Application Packaging
&
Deployment Strategy

DEPLOYMENT STRATEGY

1. **Windows & macOS - Native Executable Application**

2. Linux - Docker Containerized Deployment

Our deployment strategy is tailored to leverage platform-specific advantages while ensuring optimal performance and functionality. This approach addresses the unique requirements for packet capture across diverse operating environments.

KEY CHALLENGES IN NATIVE EXECUTABLE

1. **Complex Architecture** - Integrating a Python backend with a React frontend within a single executable
2. **External Dependencies** - Ensuring reliable inclusion and interaction with critical external tools like TShark and Npcap is paramount. Other dependencies include python and node.js.
3. **User-Friendly Installation** - The goal is a single-click experience, abstracting away Python or Node.js installations for the end-user, highlighting engineering complexity.
4. **Multiple Background Services -** The application orchestrates HTTP, WebSocket, and packet capture processes concurrently, all managed seamlessly.

Integrating Frontend into Executable: Static Build Strategy

Why Avoid Dynamic Frontend Serving?

- Development servers (like Vite/Node.js) are not suitable for production.
- Require Node.js runtime on user machines, increasing prerequisites.
- Introduce additional background processes, consuming resources.
- Difficult to bundle into a single native executable.
- Increase deployment and maintenance complexity.

The Static Frontend Build Strategy

Our React frontend is compiled using `npm run build`, converting code into:

- Static HTML files
- Minified JavaScript bundles
- Optimized CSS assets

This process removes all development-time dependencies, resulting in a lightweight `dist/` folder of static web assets.

Packaging Tool: PyInstaller

- PyInstaller is a robust tool designed to transform Python applications into **standalone native binaries**.
- It meticulously bundles the Python interpreter, all required Python libraries, and your application's source code into a single, redistributable package.
- The final output is a user-friendly .exe file for Windows or an .app bundle for macOS, eliminating external dependencies and streamlining deployment.
- **No Python, Node.js or Vite required.**

Runtime Dependency Validation

Pre-flight Checks

Upon launch, the application automatically verifies the availability of critical component Tshark.

System PATH Validation

Validation is performed by scrutinizing the system's PATH environment variable, ensuring that all necessary executables are accessible.

User Experience

In case of missing dependencies, an informational popup is displayed, guiding the user with clear instructions and preventing silent failures, showcasing production-grade UX thinking.

Runtime Dependency Validation

Application Startup

The application initiates, triggering its internal dependency checks.

Initial Requirements Popup

A preliminary popup informs the user about essential prerequisites.

TShark Validation

The system specifically checks for the presence and accessibility of TShark.

