

**Department of Computer Science**

This project has been satisfactorily demonstrated and is of suitable form.

This project report is acceptable in partial completion of the requirements for the Master of Science degree in Software Engineering.

Naval Intelligence Platform

Project Title

Ryan Truong Pham

Student Name

Ning Chen, Ph.D.

Advisor's Name

Advisor's signature

Date

Reviewer's Name

Reviewer's signature

Date

Abstract

Maritime operations centers depend on aggregating live vessel positions and environmental conditions from multiple sources into a single situational picture, but the existing landscape forces a tradeoff: commercial platforms like MarineTraffic are closed-source and expensive to license, while building from raw data feeds means stitching together unstable government APIs and managing real-time rendering at scale. This project builds a full-stack Naval Intelligence Platform for tracking ships in real time as a focused, engineering-rigorous demonstration of how those tradeoffs can be solved with modern web technologies.

The system pulls live vessel data from AISStream.io's Automatic Identification System (AIS) to combine it with marine and atmospheric weather data from Open-Meteo in order to create a surveillance interface similar to what defense contractors use for maritime operations. The platform tracks roughly 1,400 vessels at any given moment across North American waters.

This application will run on React and Node.js. The backend is designed to keep a persistent WebSocket connection open to AISStream.io to expose the REST endpoint that the frontend polls every 30 seconds. This polling approach keeps the map synchronized with incoming vessel data while avoiding the performance hit of constant re-renders.

The Naval Intelligence platform is divided into four parts. First, a vessel tracker displays ship positions on an interactive Leaflet map with custom SVG markers that rotate based on each vessel's heading. Second, environmental overlays show wave height and wind direction layers as toggleable overlays from Open-Meteo. Third, a system health dashboard monitors the WebSocket connection, API status, and displays live log messages from a server-side circular buffer. Fourth, a fleet analytics panel computes ten sections of real-time statistics, which include the total vessel count, average speed, fleet composition, regional distribution, and underway versus stationary breakdowns.

Development followed a four-milestone structure spanning core vessel tracking, weather overlays and filtering, system health monitoring, and fleet analytics. All four milestones were completed within the project timeline.

I built this project for my CPSC 597 project at Cal State Fullerton, but it also serves as a portfolio piece for transitioning into defense industry roles. The work directly demonstrates skills needed for fleet health monitoring and systems engineering positions at contractors, specifically for real-time data processing, multi-source API integration, and event-driven architecture patterns common in command center operations.

Table of Contents

Abstract	2
Table of Contents	3
1. Introduction	5
2. Project Objectives	6
2.1 Objective 1: Real-Time Vessel Tracking	6
2.2 Objective 2: Environmental Intelligence Integration	6
2.3 Objective 3: System Operations Monitoring	6
2.4 Objective 4: Fleet Analytics Dashboard	6
3. System Architecture & Design	7
3.1 Architecture Overview	7
3.2 Data Flow	7
3.3 Component Structure	8
3.3 Component Structure	8
4. Technology Stack	10
4.1 Frontend	10
4.1.1 React State Management Strategy	11
4.1.2 Leaflet Integration Challenges	11
4.2 Backend	11
4.2.1 Alternative Backend Approaches Considered	12
4.3 External APIs	12
4.3.1 AISStream.io Message Type Filtering	13
4.3.2 Why Open-Meteo Instead of NOAA	13
4.4 Deployment & Tooling	13
5. Implementation	15
5.1 Milestone 1: Core Infrastructure & Vessel Tracking (Complete)	15
5.1.1 Map Interface	15
5.1.2 AIS Integration & Vessel Tracking	15
5.1.3 System Health Dashboard	16
5.1.4 Work Products	16
5.2 Milestone 2: Environmental Intelligence Integration	20
5.2.1 Wave Height Overlay	20
5.2.2 Why L.circle Instead of a Heatmap Library	20
5.2.3 Wind Direction Overlay	20
5.2.4 Weather Data Panel and Filter Vessels Panel	20
5.2.5 Server-Side Cache and Lazy Fetching	21
5.2.6 Work Products	22
5.3 Milestone 3: System Health and Live Monitoring	24

5.3.1 Work Products	24
5.4 Milestone 4: Fleet Analytics Dashboard	26
5.4.1 Work products	26
6. Challenges & Lessons Learned	28
6.1 API Cost Constraints & Feature Scoping	28
6.2 Real-Time State Management in React	28
6.3 Dual Data Layer Rendering	28
6.4 Learning a New Technology Stack	28
6.5 WebSocket Reconnection and Error Handling	29
6.6 Marker Clustering Performance Optimization	29
6.7 Open-Meteo as a NOAA Replacement	30
6.8 Heatmap Canvas Clipping	30
7. Conclusion	31
8. References	32

1. Introduction

Over the past decade, the aerospace and defense industry has invested heavily in maritime surveillance systems that can track and analyze vessel movements in real time. These systems are fundamental to national security, naval operations, and commercial shipping logistics. As defense contractors and government agencies upgrade their command centers, they need engineers who understand both modern web development and maritime operations. With that in mind, the Naval Intelligence Platform is a full-stack web app that simulates how a real maritime operations command center works. It combines live Automatic Identification System (AIS) vessel data, marine and atmospheric weather feeds from Open-Meteo, and interactive mapping to demonstrate how modern web technologies can handle the complex, data-intensive requirements of defense surveillance systems.

I built this project for the CPSC 597 Graduate Project course at Cal State Fullerton. Development broke down into four milestones: core infrastructure and vessel tracking, environmental intelligence integration, system health and live monitoring, and a fleet analytics dashboard. Beyond the technical work itself, this project serves as a portfolio piece demonstrating the systems engineering skills needed for fleet health monitoring and real-time data processing roles at companies like Boeing, Northrop Grumman, and Lockheed Martin.

2. Project Objectives

The goal of this project is to build a command interface that pulls together multiple intelligence sources into a single, coherent view, providing operators with real-time situational awareness.

2.1 Objective 1: Real-Time Vessel Tracking

Build a live maritime tracker using the AISStream.io API to display current vessel positions on an interactive map with automatic updates. This includes custom ship markers that rotate with the heading, a CSV export of vessel data, and an information pop-up when you click any marker.

2.2 Objective 2: Environmental Intelligence Integration

Layer marine weather data directly onto the vessel map using the Open-Meteo API. This means wave height and wind direction overlays as toggleable Leaflet layers, plus a tabbed weather data panel with CSV export. Also, build a six-control filter panel that lets operators narrow the map by category, movement, speed, destination, navigational status, and geographic region.

2.3 Objective 3: System Operations Monitoring

Create a system health dashboard that shows the status of all data connections in real time. This includes status indicators for the AISStream.io WebSocket and weather API, a live log feed sourced from a 150-entry server-side circular buffer, and vessel count metrics broken down by category. The panel polls a combined health endpoint every 5 seconds.

2.4 Objective 4: Fleet Analytics Dashboard

Build an analytics interface that computes and displays ten sections of statistics from the live AIS stream and cached weather data. Metrics include total active vessels, speed distribution, navigational status breakdowns, regional distribution, top fastest vessels, top destinations, fleet composition by category, and summary statistics for wave height and wind speed. This gives operators a high-level operational snapshot at any moment.

3. System Architecture & Design

The Naval Intelligence Platform uses a client-server architecture built around a persistent WebSocket connection and a REST API layer. The system is broken down into three main sections: external data sources, a Node.js backend server, and a React frontend client.

