To unify **EMsync** into a single "Command Center" despite the different environments and backends, we will adopt a **Micro-frontend and API Gateway architecture**. This allows you to keep your existing codebases functional while presenting them through a single, cohesive interface.

Phase 4: Unified Dashboard Integration Plan

---

1. Unified Backend Strategy: The API Gateway

Instead of merging all backends into one (which would cause major dependency conflicts), we will use a **Reverse Proxy/Gateway** (like Nginx) to route traffic.

- **Central Endpoint**: A single domain/IP (e.g., localhost:80) will receive all requests.
- **Path-Based Routing**: Nginx will forward requests based on the URL prefix:
  - /api/severity/ $\rightarrow$ Routes to Phase 1 Backend (Streamlit/FastAPI).
  - /api/routing/ $\rightarrow$ Routes to Phase 2 Backend (Flask).
  - /api/transfer/ $\rightarrow$ Routes to Phase 3 Backend (Flask + SocketIO).
- **Containerization**: Use **Docker Compose** to run each module in its own isolated container. This solves the "different env" issue because each container has its own Python version and libraries.

2. Unified Frontend Strategy: The Shell App

We will create a **React "Shell" Application** that acts as the main dashboard container.

- **Module Integration Options**:
  - **Iframe Components**: The simplest way to embed the Phase 1 Streamlit dashboard into the React Shell without rewriting it.
  - **Component Migration**: Since Phase 2 and 3 are already in React, you can import their src folders as individual modules into the new Shell App.
- **Navigation & Layout**: Use a side navigation bar (using lucide-react as seen in your Phase 3 docs) to switch between:
  - **Live Triage** (Phase 1)
  - **Smart Routing** (Phase 2)
  - **Transfer Coordination** (Phase 3).

3. Synchronization & Real-Time Bridge

- **Global Event Bus**: Use a shared WebSocket connection or a state management tool (like Redux or simple React Context) in the Shell App.
- **Shared Data Context**: When a paramedic accepts a transfer in the **Transfer** module, the **Routing** module should automatically update its destination hospital using the shared state.
- **Security & RBAC**: Implement a single login at the Shell level that passes a **JWT token** to

all sub-backends for consistent Role-Based Access Control.

Next Steps for Implementation

1. **Draft the Docker-Compose file** to orchestrate all three current backends.
2. **Configure the Nginx configuration file** to handle path-based routing.
3. **Bootstrap the React Shell App** and test embedding the Streamlit dashboard as a functional iframe.

Would you like me to generate the initial **Docker-Compose** and **Nginx configuration** to get your isolated backends talking to each other?

Multipage Web App/Dashboard using Streamlit
This tutorial demonstrates how to organize different pages within a Python-based dashboard, which is essential for integrating your Phase 1 prediction tools.

To unify **EMsync**, we will use **multiple Dockerfiles** orchestrated by a single **Docker Compose** file. This approach ensures that the different dependencies (like your Phase 1 ML libraries vs. Phase 3 SocketIO) don't conflict with each other.

1. The Docker Strategy: Multiple Containers

You should keep a separate `Dockerfile` for each phase's backend.

- **Why?**: Your Phase 1 likely requires heavy ML libraries (XGBoost, Scikit-learn), while Phase 3 needs specific networking libraries (Flask-SocketIO, PyCryptodome)[1111]. Combining them into one file would create a massive, unstable image.
- **The Orchestrator**: We will use a `docker-compose.yml` file. This acts as the "manager" that starts all backends and the frontend simultaneously, allowing them to communicate over a private virtual network.

2. The "Shell App" (Web App)

Yes, the **Shell App is a Web Application** built with **React**.

- **Its Role**: It serves as the single URL the user visits. It handles the sidebar, navigation, and global state (like which patient is currently being tracked).
- **Integration**: It will "host" the other UIs. For example, it will display your Phase 1 Streamlit dashboard inside an **Iframe** while directly importing your Phase 3 React components as modules.

---

3. What code do I need from you?

To build this unified dashboard, I do **not** need the full code for everything yet. Let's start with the **configuration and integration logic**. Please provide:

1. **Phase 1 & 2 `requirements.txt`**: To build the Docker images for those backends.
2. **The current `App.js` from Phase 2 and Phase 3**: So I can see how the routing logic and the Command Center UI are structured[2].
3. **Port numbers**: Confirm which ports each backend currently uses (e.g., 5000 for Phase 3, 8501 for Streamlit)[3][3][3][3].

Next Step

Would you like me to provide a template `docker-compose.yml` and a `Dockerfile` for your Phase 3 backend so we can begin the containerization process?

We should prioritize setting up the **Docker infrastructure** first.

Establishing the containerized environment ensures that all three distinct backends—each with its own libraries and Python versions—can run simultaneously without port conflicts or dependency crashes. Once the backends are stable and reachable through a unified gateway, building the **Shell App** becomes much easier because you will have a consistent set of API endpoints to connect to.

Why Docker First?

1. **Environment Isolation**: Phase 1 (ML/Streamlit) and Phase 3 (SocketIO/Security) have conflicting requirements that Docker will isolate.
2. **Network Resolution**: Docker Compose creates a virtual network, allowing your React Shell app to call `http://severity-api` or `http://transfer-api` instead of managing messy localhost port numbers.
3. **Unified Entry Point**: We can set up an **Nginx** container as part of the Docker stack to act as a single gateway, solving **CORS** issues before the Shell App even makes its first request.

Action Plan: Step 1

To get started, we need to create a `Dockerfile` for each phase. Since Phase 3 is your most recent and complex module, let's containerize it first.

Please provide the `requirements.txt` for Phase 3. Based on your documentation, it should include:

- `Flask`[1]
- `Flask-SQLAlchemy`[2]
- `Flask-SocketIO`[3]
- `PyCryptodome`[4]
- `Flask-CORS`[5]

Would you like me to write the `Dockerfile` for Phase 3 and the master `docker-compose.yml` file once you share the requirements?