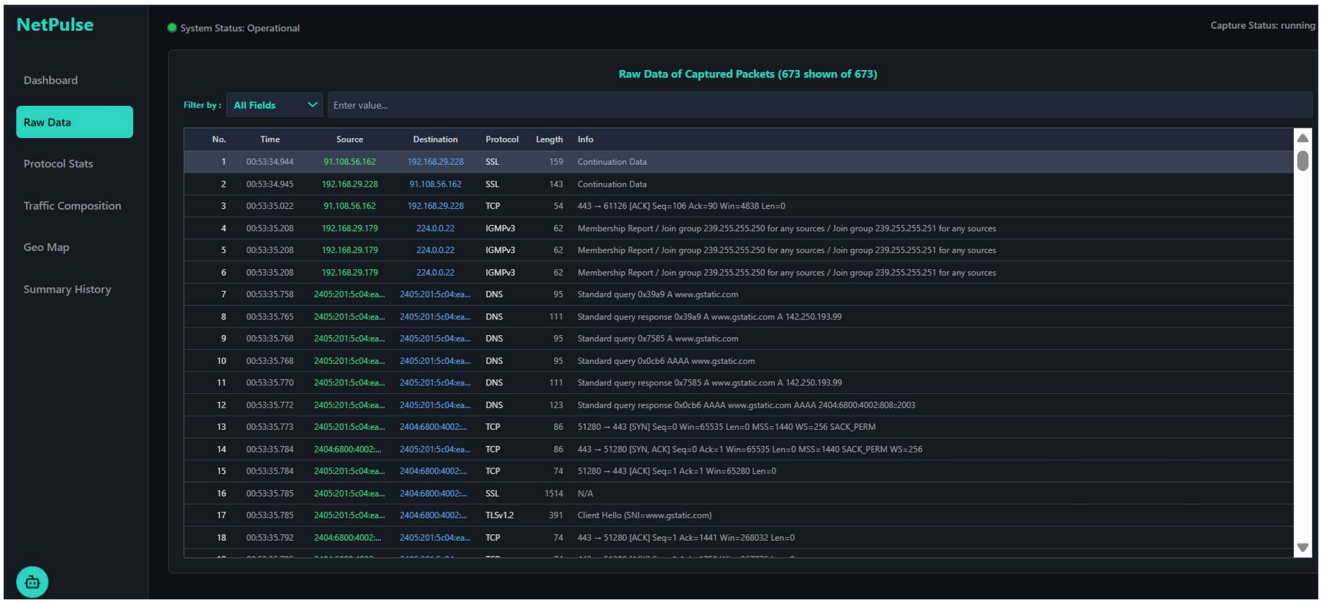
1. Finalize Python backend
2. Build React frontend (npm run build)
3. Move dist/ into backend directory
4. Use PyInstaller with static asset inclusion
5. Generate single .exe file
6. Test on a clean Windows system

10.2 Output

- One .exe file
- No Python, Node.js or Vite dependency

- External dependency checks handled at runtime





11. macOS Executable Creation Process

11.1 Steps

- Same Python and frontend codebase
- PyInstaller generates a macOS .app bundle
- Static assets embedded identically
- Runtime dependency checks remain unchanged

12. Conclusion

The executable-based deployment strategy enables a **production-ready, enterprise-friendly network monitoring application** that runs reliably on Windows and macOS while maintaining accurate access to host network interfaces.