3.1 Architecture Overview

The backend server, located in the `server/` directory, is a Node.js application that establishes and maintains a WebSocket connection to AISStream.io. As vessel position data is received, the server parses and stores each vessel's most recent state in memory. A REST endpoint (`GET /api/vessels`) is exposed, which the frontend polls at regular intervals to retrieve vessel snapshots without needing a direct WebSocket connection from the browser.

The frontend client, built with React and located in `client/src/`, follows a standard `App.jsx` component that manages global application state and defines the data flow between child components. The `ShipMap.jsx` component manages all Leaflet/OpenStreetMap map rendering, including vessel marker placement, heading-based rotation, and weather overlay layers. Supplementary modules manage the system health dashboard, fleet analytics, and vessel detail views.

3.2 Data Flow

Data flows through the system as follows: AISStream.io broadcasts vessel position updates over a WebSocket connection, which the Node.js backend receives and stores in memory. The React frontend polls the REST endpoint every 30 seconds, receives the latest vessel snapshot array, and triggers a state update. React's reconciliation process then re-renders only the map markers and UI components that reflect changed vessel data, minimizing unnecessary DOM updates and keeping the interface responsive under high vessel counts.

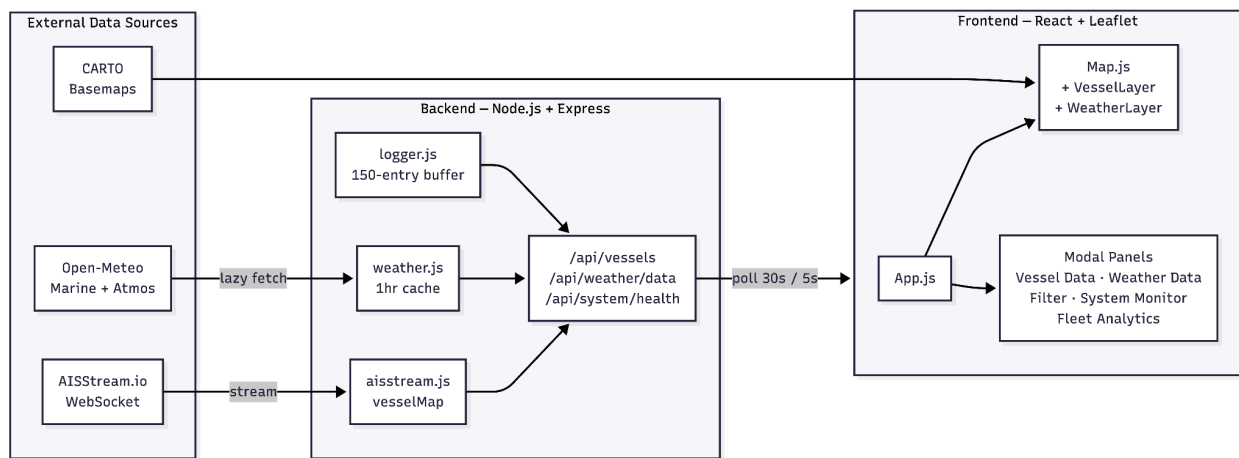
Weather data follows a different pattern. The frontend requests cached weather data from the backend only the first time a weather overlay is toggled on, guarded by a `useRef` flag. The backend checks its in-memory cache; if it's older than 1 hour, it kicks off two parallel HTTPS requests to Open-Meteo (one for wave height, one for wind), merges the results, refreshes the cache, and returns the merged payload. Subsequent requests within the hour are served directly from cache.

A 30-second polling interval was selected to balance responsiveness with backend load. At that rate, a vessel traveling at 20 knots (typical for merchant ships) moves approximately 300 meters between updates, well within the map's visual precision at normal zoom levels. Server-sent events and direct WebSocket-to-browser updates were considered as alternatives, but polling keeps the architecture simpler: the backend does not need to track connected clients or manage WebSocket lifecycles, and

remains stateless with respect to frontend clients. This stateless property simplifies deployment and would allow horizontal scaling of the backend behind a load balancer if needed.

3.3 Component Structure

The four core modules of the application will consist of System, Fleet, Environmental, and Vessel management. These milestones will be implemented as distinct React components, each responsible for a specific domain of the command interface. This separation of concerns ensures that each module can be developed, tested, and updated independently without introducing regressions in other areas of the application.



3.3 Component Structure

The AISStream.io WebSocket connection lifecycle is managed entirely within the aisstream.js service module. When the Node.js server process starts, server.js invokes startAISStream(), which opens a WebSocket connection to wss://stream.aisstream.io/v0/stream. Upon connection, the service immediately sends a subscription message containing the AISSTREAM_API_KEY from the environment configuration, along with filtering parameters specifying the geographic bounding box and desired message types. The bounding box coordinates [[[24, -130], [50, -60]]] define a region covering U.S. coastal waters from the Gulf of Mexico to the Atlantic and Pacific coasts. The FilterMessageType array limits incoming data to PositionReport and ShipStaticData messages, excluding irrelevant AIS message types such as safety-related broadcasts and aids-to-navigation beacons.

The WebSocket 'message' event handler parses each incoming JSON frame and dispatches it based on the MessageType field. For PositionReport messages, the handler extracts the vessel's current latitude, longitude, speed over ground, course over ground, true heading, and navigational status, then merges these fields into the existing vessel record in vesselMap using the MMSI as the key. This merge operation preserves any previously received static data fields, such as ship name, IMO number, call sign, and

destination. For ShipStaticData messages, the handler extracts vessel identification fields and the AIS type code, derives a human-readable ship category using the `getShipCategory()` helper function, and merges these static fields into the vessel record. The `lastUpdated` timestamp is set to `Date.now()` on every update to support the stale vessel cleanup routine.

The service implements automatic reconnection handling to ensure the AIS data stream remains available in the event of network interruptions or server-side disconnections. When the WebSocket 'close' event fires, the handler checks whether any data was successfully received during the connection (tracked by a `hasReceivedData` flag). If data is received indicating a legitimate connection that subsequently drops, the handler schedules a reconnection attempt after a 10-second delay. The guard condition at the beginning of `startAISStream()` prevents duplicate WebSocket instances by checking whether an existing connection is already in the OPEN ready state before creating a new WebSocket object. This reconnection logic enables the system to automatically recover from transient network failures without requiring manual intervention.

4. Technology Stack

The technology stack was selected to reflect industry-standard tools for real-time, event-driven web applications while remaining accessible for a single-developer project within a 14-week timeline.

4.1 Frontend

1. **React.js:** The frontend framework of choice. Functional components and React hooks (`useState`, `useEffect`) manage real-time state updates triggered by AIS data polling, keeping the UI in sync with live vessel positions without full page re-renders.
2. **Leaflet / react-leaflet:** An open-source interactive mapping library used to render the vessel tracking map. Leaflet's layer system enables clean separation between the base map tiles, vessel markers, and weather overlay layers, each of which can be toggled independently.
3. **OpenStreetMap:** Provides the base map tile layer. Selected for its free tier, global coverage, and compatibility with Leaflet.
4. **Create React App (CRA):** Used as the frontend build tooling with React 19 and JavaScript (no TypeScript). Chosen for stability and to keep type-system overhead off the academic timeline.

I chose React for the frontend for a few specific reasons beyond just its popularity. First, React's virtual DOM handles the frequent UI updates cleanly, since the map updates vessel positions every 30 seconds, and React's reconciliation algorithm only re-renders the changed parts of the DOM instead of redrawing everything. React's component model also made it easy to separate concerns. The map interface, vessel markers, system health dashboard, and data panels each live in their own components. React's extensive ecosystem includes `react-leaflet`, a well-maintained wrapper around the Leaflet mapping library that provides declarative React components for map rendering. This integration proved essential for managing the dual-layer rendering challenge of overlaying weather data on top of live vessel markers while maintaining independent update cycles for each data source.