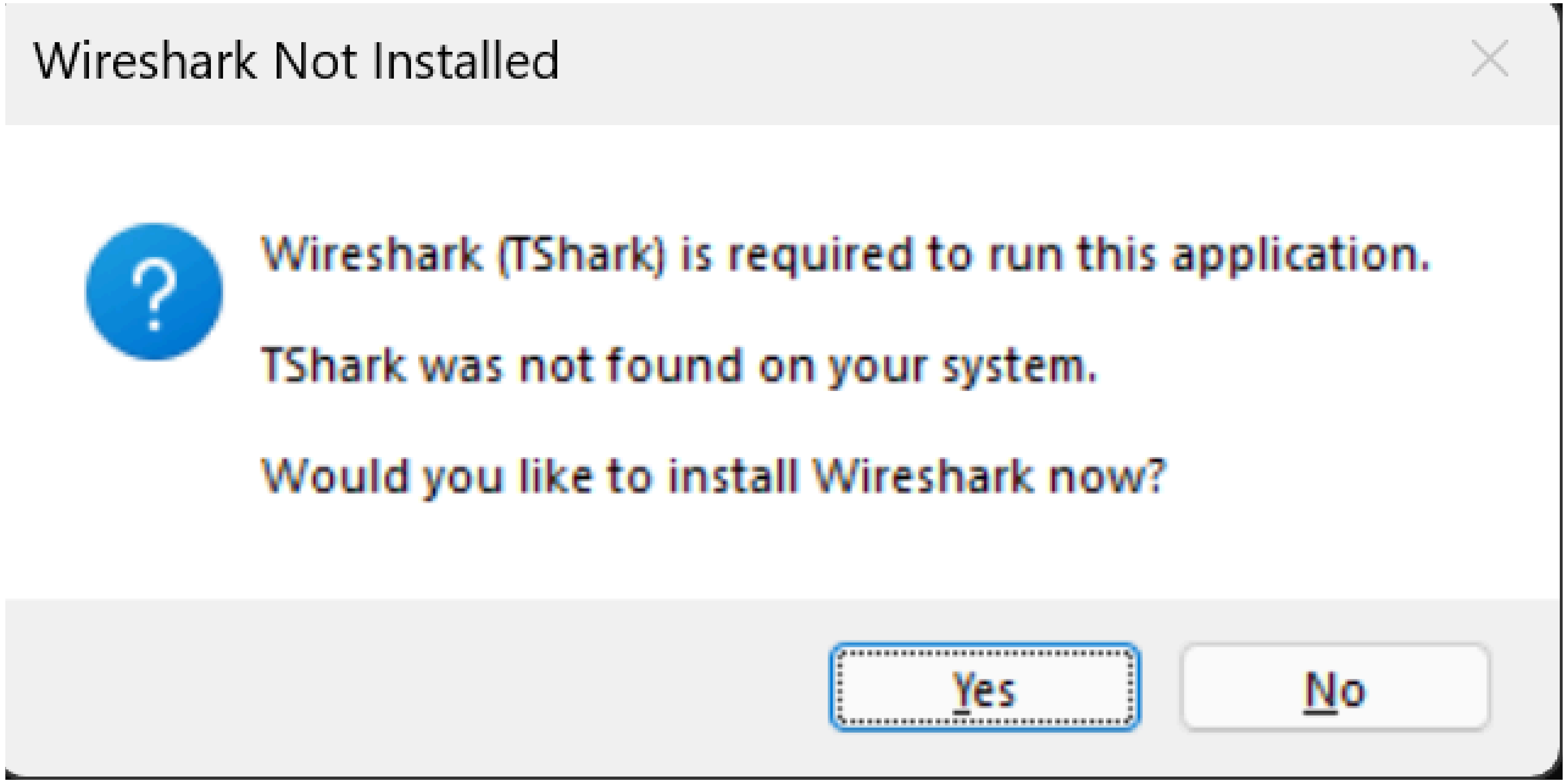
Missing TShark Notification

If TShark is not detected, a clear popup appears, offering "Install Wireshark" or "Exit" options.

