Docker Containerization Strategy: NetPulse Network Analysis Dashboard

1. Why Containerization & Docker for Linux Systems:

For Linux-based deployments, we utilize Docker to adhere to Server-Grade Standards. Unlike consumer OS environments (Windows/macOS), Linux servers allow containers deeper access to network interfaces, making Docker the ideal deployment strategy for this environment.

- **Solves Fragmentation:** Linux distributions (Ubuntu, CentOS, Debian) vary significantly. Docker abstracts these OS-level differences, ensuring the application runs identically on any Linux kernel.
- **Guarantees Reproducibility:** The exact versions of Python, Node.js, and system libraries (like Tshark) are locked within the image. If it runs on the development machine, it runs on the server.
- **Simplified Lifecycle:** Updates are deployed by simply pulling a new image tag, rather than running manual update scripts or installer wizards on the host.

2. Container Architecture:

The application follows a decoupled microservices architecture, orchestrated to run seamlessly on the Linux host.

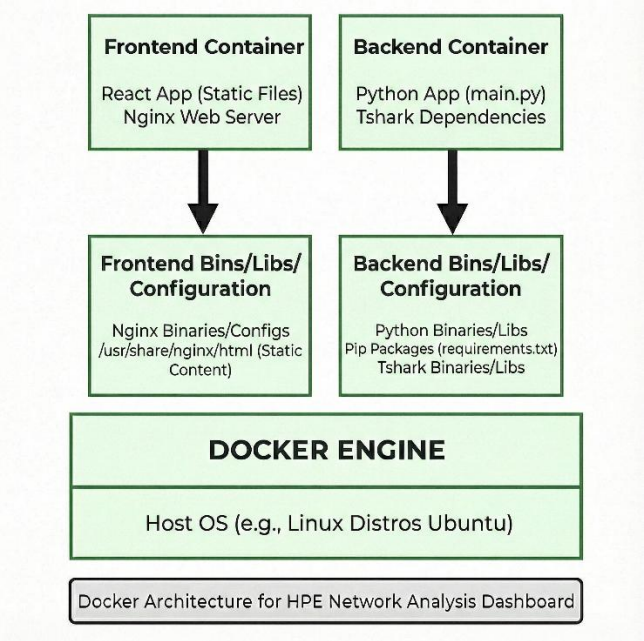

Figure 1 : Containers Architecture

Frontend Container:

- **Role:** Serves the React UI and static assets.
- **Technology:** Nginx (Alpine Linux variant).

- **Behavior:** Acts as a high-performance web server. It does not run Node.js at runtime; instead, it serves the pre-compiled production build of the React application.

Backend Container:

- **Role:** Handles business logic and packet capture.
- **Technology:** Python 3.11 Slim.
- **Critical Dependency:** Includes tshark (Wireshark command-line tool) installed directly at the system level within the container to enable packet analysis.

3. Building the Images:

Frontend Build Strategy (Multi-Stage)

We use a Multi-Stage Build to keep the final image extremely lightweight.

1. Build Stage: Uses node:18-alpine to install dependencies and compile the React code (npm run build), creating a /dist folder.
2. Production Stage: Uses nginx:alpine to serve only the static files from the build stage. The heavy Node.js environment is discarded

```
# Stage 1: Build
FROM node:18-alpine AS build_stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build  # Creates /app/dist

# Stage 2: Serve
FROM nginx:alpine
COPY --from=build_stage /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Backend Build Strategy

The backend image is built on python:3.11-slim. Crucially, we suppress interactive prompts during the installation of Tshark to ensure the build process is fully automated.

```
# Install System Deps (Tshark) non-interactively
RUN apt-get update && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y tshark && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

4. Docker Compose Orchestration:

To manage the lifecycle of both containers simultaneously, we use Docker Compose. This defines the networking bridge between the services.

- Services: frontend (mapped to host port 3000/80) and backend (mapped to host port 8000/8765).
- Networking: Both containers share a dedicated bridge network, allowing the frontend to query the backend API using the internal service name or localhost mapping depending on the Nginx config.

5. Running on a Virtual Machine (Ubuntu):

The application is validated for deployment on Ubuntu Server VMs.

Deployment Steps:

1. **Provision:** Set up an Ubuntu VM.
2. **Install Engine:** sudo apt install docker.io docker-compose.
3. **Deploy:** Transfer the docker-compose.yml and run docker-compose up -d.