Leaflet was chosen over alternatives such as the Google Maps API or Mapbox GL due to its open-source licensing, zero runtime costs, and extensive plugin ecosystem. The library's layer management system proved critical for implementing toggleable weather overlays, as each data source (vessel markers, wave height heatmap, wind direction indicators) could be rendered as an independent layer with separate visibility controls. The `leaflet.markercluster` plugin was integrated to handle visual marker clustering for the ~1,400 vessels tracked simultaneously. Without clustering, overlapping markers in congested coastal areas such as the Los Angeles and Long Beach port complex rendered the map unusable at low zoom levels. The cluster group settings were configured with `maxClusterRadius` set to 40 pixels, `disableClusteringAtZoom` set to 12 to separate markers at detailed zoom levels, and `spiderfyOnMaxZoom` set to true to spread overlapping markers when fully zoomed.

4.1.1 React State Management Strategy

The application uses React's built-in `useState` and `useEffect` hooks for all state management, deliberately avoiding external state management libraries such as Redux or MobX. This decision was made based on the application's relatively simple state requirements: sidebar visibility, vessel data panel visibility, map theme selection, and the vessel data array. All states are component-local and do not require sharing across deeply nested component trees, making React's Context API or external state libraries unnecessary overhead. The `VesselLayer` component uses a polling pattern implemented with `useEffect` and `setInterval` to trigger periodic fetch calls to the backend API. The `useEffect` cleanup function clears the interval timer when the component unmounts, preventing memory leaks and unnecessary network requests when the map is not visible.

4.1.2 Leaflet Integration Challenges

Integrating Leaflet with React presented several technical challenges due to the impedance mismatch between Leaflet's imperative DOM manipulation model and React's declarative rendering paradigm. Leaflet was originally designed as a standalone library that directly creates and manages DOM elements for map tiles, markers, and popups. React's virtual DOM reconciliation, by contrast, expects components to declare what the UI should look like based on the current state, allowing React to compute the minimal set of DOM changes needed to reach the target state. The `react-leaflet` library bridges this gap by providing React components (`MapContainer`, `TileLayer`, `Marker`) that wrap Leaflet's imperative APIs behind declarative interfaces. However, certain advanced use cases, such as the marker clustering implementation in `VesselLayer`, still require direct access to the underlying Leaflet map instance via the `useMap()` hook, resulting in a hybrid imperative-declarative programming model. This approach proved workable but required careful attention to component lifecycle timing so Leaflet layers were added to the map only after the `MapContainer` had fully initialized.

4.2 Backend

Node.js: The backend runtime environment. Chosen for its event-driven, non-blocking I/O model, which is well-suited for maintaining persistent WebSocket connections alongside concurrent REST API requests.

Express.js: Provides the REST API layer. A single endpoint (`GET /api/vessels`) exposes the in-memory vessel state to the frontend client.

WebSocket (ws library): Maintains a persistent connection to AISStream.io and receives real-time AIS position updates as they are broadcast.

Node.js was selected for the backend because its event-driven, non-blocking I/O model fits this use case well. The application needs to maintain a persistent WebSocket connection while also handling HTTP REST API requests from the frontend, and Node's single-threaded event loop handles both concurrently with minimal overhead. Using JavaScript on both ends of the stack also reduces context switching during development and keeps the build tooling unified.

Express was chosen to provide the REST API layer due to its minimal overhead and ubiquity in the Node.js ecosystem. The application exposes a single endpoint (GET /api/vessels) that returns a JSON snapshot of the current in-memory vessel state. Express middleware capabilities for CORS and JSON body parsing, simplified development server configuration during the frontend development phase. The ws library for WebSocket client connections provided a lightweight, well-tested implementation with automatic reconnection support built into its event-handling model.

4.2.1 Alternative Backend Approaches Considered

Alternative backend runtime environments were evaluated during the initial design phase. Python with asyncio and the websockets library would have provided similar WebSocket client capabilities, potentially with more readable asynchronous code patterns using async/await. However, Python was rejected due to the additional complexity of maintaining separate runtime environments for the frontend (Node.js for the React build tooling) and backend, along with the serialization overhead of converting Python data structures to JSON for API responses. Go was considered for its strong concurrency primitives (goroutines) and low memory footprint, particularly attractive for applications maintaining thousands of concurrent WebSocket connections. However, Go was rejected because the application's connection requirements are modest (a single WebSocket connection to AISStream.io), and the benefits of Go's performance characteristics did not justify the learning curve and reduced ecosystem familiarity compared to JavaScript.

The decision to use an in-memory Map data structure for vessel state storage, rather than a database, was driven by several factors. First, vessel tracking is inherently a real-time application focused on current positions rather than historical analysis. Operators viewing the map care about where ships are now, not where they were hours or days ago. Second, the active vessel count within the monitored North American bounding box peaks at approximately 1,400 vessels, resulting in a memory footprint well under 10 MB even with full AIS metadata stored per vessel. Third, eliminating database query latency from the critical path between AIS message receipt and API response delivery keeps the system responsive. A database would introduce 10-50 milliseconds of query overhead per API poll, compared to sub-millisecond Map lookup times in memory. Alternative approaches using Redis for in-memory caching with optional persistence were considered but rejected as over-engineering for a single-user demonstration application with no high-availability requirements.

4.3 External APIs

AISStream.io: Provides real-time Automatic Identification System vessel data via WebSocket. Each message contains vessel MMSI, position (latitude/longitude), speed over ground, course, heading, and navigation status.

Open-Meteo Marine API: Supplies significant wave height and swell direction at 20 ocean waypoints across the North American maritime zone, drawn from the same global wave model family as NOAA's WaveWatch III. No API key required. Multi-location batch requests return all 20 points in a single response in roughly 2 seconds.

Open-Meteo Atmospheric API: Supplies current 10-meter wind speed and direction at a 5×5 grid of 90 points covering the full AIS bounding box. All 90 points come back in a single batched HTTPS request.

4.3.1 AISStream.io Message Type Filtering

The AISStream.io subscription configuration specifies `FilterMessageTypes`: `['PositionReport', 'ShipStaticData']`, deliberately excluding other AIS message types such as `BaseStationReport`, `AidToNavigationReport`, `SafetyRelatedBroadcast`, and several others defined in the AIS protocol specification. This filtering decision substantially reduces incoming message volume, since the majority of AIS traffic consists of position reports from moving vessels and static data updates when vessels change voyage parameters. The excluded message types (`BaseStationReport`, `AidToNavigationReport`, `SafetyRelatedBroadcast`) are emitted at much lower frequencies than position updates, so filtering them out at the subscription level prevents unnecessary processing on the backend. The excluded message types, while potentially useful for specialized applications (e.g., safety zones, navigational hazards, search-and-rescue coordination), do not meet the core vessel-tracking visualization requirements of this project. The geographic bounding box filter similarly reduces message volume by rejecting AIS transmissions from vessels outside U.S. coastal waters, so the backend only processes data relevant to the displayed map region.