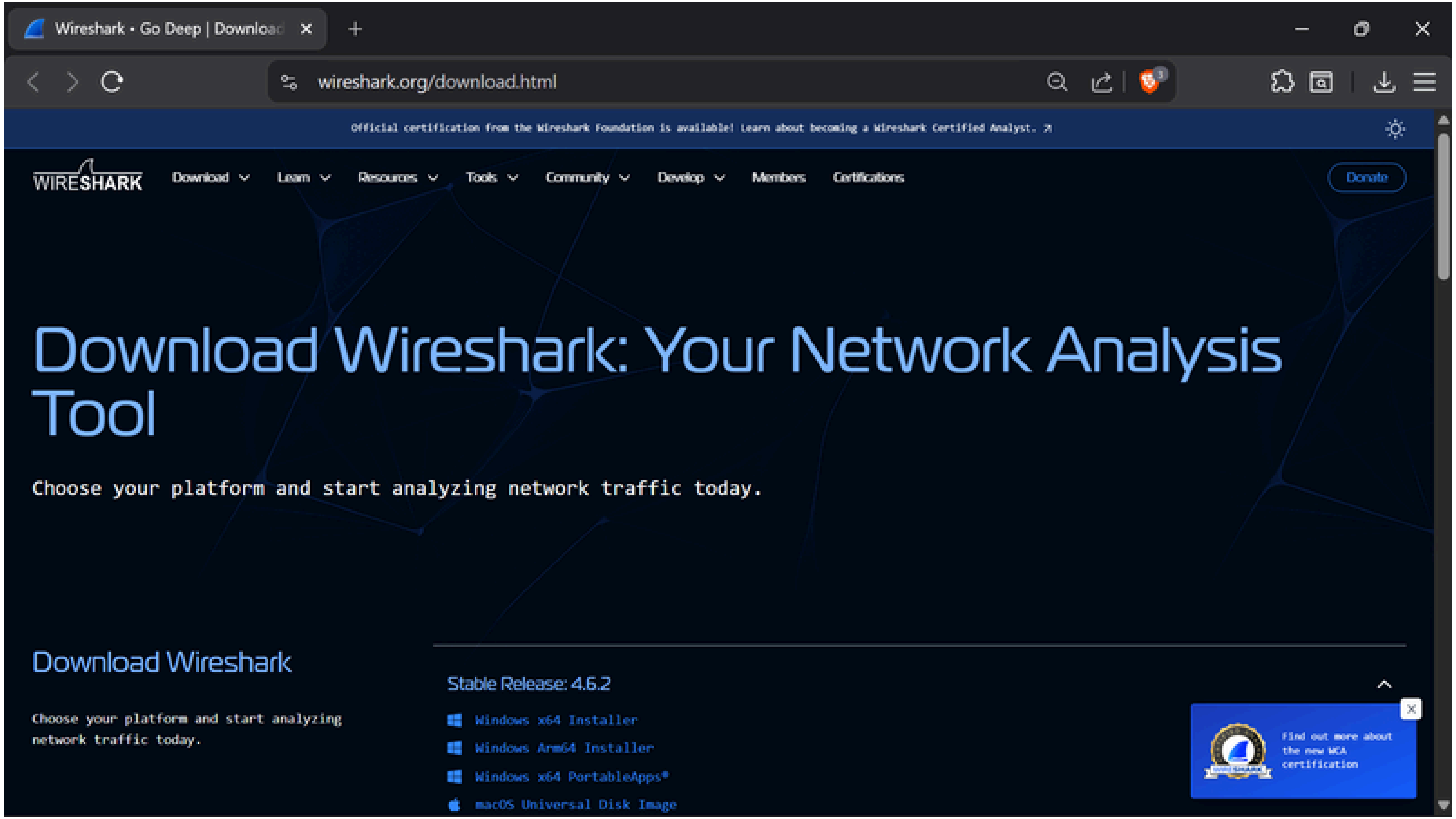
Guided Installation

Clicking "Install" directs the user to the official Wireshark website, followed by a safe termination of the application, aligning with enterprise software best practices.

Runtime Dependency Validation



Runtime Dependency Validation



Windows Executable Creation Flow

1

Backend Finalization

The Python backend code is meticulously finalized and optimized for deployment.

2

Frontend Bundling

The React frontend is built and its static assets are efficiently bundled.

3

PyInstaller Execution

PyInstaller is invoked to bundle both the backend and frontend components, including all static assets.

4

Single .exe Output

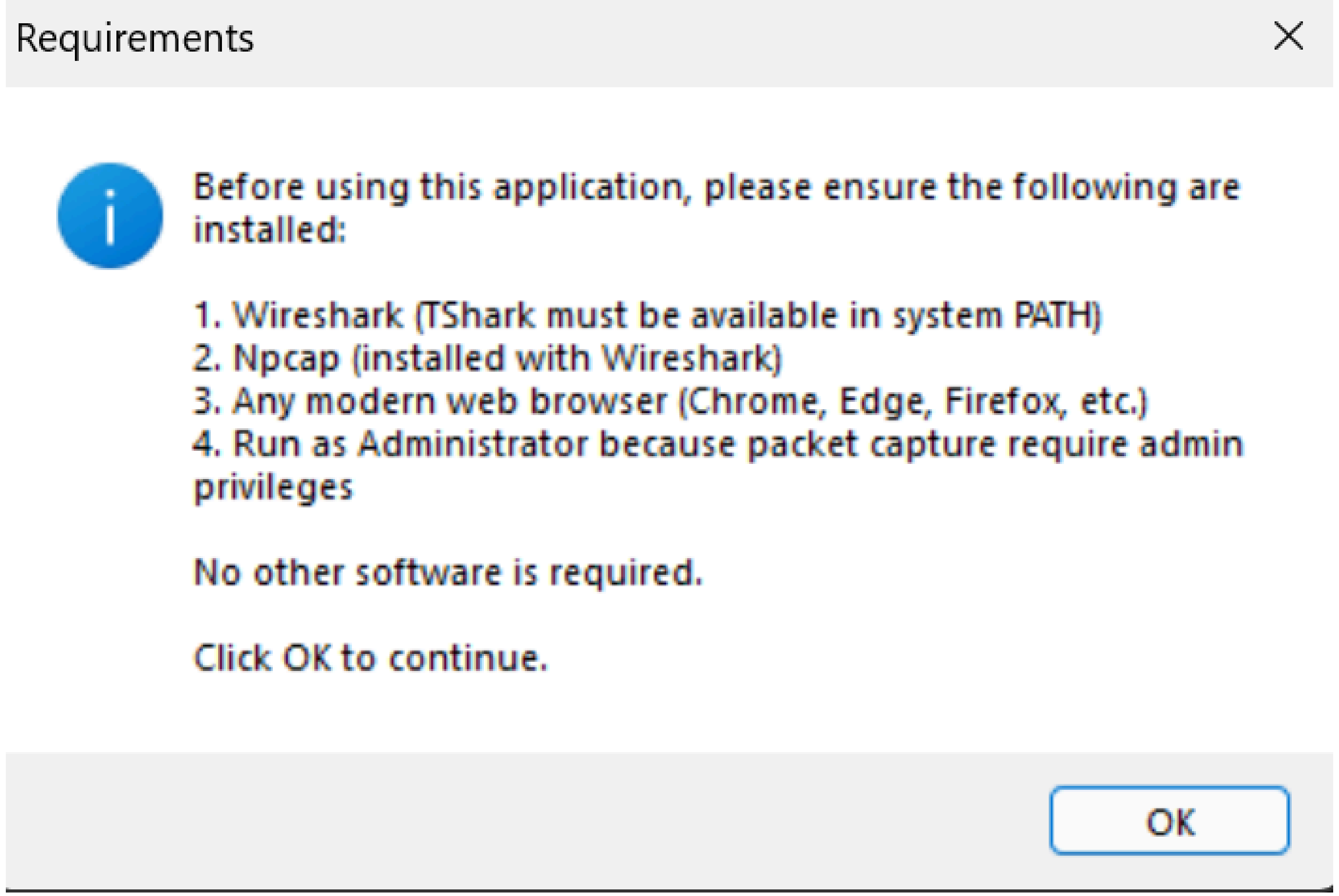
The culmination is a single, self-contained `.exe` file, ready for distribution and execution.