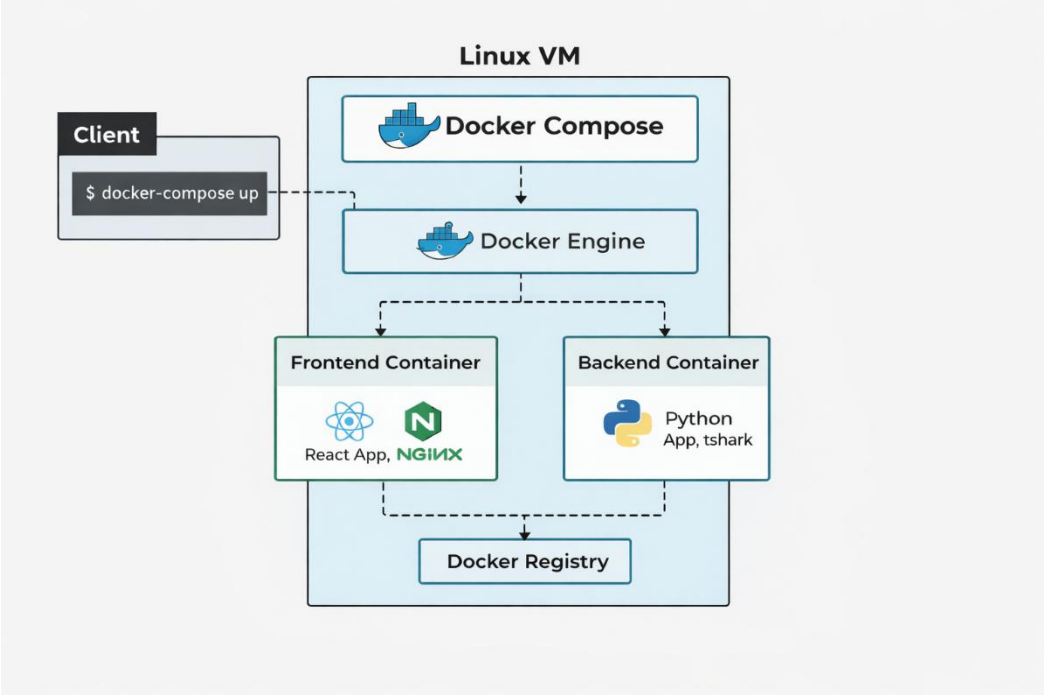

Figure 2: Simple Installation on a Virtual Machine

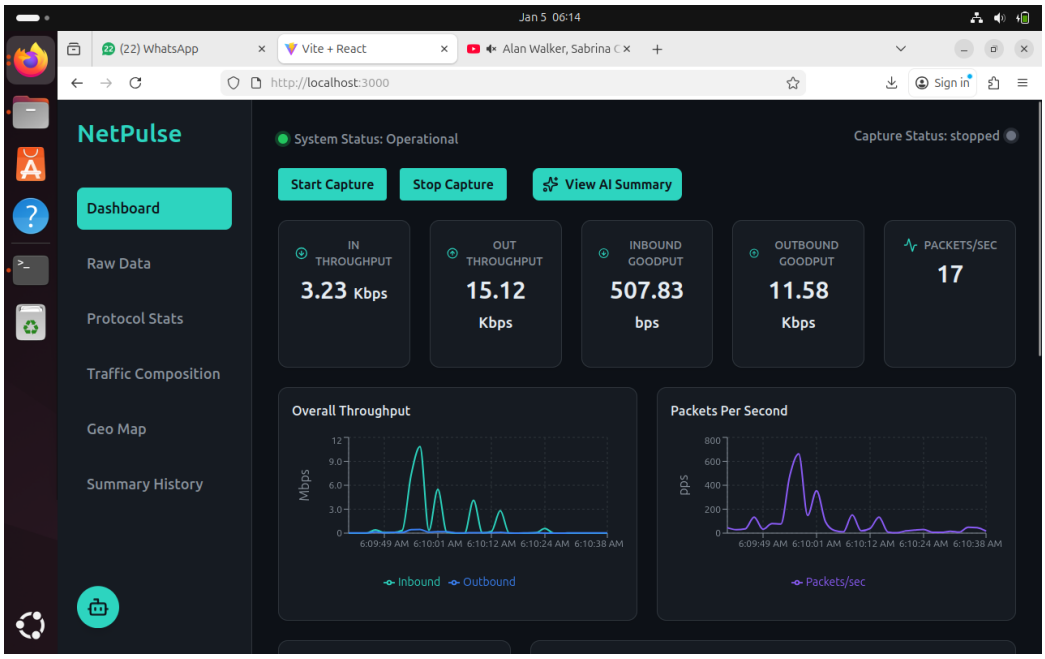

Figure 3 : Running Container on VM

6. Why NOT Containerization for Windows/macOS?

While Docker is excellent for Linux, we explicitly avoid it for the Windows/macOS client releases.

- **Virtualization Barrier:** On Windows and macOS, Docker runs inside a hidden Linux VM (via WSL2 or HyperKit).
- **Promiscuous Mode Failure:** The network adapters inside this VM cannot access the *host's* physical Network Interface Card (NIC) in promiscuous mode.
- **Result:** A containerized Tshark on Windows/Mac would only see internal traffic between the Docker containers, failing to capture the actual network traffic of the user's machine.