4.3.2 Why Open-Meteo Instead of NOAA

My original spec called for the NOAA Marine Weather API. Three NOAA options were evaluated and ruled out before I landed on Open-Meteo. The first option was `api.weather.gov`, the National Weather Service forecast API. It's designed for point-based forecast queries, with no bulk endpoint. Covering the AIS bounding box would have meant hundreds of individual lat/lon requests per refresh cycle, and open-ocean points frequently return errors because NWS coverage is built around Weather Forecast Offices that don't extend far offshore. The second option was the NOAA ERDDAP WMS endpoint, which historically served WaveWatch III tile data, but the endpoint at `coastwatch.pfeg.noaa.gov` now returns an HTTP 302 redirect to PacIOOS, which then returns an HTTP 404. The service is effectively dead for programmatic use. The third option was NDBC, which distributes individual text files for around 50 buoy stations, but fetching them sequentially or in parallel takes 15 to 30 seconds because of DNS, TCP, and HTTP overhead per station. The first-load latency made the feature feel broken. Open-Meteo solved all three problems with a single migration: multi-location batch requests, ~2-second response time, and no API key. The marine API specifically draws from the same global wave model family as NOAA's WaveWatch III, so the data quality is comparable.

4.4 Deployment & Tooling

Local development environment: The application runs locally with `npm run dev` for the backend and `npm start` for the frontend. Vercel deployment is documented as a future step.

GitHub: Version control and repository hosting. Pull requests were used to structure each major feature addition as a discrete, reviewable unit of work.

5. Implementation

Development was structured around four sequential milestones, each with defined deliverables and success criteria. All four milestones are complete. The following sections describe the work delivered in each.

5.1 Milestone 1: Core Infrastructure & Vessel Tracking (Complete)

Target Date: March 14, 2026 - Status: Complete

Milestone 1 established the foundational infrastructure of the Naval Intelligence Platform. The primary deliverable was a functioning real-time vessel-tracking system displayed on an interactive map.

5.1.1 Map Interface

The first implementation task was to create an interactive geographical map using Leaflet and react-leaflet. The map supports both light and dark tile modes, toggled via a button in the map interface. Leaflet's tile layer system was used to swap between CARTO's light and dark basemap tiles without re-initializing the map component.

5.1.2 AIS Integration & Vessel Tracking

The AISStream.io WebSocket connection was implemented in the Node.js backend. Upon connection, the server subscribes to a bounding box covering US coastal waters and receives a continuous stream of vessel position messages. Each incoming message is parsed and used to update an in-memory vessel registry keyed by MMSI. The frontend polls the GET /api/vessels REST endpoint every 30 seconds, retrieves the current vessel snapshot array, and updates React state, triggering re-renders of vessel markers on the map.

Vessel markers are rendered as custom SVG icons that rotate based on each vessel's reported heading, providing an at-a-glance directional indicator for each tracked ship. Clicking a marker opens a basic information pop-up displaying the vessel's MMSI, name, speed, course, heading, and navigation status.

The vessel marker update logic uses a diffing algorithm to avoid unnecessary DOM manipulation. When new vessel data arrives from the API, the code first builds a Set of MMSI identifiers from the new data. Then it loops through the existing markers (stored in a React ref so they persist across renders) and removes any whose MMSI isn't in the new snapshot. Those vessels either stopped transmitting or left the area we're monitoring. For vessels present in both the old and new snapshots, the component updates the marker's position, icon rotation, and popup content in place without destroying and recreating the marker DOM elements. For newly appeared vessels, the component creates new marker instances and adds them to the marker cluster group. This differential update approach prevents the

visual flickering that would occur if all markers were cleared and recreated on every poll cycle, and reduces browser layout thrashing by batching DOM updates.

The vessel icon rendering implementation uses Leaflet's L.divIcon API to create custom SVG markers rather than traditional image-based icons. Each icon consists of an SVG triangle element embedded in an HTML string, with a CSS transform applied to rotate it based on the vessel's heading value. The AIS protocol defines true heading values in the range 0-359 degrees, with 511 as a sentinel value indicating heading data is not available. The icon creation function implements conditional logic to prefer true heading when available (values < 511) and fall back to course over ground when heading is unavailable. This way, vessel icons always display meaningful directional information even when heading sensors are malfunctioning or not reporting. The triangle shape provides a clear visual indication of vessel direction at all zoom levels, unlike circular markers, which lack directional information.

5.1.3 System Health Dashboard

A status bar component was implemented to display live connection health for the primary data sources: the AISStream.io WebSocket and the weather API. The backend exposes connection status information through GET /api/system/health, which returns a JSON object containing connection state for each dependency along with timestamp and message count metadata. The frontend polls this health endpoint every 5 seconds, updating the dashboard's status indicators in response to state changes. Each indicator displays a colored badge (green for connected/operational, pulsing yellow for connecting, red for disconnected/error) along with a text label describing the current state. This real-time health-monitoring capability enables operators to quickly diagnose data-feed issues and distinguish between frontend rendering problems and backend ingestion failures.

The live log feed is backed by a server-side circular buffer that holds the most recent 150 log entries. It is implemented as a 12-line module (logger.js) that exposes a pushLog(source, level, message) function. Both aisstream.js and weather.js call pushLog on all meaningful events: connection open, close, error, first data received, stale vessel cleanup, periodic stats, weather fetch start, success, error, and cache writes. When the buffer exceeds 150 entries, the oldest entry is removed. This bounded buffer approach prevents unbounded memory growth while retaining enough history to diagnose recent operational issues. The log feed component renders messages in reverse chronological order (newest first) and auto-scrolls when new entries arrive.

5.1.4 Work Products

1. Interactive geographical map with light and dark mode toggle
2. Real-time vessel position markers with heading-based rotation
3. Node.js backend with WebSocket-to-REST architecture
4. Vessel data panel with CSV download support
5. System health status bar with live log output
6. Merged GitHub PR: feature/geographical-map

Photos of what was accomplished in milestone 1:



ryantpham commented [now](#)

Owner

1. Deleted unused react template files
2. Created gitignore file for security purposes
3. Added leaflet and react-leaflet dependencies
4. Creation of a geographical map with light and dark mode

Work Product:

System mode:



Dark mode:






ryanthpham commented 32 minutes ago
Owner

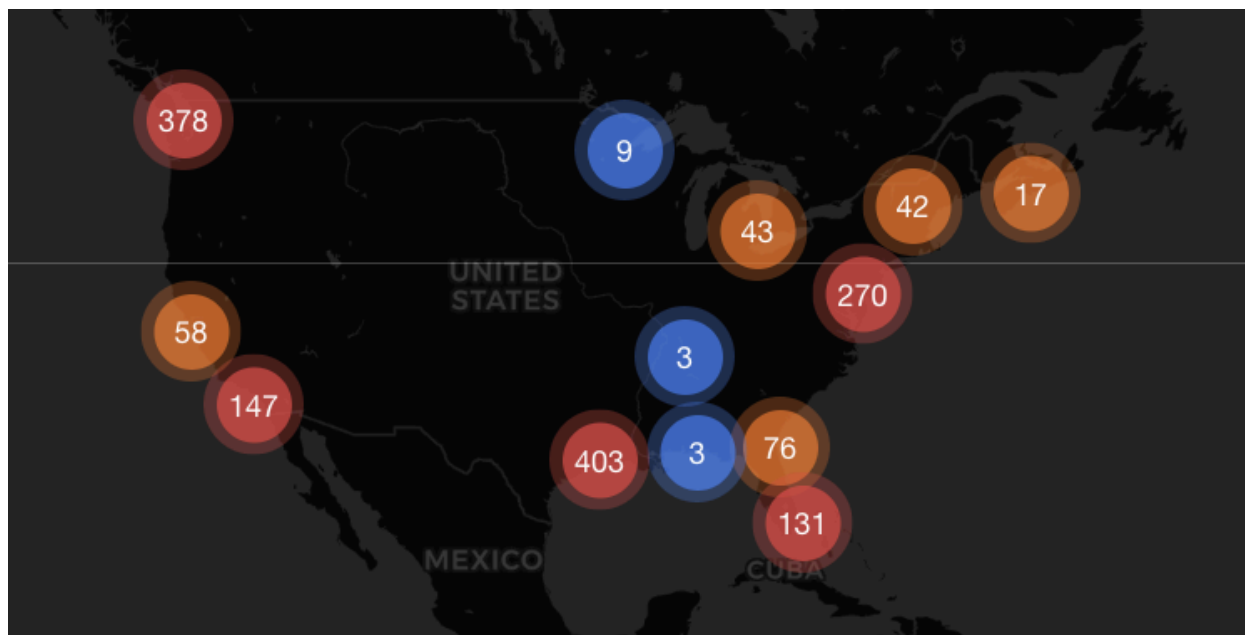
- Add Node.js/Express server with WebSocket connection to AISStream.io
- Server collects real-time vessel data (position, speed, heading) in memory
- Add REST endpoint GET /api/vessels for fetching vessel snapshots
- Add hamburger menu with sidebar navigation and View Vessel Data panel
- Vessel data panel displays tabular AIS data with CSV download support
- Move zoom controls to bottom-right, add map resize handler for sidebar toggle