5

Rigorous Testing

The executable undergoes thorough testing on a clean system to validate its functionality and independence.

Screenshots



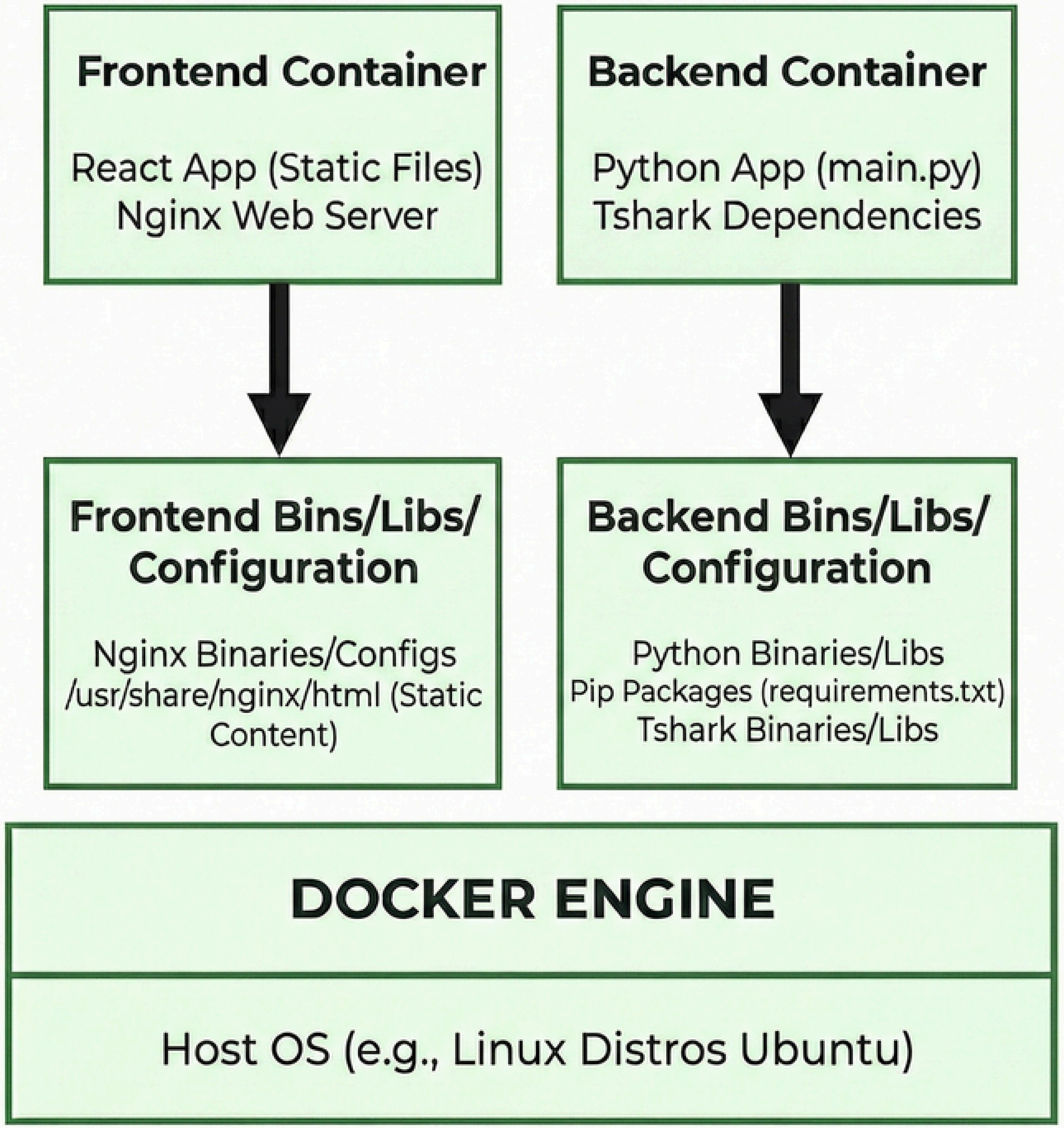
Screenshots

```
Network Monitoring Dashboard - Backend
Starting WebSocket server on ws://localhost:8765
Starting HTTP static server at http://localhost:8000 serving C:\Users\madhu\AppData\Local\Temp\_MEI168642\dist
Client 2586592691936 connected. Total clients: 1
Found 8 network interfaces
Found 8 network interfaces
All packets cleared
Starting tshark on interface: 4
Tshark started successfully on interface 4
Metrics calculation took: 0.41ms
Metrics calculation took: 2.90ms
Metrics calculation took: 0.23ms
Metrics calculation took: 0.24ms
Metrics calculation took: 0.83ms
Metrics calculation took: 0.36ms
Metrics calculation took: 0.28ms
Metrics calculation took: 0.19ms
Metrics calculation took: 0.56ms
Metrics calculation took: 0.27ms
Metrics calculation took: 0.42ms
Metrics calculation took: 0.27ms
Metrics calculation took: 0.42ms
Metrics calculation took: 0.53ms
```

