<u>Top 3 NFRs:</u>

Performance - Chosen because cold-chain logistics are time-critical (temperature variations can ruin medical products in minutes), requires quick response times, and prevents billions of dollars in annual pharmaceutical losses
Security - Chosen because the system handles sensitive medical shipment data requiring health authorities compliance, uses bearer token authentication, and must protect valuable cargo information
Usability - Chosen because diverse users need intuitive interfaces for emergency responses, must create shipments in under 2 minutes, and requires mobile accessibility for field operations

Trade-offs Made to Prioritize NFRs:

1. Real-time Data vs. Mock Data - Our team chose to use simulated/mock data instead of real API integrations (OpenWeatherAPI, live sensor feeds) to prioritize Performance and Usability. This trade-off was made because the founder couldn't provide real API access in time, but it allowed the team to focus on building a responsive, user-friendly interface that meets the 300ms response time requirement. The benefit received was a fully functional MVP with excellent user experience that can be easily upgraded with real data sources later.

2. Comprehensive Testing vs. Feature Development - Our team allocated an average amount of time for testing by testing all the sprint 3 features and some additional related sprint 2 features to ensure the app is working properly. This trade-off was made because the team needed to ensure robust authentication, role-based access control, and intuitive user interfaces were properly implemented. The benefit received was a secure, user-friendly system with proper authorization that protects sensitive medical shipment data, even though it meant some testing had to be deferred to later sprints.

Performance Testing Report

• Evidence of system responsiveness, including load times and system behavior under different loads.

The system demonstrates excellent responsiveness with backend API responses consistently under 300ms as measured through manual testing and Postman API testing. Frontend load times are optimized through Vite build system with hot module replacement. Real-time socket connections maintain sub-second latency for temperature alerts and shipment updates. The system handles concurrent users efficiently through Flask-SocketIO with eventlet async handling and the backend is able to serve multiple users at once while being very responsive due to running on gunicorn.

• Results from performance testing tools (e.g., JMeter, Lighthouse, or browser dev tools).

Performance testing using browser developer tools shows initial page load times under 2 seconds, with subsequent navigation under 500ms. React component rendering is optimized with proper state management and lazy loading. Database queries are optimized through SQLAlchemy ORM with connection pooling. Automated testing with Jest confirms component rendering performance meets expectations.

• Explanation of how the system meets predefined performance expectations.

The system successfully meets the 300ms backend response time requirement for 100% of shipment CRUD operations as documented in Sprint 1 and Sprint 2 RPMs. Frontend usability targets are achieved with users able to create shipments in under 2 minutes with zero errors. Real-time monitoring maintains responsive updates through WebSocket connections, ensuring critical cold-chain alerts are delivered promptly to prevent product loss.

Security Measures & Testing

• Description of authentication, authorization, and data protection mechanisms.
The system implements Bearer token-based authentication using JWT tokens with 1-hour expiration. Role-based access control (RBAC) is enforced with three user roles: manufacturer, transporter, and transporter_manager. All sensitive routes are protected with @token_required decorator. Password hashing is implemented using bcrypt with salt generation. Data protection includes CORS configuration, environment variable management for secrets, and SQLAlchemy ORM to prevent direct SQL injection.

• Security best practices followed (e.g., HTTPS, password hashing, input validation).
The system follows OWASP security guidelines: bcrypt password hashing with salt, JWT token validation with expiration checks, input validation on all API endpoints (email format, temperature ranges, required fields), CORS configuration to restrict origins, environment variable management for sensitive data (SECRET_KEY, database credentials), SQLAlchemy ORM to prevent SQL injection, and comprehensive error handling without exposing sensitive information.

• Results of basic security tests, such as SQL injection and cross-site scripting (XSS)
prevention.

As the shipment data is sensitive and critical information, it is necessary to prevent any attempts of hacking or information leaks which is why we used best practices for security purposes. We are using Token Authorization to be additionally safe and also using ORMs to prevent issues like SQL injections. We use password hashing for storing the password which maintains privacy and are serving the website through nginx reverse proxy with proper ssl certificates(Https) making the website more secure. We also have CORS policy strictly set to our frontend url.

We did security testing on our product using Burp Suite where we performed 4 different tests:

1. We added XSS attacks by inputting "><script>alert('XSS')</script> in several inputs for the new features and the signup and login and the input is always encoded and doesn't activate the script and displays the input like a normal text. We also tested for stored XSS using the same string and the results were same
2. We then tried performing CSRF which failed as we were using token based authentication, hence, no cookies were involved.
3. We also tried to access other people's accounts by intercepting and modifying requests and tried sending incorrect or invalid inputs like blank passwords to ensure integrity.
4. Lastly, we checked for clickjacking by making sure the website can't be served through iframe which, to our surprise, was. Therefore we added restrictive headers add_header X-Frame-Options "DENY"; and add_header Content-Security-Policy "frame-ancestors 'none'";
   which don't allow iframe embedding and serve our site with them disabling any clickjacking attempts.

Scalability & Availability Considerations

• Explanation of how the system handles increased users or data.
The system uses PostgreSQL database with connection pooling (SQLAlchemy pool_pre_ping=True), Flask-SocketIO for real-time communication with eventlet for async handling, and Gunicorn WSGI server for production deployment. The modular architecture with blueprints allows for horizontal scaling. Database migrations using Alembic ensure schema evolution without downtime.

• Load balancing or caching strategies if applicable.
Redis is included in the Docker Compose setup for potential caching and session storage. The system uses connection pooling for database

connections. Static assets are served through Nginx in production making the build lightweight.

• Deployment strategy and uptime considerations.
As the images are lightweight and the docker image is pulled before recreating, recreating deployment is fast to implement and gives us more than 99% uptime due to the size of the development, which is why we are implementing Recreate Deployment strategy.