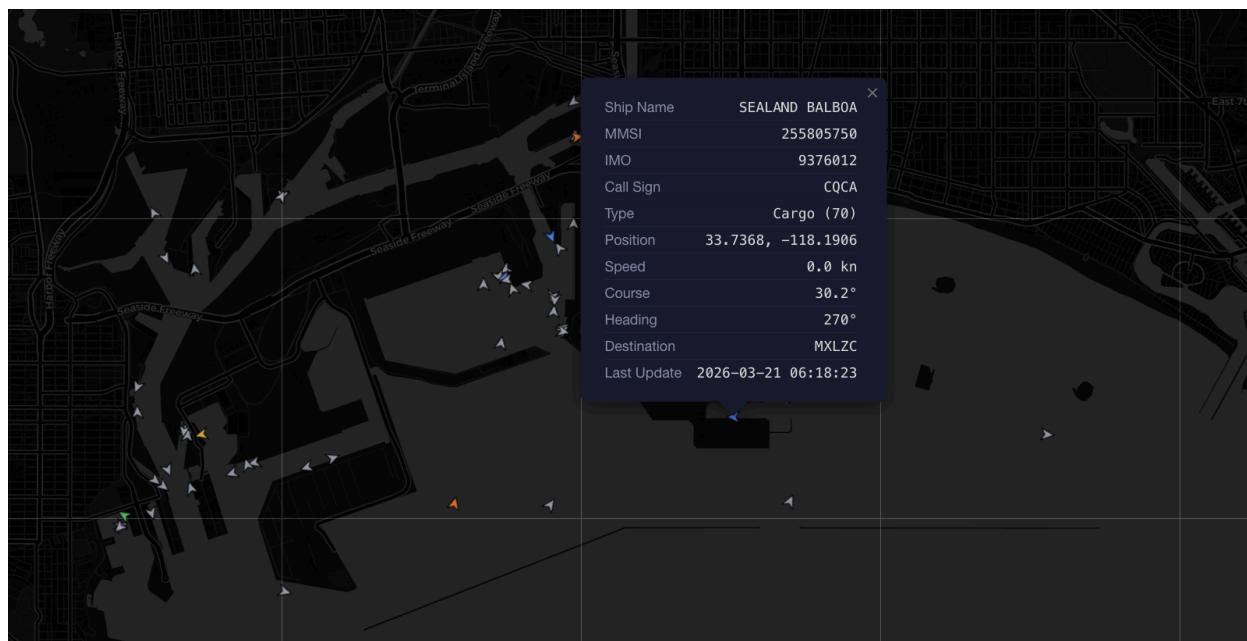
Vessel Data

2311 vessels · 11:30:41 PM

[Download CSV](#) [Refresh](#) [Close](#)

MMIS	SHIP NAME	LATITUDE	LONGITUDE	SOG (KN)	COG	HEADING	NAV STATUS	LAST UPDATED
369810000	DREDGE TEXAS	29.6478	-91.1216	0.0	104.9°	511°	3	2026-02-08 07:09:33.458056480 +0000 UTC
360841400	SAMANTHA S	45.6156	-122.7877	0.1	336.3°	18°	0	2026-02-08 07:10:40.119735855 +0000 UTC
366999979	CG MOBILE BAY	44.8208	-87.3983	0.1	71.4°	511°	5	2026-02-08 07:10:13.533048725 +0000 UTC
316818851	SEASPAR RAVEN	49.8059	-123.1266	0.1	171.8°	48°	0	2026-02-08 07:10:33.456321527 +0000 UTC
368413540	RISEN SON	29.3975	-92.2227	13.9	388.7°	511°	13	2026-02-08 07:09:46.116275202 +0000 UTC
366972260	KENNETH J DEVALL	29.7513	-95.8663	7.2	154.6°	511°	5	2026-02-08 07:10:04.200891036 +0000 UTC
368263020	LOU GOLDING	30.1864	-93.3839	0.0	220.9°	63°	12	2026-02-08 07:10:12.659546669 +0000 UTC
367603210	AMERICAN WAY	30.8220	-93.5871	0.0	43.3°	352°	12	2026-02-08 07:10:05.244389911 +0000 UTC
636822754	LIBRA LEADER	45.6683	-122.7661	9.9	163.3°	163°	0	2026-02-08 07:09:48.160536744 +0000 UTC
367369720	ALTA JUNE	37.9857	-122.3716	0.0	19.3°	235°	0	2026-02-08 07:10:04.243479673 +0000 UTC
367513010	JUSTICE	47.3803	-71.8428	0.0	77.6°	87°	0	2026-02-08 07:09:45.746648647 +0000 UTC
316080255	OCEAN JUPITER	45.5575	-73.5258	0.0	88.1°	218°	0	2026-02-08 07:10:33.481446181 +0000 UTC
303390000	TAURUS	48.5878	-122.7592	7.2	175.2°	181°	0	2026-02-08 07:10:13.774457659 +0000 UTC
367605210	GYRFALCON	47.6535	-122.3637	0.0	105.0°	105°	0	2026-02-08 07:10:24.791328487 +0000 UTC
366964910	J, CAROLE NEALE	39.3510	-81.5480	0.0	219.3°	49°	0	2026-02-08 07:10:03.639312420 +0000 UTC
311000710	NORWEGIAN BLESS	31.9465	-117.4663	19.8	343.4°	343°	0	2026-02-08 07:10:32.046680577 +0000 UTC
367413760	BAY LADY	47.6808	-122.4864	0.0	0.0°	511°	8	2026-02-08 07:10:15.06815179 +0000 UTC
368109530	DARLAN ALEXANDRA	29.9517	-92.8713	0.7	270.7°	256°	12	2026-02-08 07:09:44.647889162 +0000 UTC
368446630	DB CATALINA	29.7550	-95.8945	0.0	15.2°	28°	3	2026-02-08 07:10:15.101437442 +0000 UTC





5.2 Milestone 2: Environmental Intelligence Integration

Milestone 2 layered environmental intelligence on top of the vessel tracking interface and added the filter and search tools that turn the platform from a passive map into a tool operators can use to narrow in on what they care about.