Screenshots

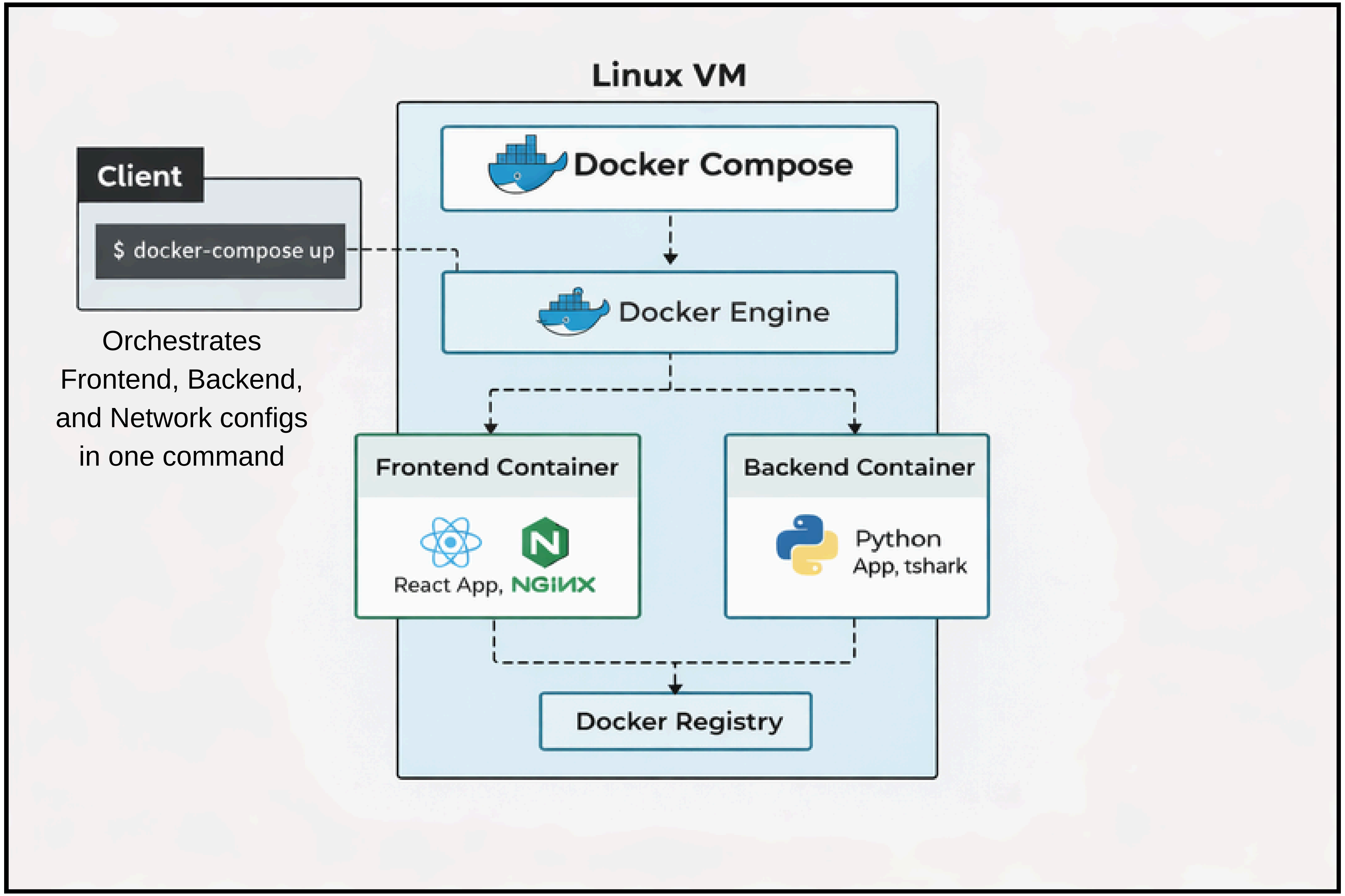
Docker Containerisation for Linux Distros

- Server-Grade Standard
- Solves Fragmentation
- Guarantees Reproducibility
- Simplified Lifecycle