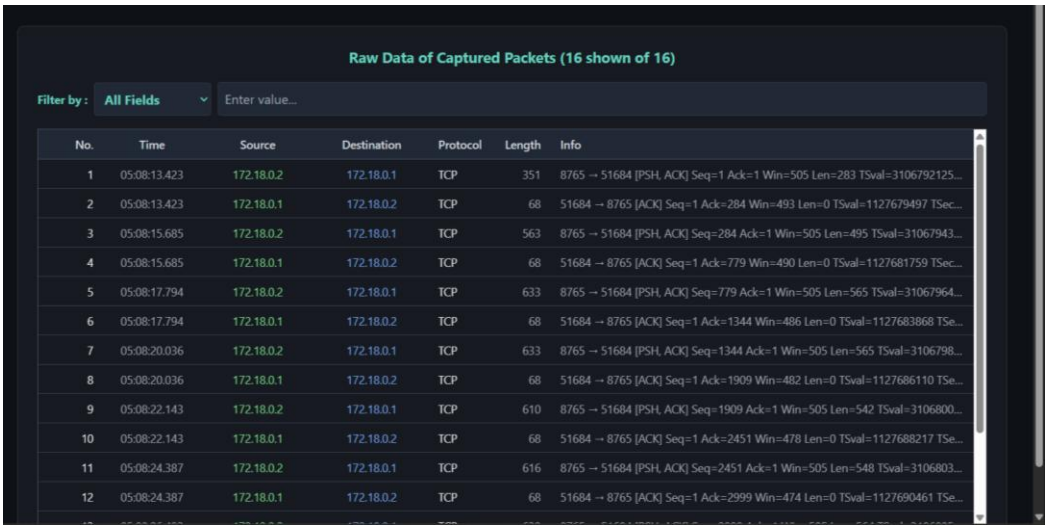

Figure 4: Running Container on Windows

Conclusion:

For Windows/macOS, we utilize the Native Executable approach (documented separately) to ensure direct hardware access. For Linux servers, we use Docker.

7. Deployment Workflow: Docker Hub & Virtual Server:

To distribute the Linux version, we utilize a centralized registry workflow.

- **Tagging**
- **Pushing to Hub:** Images are uploaded to Docker Hub.
- **Production Run:** On the target Linux Virtual Server, we simply pull and run.

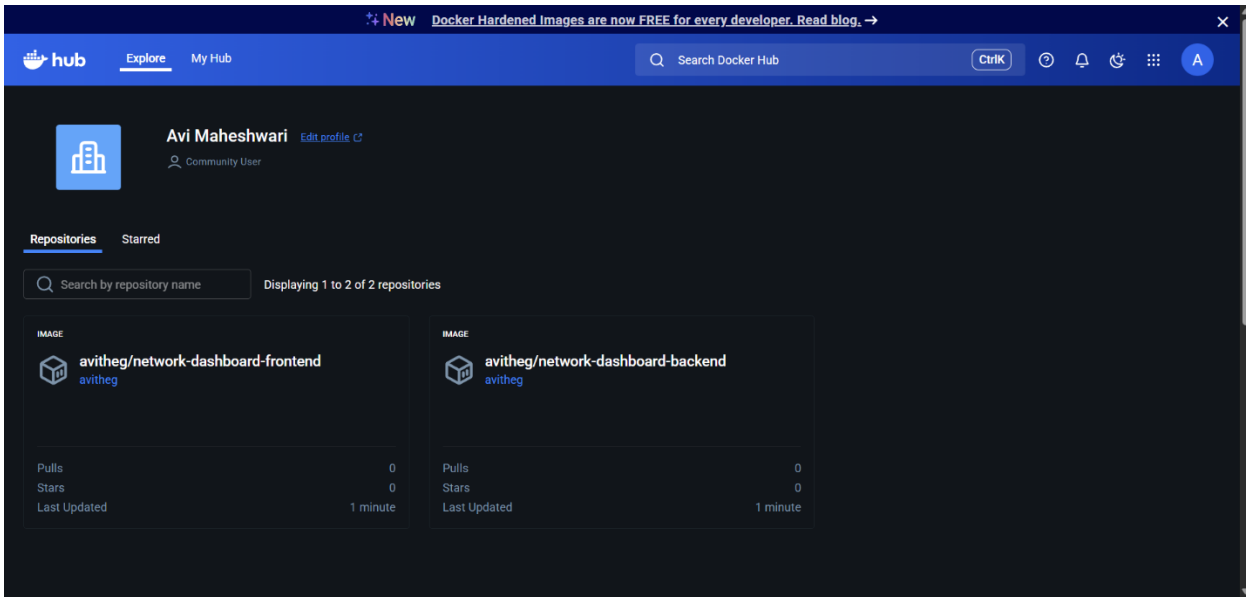

Figure 5 : Images on Docker Hub