5.2.1 Wave Height Overlay

The wave height layer renders color-coded geographic circles using Leaflet's L.circle at an 800 km radius across 20 confirmed ocean waypoints in the North American maritime zone. The circles overlap and blend, producing a continuous heatmap-like effect that scales correctly with map zoom. Wave height is classified into six conditions: Calm under 0.5 m, Light under 1.5 m, Moderate under 2.5 m, Rough under 3.5 m, Very Rough under 5.0 m, and Dangerous at 5.0 m or above. Each condition maps to a distinct color from sky blue to red. Hovering any circle shows a tooltip with the exact wave height in meters, the condition label, and the swell direction in degrees with compass bearing.

5.2.2 Why L.circle Instead of a Heatmap Library

I initially prototyped this layer with leaflet.heat, a canvas-based kernel density heatmap library. leaflet.heat renders into a viewport-sized canvas element positioned at the map's top-left corner. With the map centered over the Atlantic at zoom level 3, all 20 North American ocean points appeared in the leftmost quarter of the visible screen, and the heatmap blobs got clipped at the canvas boundary, producing a quarter-circle artifact instead of a full ocean overlay. Switching to L.circle solved this cleanly. Leaflet projects and draws each circle relative to its actual lat/lon anchor, regardless of where it appears on screen. The circles scale naturally with zoom, they overlap to blend colors, and they're properly clipped at the world boundary rather than at a canvas edge.

5.2.3 Wind Direction Overlay

The wind layer renders rotated SVG arrow markers at 90 grid points across a 5°×5° lattice spanning the full AIS bounding box. Arrow color encodes wind speed: green at 8 m/s or below, orange between 8 and 15 m/s, and red above 15 m/s. Each arrow pulses with a staggered CSS animation, where the animation delay is driven by a per-marker CSS custom property (--delay) so the layer reads as a live, flowing field rather than a static set of icons. Hovering any arrow shows the wind direction in degrees and speed in m/s.

5.2.4 Weather Data Panel and Filter Vessels Panel

The Weather Data button in the sidebar opens a modal with two tabs: Wave Height and Wind Direction. Both tabs render a sortable table with coordinates, the relevant measurement, and a color-coded condition badge, plus CSV export. A separate Filter Vessels panel provides six independent map filter

controls: vessel category (19 AIS types), movement status (underway / stationary), speed range, destination (with port quick-chips), navigational status, and geographic region (Pacific / Gulf / Caribbean / Atlantic). Filters persist when the panel is closed and re-apply on every 30-second refresh.

5.2.5 Server-Side Cache and Lazy Fetching

Weather data is fetched server-side and cached for one hour rather than letting the browser call Open-Meteo directly. Multiple users share a single cached fetch, and CORS isn't a concern because server-to-server requests don't have CORS restrictions. The fetch is also lazy on the client side: WeatherLayer tracks whether a fetch has fired using a useRef guard and only requests data the first time a weather toggle is turned on. Inside the server, the wave and wind requests run in parallel via Promise.allSettled, so either can fail independently without blocking the other.

5.2.6 Work Products

1. Wave height overlay rendered with L.circle across 20 ocean waypoints, with hover tooltips and a six-condition color scale
2. Animated wind direction overlay across 90 grid points with speed-based color coding
3. Tabbed Weather Data panel with CSV export for both wave and wind data
4. Server-side weather proxy with 1-hour cache and lazy client-side fetching
5. Filter Vessels panel with six independent map filter controls (category, movement, speed, destination, navigational status, region)
6. Reorganized sidebar with three labeled sections (Vessel Data, Weather Overlays, Health and Status)

Photos of what was accomplished in Milestone 2:

ryanpham commented now Owner ...

Filter Vessels panel (Milestone 2) — 6 filter controls:

1. Vessel Category — 3-column checkbox grid for all 19 AIS vessel types, each color-coded to match the map markers. Clicking a checked category dims it out. "All" / "None (show all)" quick buttons.
2. Movement Status — pill toggle between All Vessels / Underway Only (SOG > 0.5 kn) / Stationary Only
3. Speed Range — min and max speed inputs in knots with a clear button
4. Destination — text input with partial case-insensitive match, plus one-click quick chips for 8 major ports (Seattle, Houston, Miami, New York, etc.) — clicking a chip fills the input, clicking again clears it
5. Navigational Status — 2-column checkbox list for the 8 most common AIS nav codes (Underway by Engine, At Anchor, Moored, etc.)
6. Geographic Region — pill toggle between All / Pacific / Gulf of Mexico / Caribbean / Atlantic (same regions as Fleet Analytics)

Filter Vessels 2 active Reset All Close

VESSEL CATEGORY Select All Clear All

☒ Cargo	☒ Tanker	☒ Passenger
☒ Fishing	☒ Towing	☒ Tug
☒ High Speed Craft	☒ Military	☒ Sailing
☒ Pleasure Craft	☒ Search & Rescue	☒ Pilot Vessel
☒ Law Enforcement	☒ Dredging	☒ Diving
☒ Port Tender	☒ Medical	☒ Other
☒ Unknown		

MOVEMENT STATUS

All Vessels **Underway Only (SOG > 0.5 kn)** Stationary Only (SOG ≤ 0.5 kn)

SPEED RANGE (KNOTS)