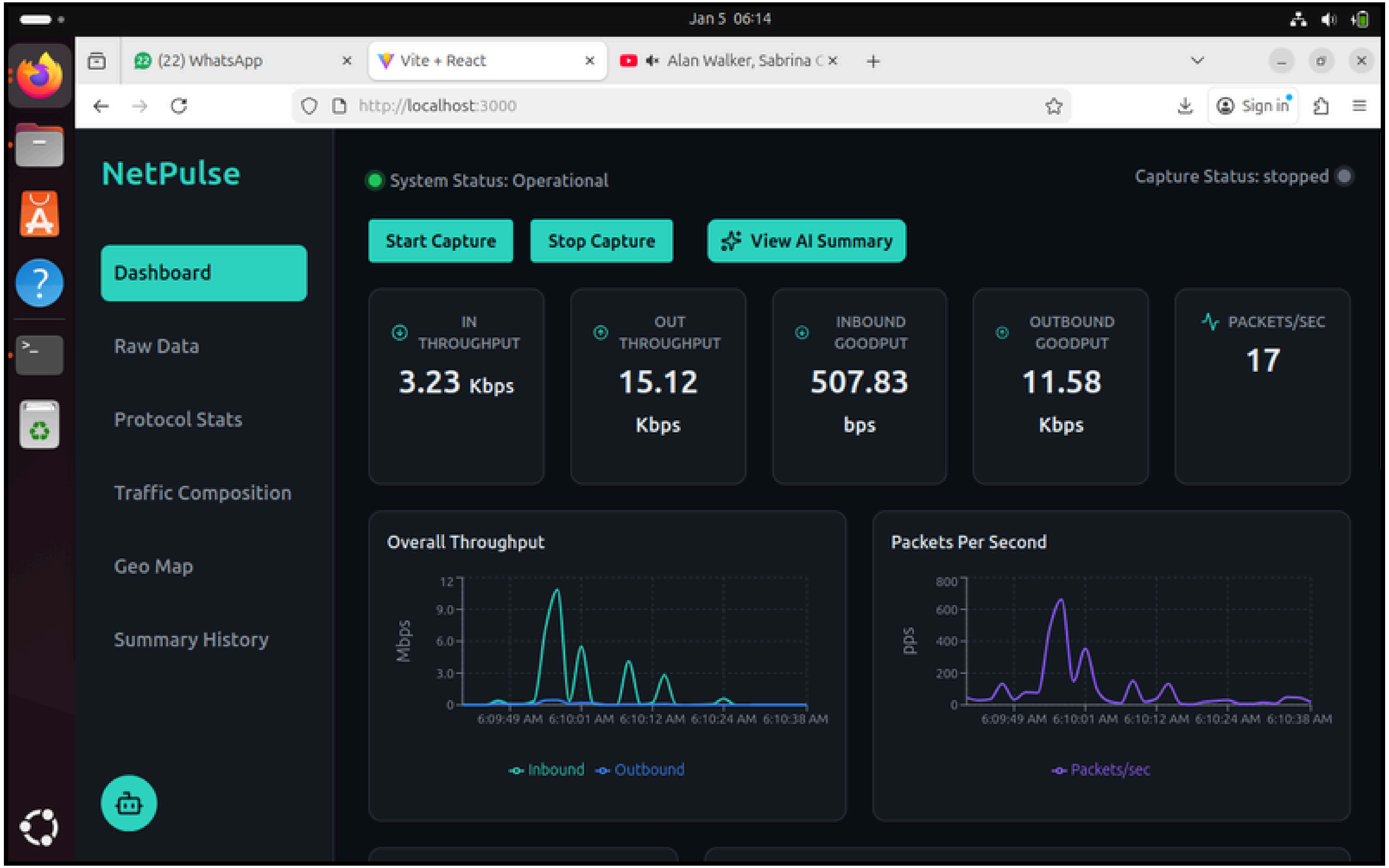


Docker Architecture for HPE Network Analysis Dashboard

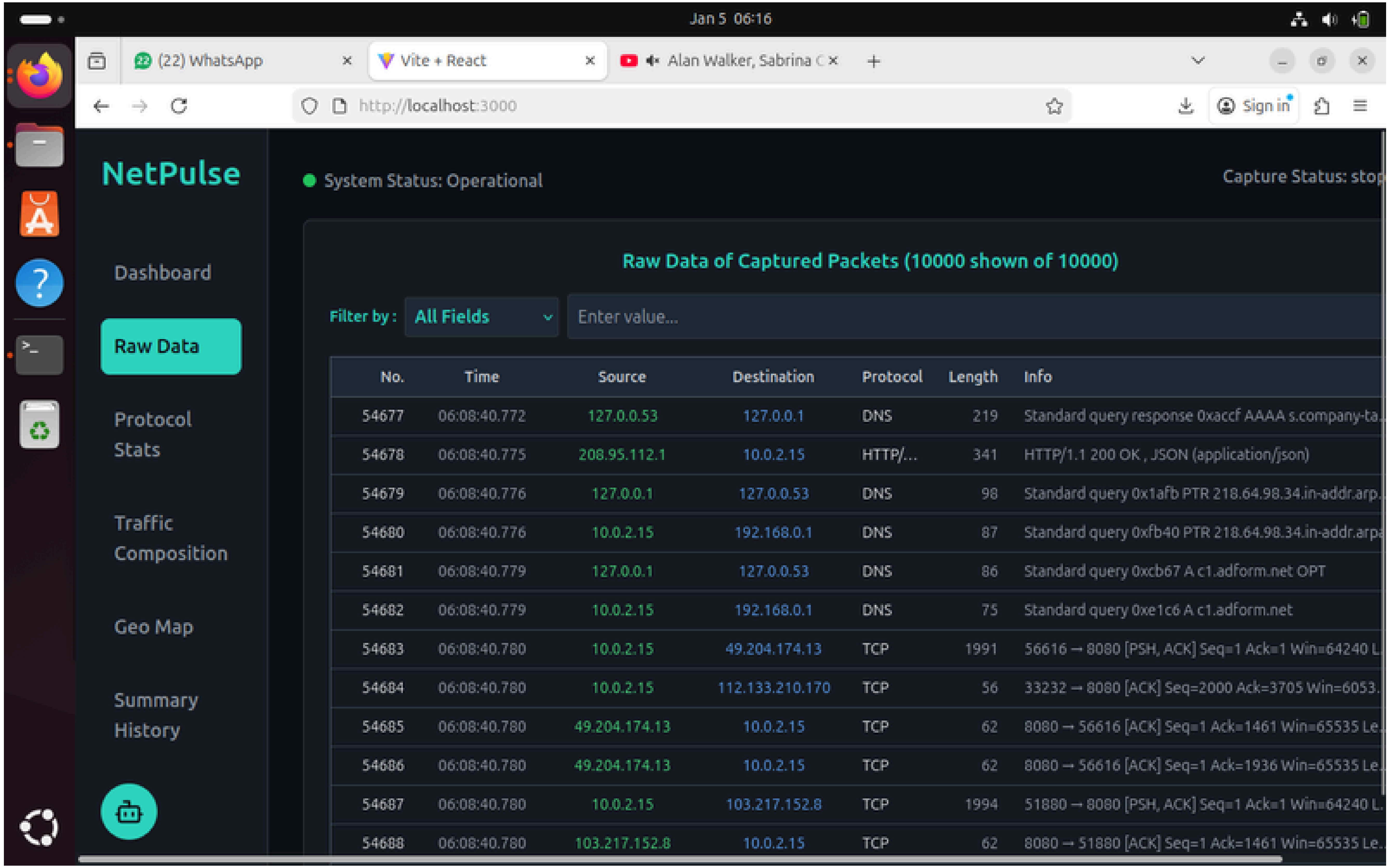
Simple Installation on VMs



Screenshots



Screenshots



Why Not Containerisation For Windows ?

- Docker on Windows runs on top of the WSL2 subsystem (a Virtual Machine), isolating it from the actual physical hardware.
- Due to this virtualization, containers cannot access the Host Network Interface Card (NIC) in "promiscuous mode" to sniff traffic.
- The application is forced to look at the virtual "Docker Bridge" network instead of the real WiFi or Ethernet adapter.
- Consequently, the app can only detect internal traffic flowing between containers, completely missing all real-world internet traffic.
- Native execution (.exe) is mandatory to bypass this virtualization layer and capture actual live data packets.