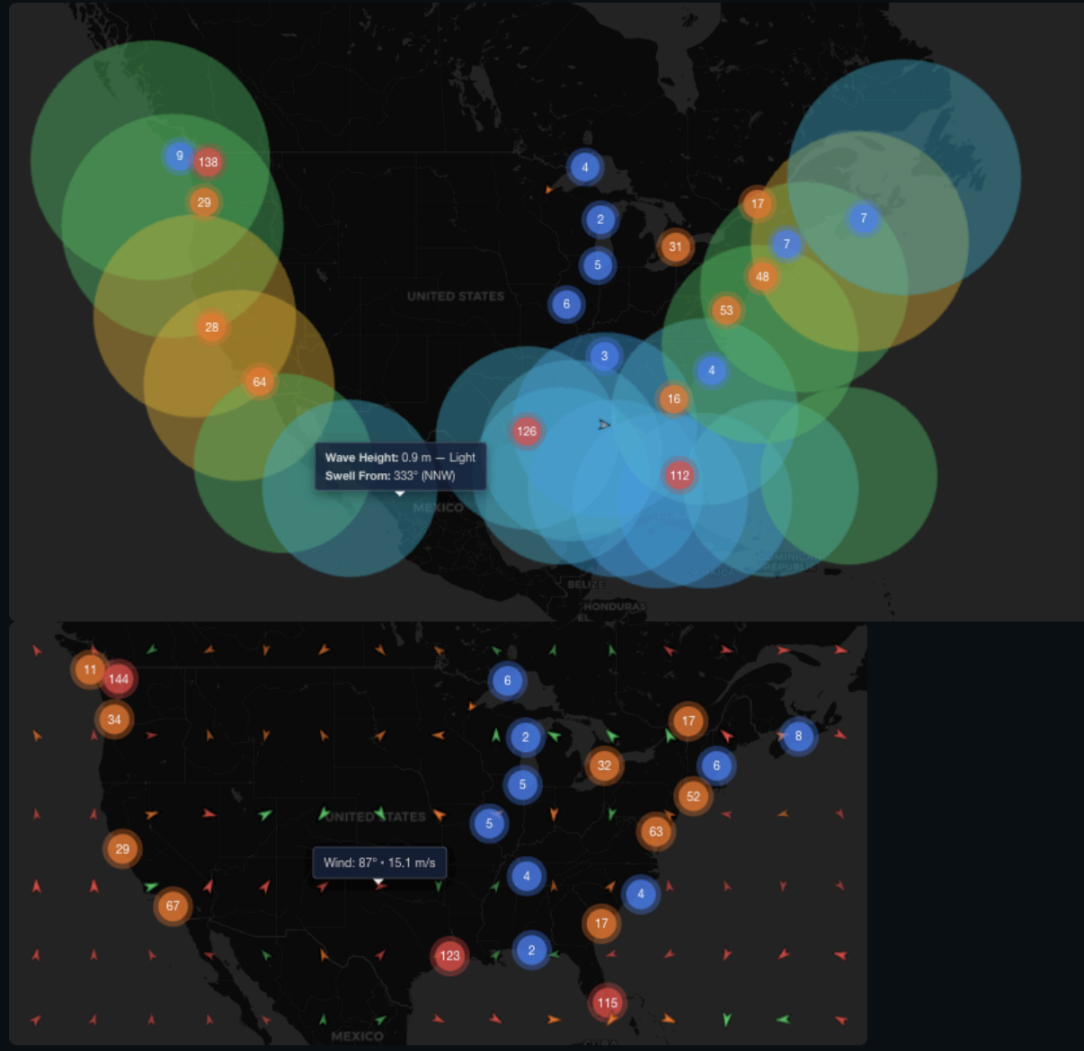
MINIMUM: kn — MAXIMUM: kn Clear

DESTINATION
Partial match, case-insensitive — e.g. "seattle", "houston", "miami"

SEATTLE HOUSTON MIAMI NEW YORK LOS ANGELES
NEW ORLEANS TAMPA BOSTON

Provide real-time weather overlays displaying wave height heatmaps and wind direction indicators

- Implement wave height overlay using L.circle with condition-based coloring
- Add animated wind direction overlay with speed-based styling
- Integrate Open-Meteo APIs with server-side caching (1-hour TTL)
- Build weather data panels with CSV export
- Add vessel search, filtering, and result counter
- Enhance sidebar with weather controls and lazy data fetching



5.3 Milestone 3: System Health and Live Monitoring

Milestone 3 added live observability into the platform itself. A new System Monitor panel, accessible from the Health and Status section of the sidebar, polls `GET /api/system/health` every 5 seconds and displays three sections. The first is API status cards: one card per backend service with a colored status dot, plain-English status label, and key stats (vessel count, message count, last fetch time, data point counts). The second is a live vessel count by category, rendered as a horizontal bar chart sorted by count and color-matched to the map markers. The third is a live log feed: a scrollable, newest-first list of log entries from both the AIS and weather services, color-coded by source and level, holding the last 150 entries from the server-side circular buffer.

A proper production system would route logs through a structured logging library to a log aggregation service. For this platform, that would be significant infrastructure overhead for a feature whose purpose is to demonstrate live observability, not to provide production-grade log retention. The in-memory circular buffer gives the same user-facing result with a single 12-line module. The tradeoff is that logs reset on server restart and aren't queryable, which is fine for the academic context.

5.3.1 Work Products

1. System Monitor panel with three sections: API status cards, live category bar chart, and live log feed
2. Server-side 150-entry circular log buffer in a `logger.js` service module
3. New `GET /api/system/health` endpoint returning a single combined health snapshot
4. 5-second polling cycle on the System Monitor panel
5. Color-coded status indicators for the AIS WebSocket and weather API

ryantpham commented now

Owner ...

Milestone 3: System Operations

- Add SystemInfoPanel with API status cards, live vessel count by category bar chart, and live log feed (polls every 5s)
- Add shared logger service with 150-entry circular buffer written to by AIS and weather services
- Expose getAISStatus, getVesselsByCategory, getMessageCount from aisstream.js
- Expose getWeatherStatus from weather.js
- Add GET /api/system/health route returning AIS status, weather status, vessel counts, and logs
- Add "System Information" button to sidebar under Vessel Data section
- Update PROJECT.md to reflect accurate Milestone 2 implementation (Open-Meteo, L.circle, WeatherDataPanel)

System Information

Updated 11:26:10 PM

Close

AIS STREAM

● Connected

Vessels tracked **1,987**

Messages received **43,651**

Source **AISStream.io**

WEATHER API

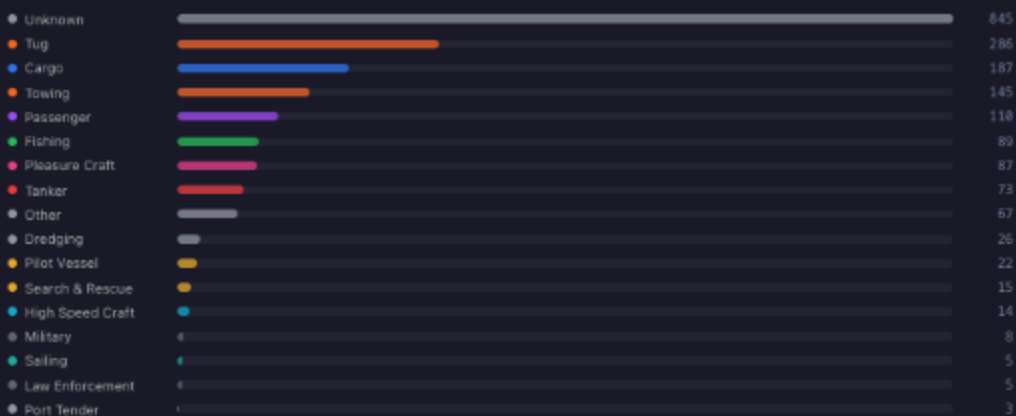
● Ready

Wave points **20**

Wind points **90**

Last fetched **11:18:10 PM**

LIVE VESSEL COUNT BY CATEGORY 1,987 positioned vessels



LIVE LOG FEED 151 entries (last 100)

```

10:52:28 PM[AIS] Active vessels: 1091 | Total messages: 4,109
10:51:50 PM[AIS] Active vessels: 1009 | Total messages: 3,492
10:51:28 PM[AIS] Active vessels: 1673 | Total messages: 2,912
10:50:50 PM[AIS] Active vessels: 1403 | Total messages: 2,317
10:50:28 PM[AIS] Active vessels: 1264 | Total messages: 1,729
10:49:50 PM[AIS] Active vessels: 1002 | Total messages: 1,176
10:49:28 PM[AIS] Active vessels: 531 | Total messages: 569
10:48:59 PM[AIS] Live vessel data stream active
10:48:59 PM[AIS] Connected - subscription sent (bbox: 24-50N, 130-60W)
10:48:50 PM[AIS] Connecting to AISStream.io...