System Status: Operational

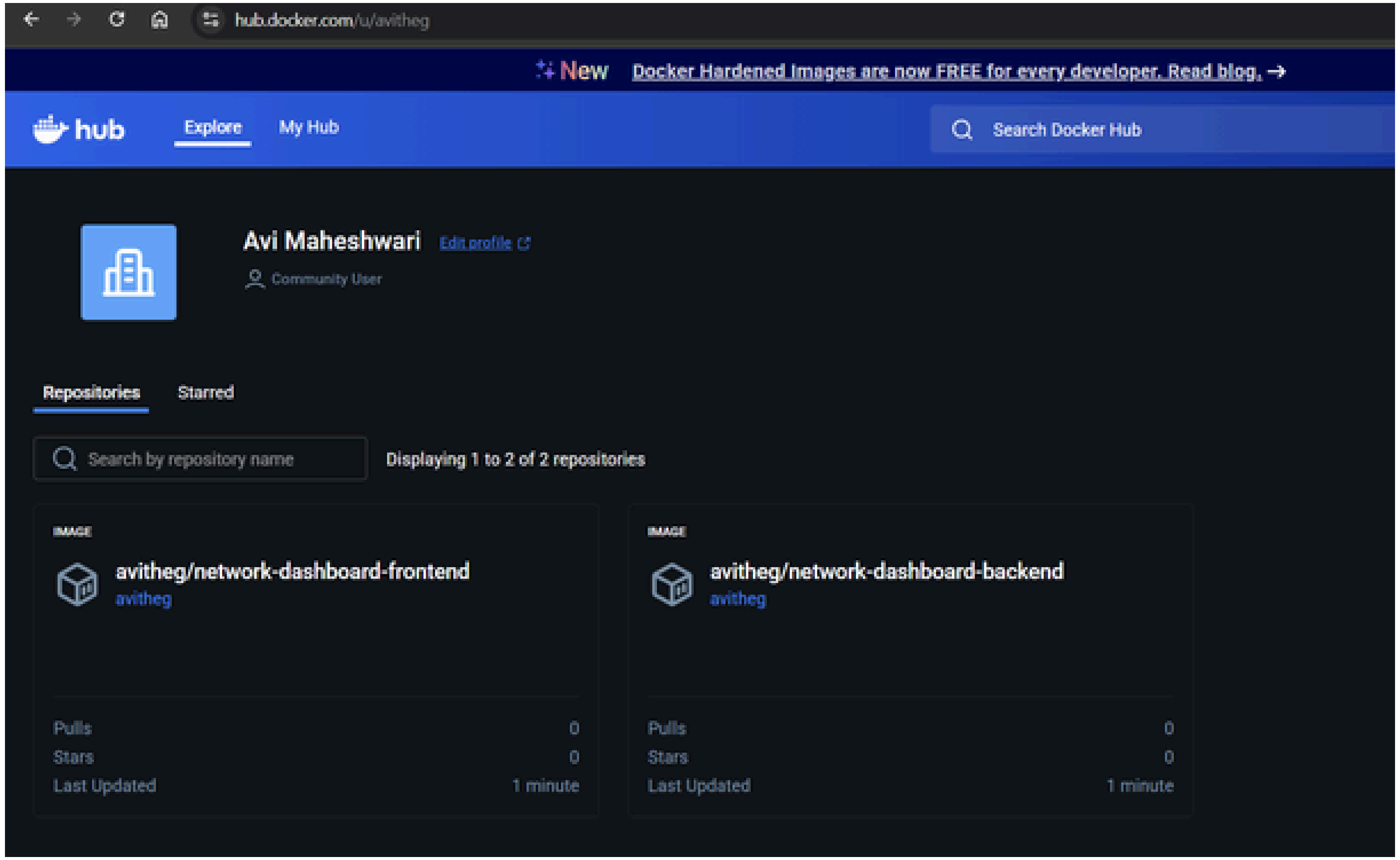
Capture Status: running

Raw Data of Captured Packets (16 shown of 16)

Filter by : All Fields | Enter value...

No.	Time	Source	Destination	Protocol	Length	Info
1	05:08:13.423	172.18.0.2	172.18.0.1	TCP	351	8765 → 51684 [PSH, ACK] Seq=1 Ack=1 Win=505 Len=283 TSval=3106792125...
2	05:08:13.423	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=284 Win=493 Len=0 TSval=1127679497 TSec...
3	05:08:15.685	172.18.0.2	172.18.0.1	TCP	563	8765 → 51684 [PSH, ACK] Seq=284 Ack=1 Win=505 Len=495 TSval=31067943...
4	05:08:15.685	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=779 Win=490 Len=0 TSval=1127681759 TSec...
5	05:08:17.794	172.18.0.2	172.18.0.1	TCP	633	8765 → 51684 [PSH, ACK] Seq=779 Ack=1 Win=505 Len=565 TSval=31067964...
6	05:08:17.794	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=1344 Win=486 Len=0 TSval=1127683868 TSe...
7	05:08:20.036	172.18.0.2	172.18.0.1	TCP	633	8765 → 51684 [PSH, ACK] Seq=1344 Ack=1 Win=505 Len=565 TSval=3106798...
8	05:08:20.036	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=1909 Win=482 Len=0 TSval=1127686110 TSe...
9	05:08:22.143	172.18.0.2	172.18.0.1	TCP	610	8765 → 51684 [PSH, ACK] Seq=1909 Ack=1 Win=505 Len=542 TSval=3106800...
10	05:08:22.143	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=2451 Win=478 Len=0 TSval=1127688217 TSe...
11	05:08:24.387	172.18.0.2	172.18.0.1	TCP	616	8765 → 51684 [PSH, ACK] Seq=2451 Ack=1 Win=505 Len=548 TSval=3106803...
12	05:08:24.387	172.18.0.1	172.18.0.2	TCP	68	51684 → 8765 [ACK] Seq=1 Ack=2999 Win=474 Len=0 TSval=1127690461 TSe...

Images on Docker Hub



docker pull avitheg/network-dashboard-backend

docker pull avitheg/network-dashboard-frontend