```

5.4 Milestone 4: Fleet Analytics Dashboard

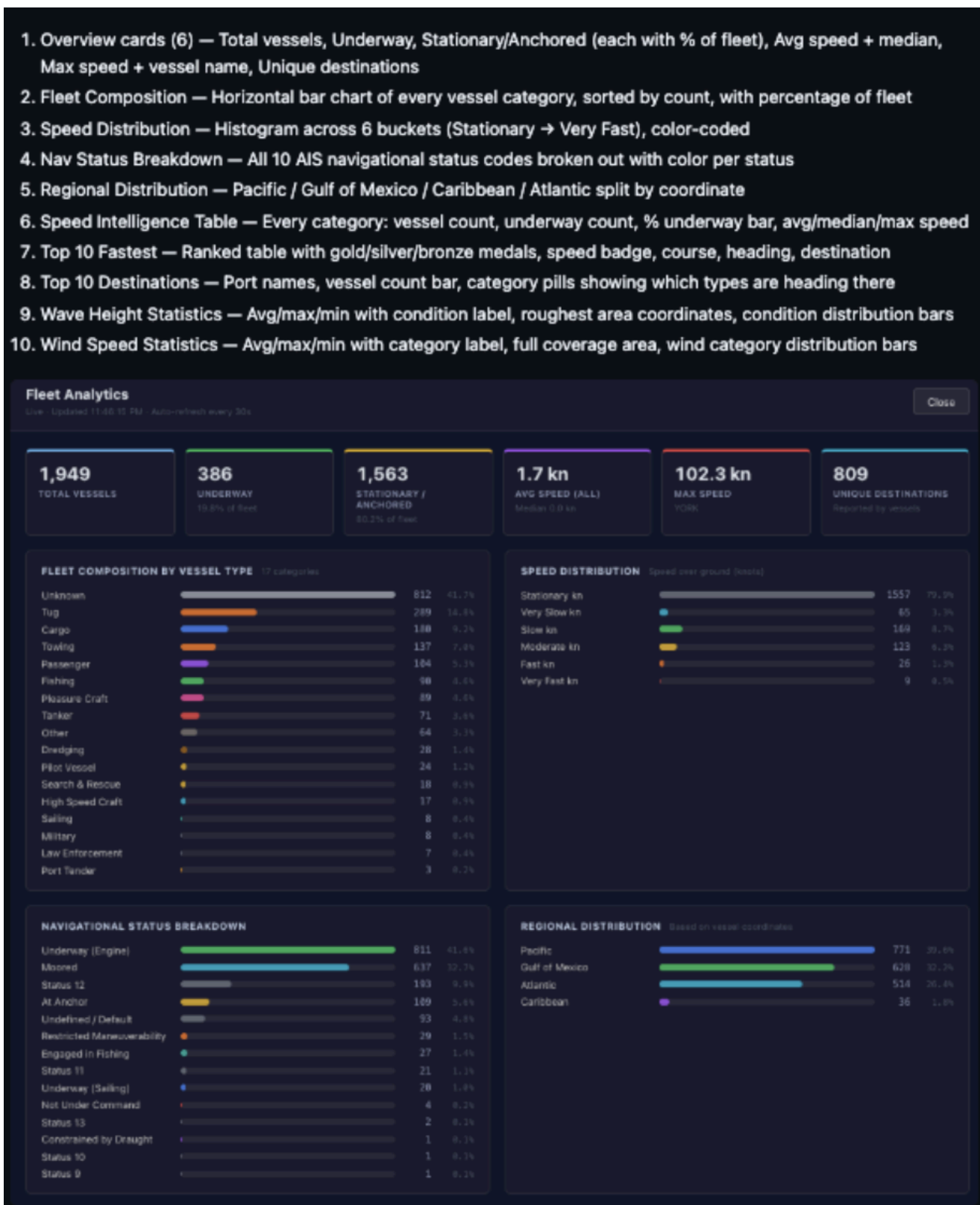
Milestone 4 brought everything together into a single analytics interface. All statistics are computed client-side from the existing `/api/vessels` and `/api/weather/data` endpoints. Vessel data auto-refreshes every 30 seconds; weather data is fetched once per session and served from the server's 1-hour cache. The panel is organized into ten sections: Fleet Overview Cards (six summary cards covering totals, motion, speed, and destinations), Fleet Composition by Vessel Type (horizontal bar chart), Speed Distribution Histogram (six buckets from Stationary to Very Fast), Navigational Status Breakdown (all ten AIS nav status codes), Regional Distribution (Pacific, Gulf, Caribbean, Atlantic by coordinate), Speed Intelligence by Vessel Type (per-category averages, medians, and max speeds), Top 10 Fastest Vessels (ranked table), Top 10 Destinations (ports with category-pill breakdown), Wave Height Statistics (avg, max, min across 20 ocean points), and Wind Speed Statistics (avg, max, min across 90 grid points).

All statistics are computed in the browser from raw vessel data. No aggregation endpoint was added to the server. This keeps the server stateless and simple. Computing 10 sorted groupings and statistical summaries on ~1,400 vessels takes well under 5 ms. If the fleet ever scaled to 10,000+ vessels, a `/api/analytics` endpoint with pre-computed aggregates would be the right migration.

5.4.1 Work products

1. Fleet Analytics panel with ten sections of live statistics
2. Six fleet overview cards covering totals, motion, speed, and destinations
3. Horizontal bar chart of fleet composition by vessel category
4. Speed distribution histogram across six buckets (Stationary to Very Fast)
5. Navigational status breakdown across all ten AIS status codes
6. Regional distribution across the Pacific, Gulf, Caribbean, and Atlantic
7. Per-category speed intelligence table (avg, median, max, % underway)
8. Top 10 Fastest Vessels and Top 10 Destinations ranked tables
9. Wave height and wind speed summary statistics with category distribution charts

Photos of what was accomplished in Milestone 4:



6. Challenges & Lessons Learned

6.1 API Cost Constraints & Feature Scoping

One of the first problems I hit was discovering that historical route visualization required a paid AISStream.io subscription beyond the project budget. The free tier only gives real-time position and no access to historical data. Rather than hack together some complicated workaround for minimal benefit, I just cut historical tracking and spent that time polishing the real-time features. That turned out to be the right call. Milestone 1 ended up more stable and complete than it would've been if I'd split development time between real-time tracking and a half-working historical feature. This was able to reinforce something important: a fully working, polished feature delivers more value than a partially working, ambitious one. Knowing when to cut scope is itself a systems engineering skill.

6.2 Real-Time State Management in React

Managing real-time state updates in React presented a non-trivial engineering challenge. The initial implementation triggered full component re-renders on every AIS polling cycle, which caused noticeable map flickering as all vessel markers were destroyed and recreated every 30 seconds. Resolving this required restructuring the vessel state management to use MMSI as a stable key, enabling React's reconciliation algorithm to identify which markers had changed position versus which could remain in place. After this refactor, map rendering became smooth even with 1,400+ vessels tracked simultaneously.

6.3 Dual Data Layer Rendering

Layering weather data on top of an already live vessel map introduced rendering coordination challenges not present in Milestone 1. The main issue was making sure weather overlay updates didn't interfere with vessel marker updates, especially since the two data sources refresh on completely different cycles, with new vessels processed every 30 seconds, and weather once per hour. Leaflet's layer management system handled this elegantly: each data source becomes an independent Leaflet layer with its own visibility control, and React just orchestrates which layers exist on the map. Once I stopped trying to push everything through React state and let Leaflet own its own layer lifecycle, the integration got noticeably simpler.

6.4 Learning a New Technology Stack

This project represented the first substantial production-oriented application built with React, Node.js, and Leaflet. The initial weeks of development required significant ramp-up time, particularly around React's component lifecycle, hook dependencies, and Leaflet's integration model within a React rendering tree. Leaflet was not designed natively for React's declarative paradigm, and the react-leaflet

abstraction layer introduced its own learning curve. Despite this, the combination proved highly capable for the real-time, geospatial use case this project required.

6.5 WebSocket Reconnection and Error Handling

The first version of the WebSocket code didn't handle errors or reconnections well. When the AISStream.io connection dropped due to a network hiccup or server maintenance, the app would just silently stop receiving data. During early testing, the backend would connect fine initially. But if that connection dropped for any reason, AIS data just stopped flowing with no error message or warning to the operator. The solution involved implementing event handlers for both the 'close' and 'error' WebSocket events, along with automatic reconnection logic that schedules a new connection attempt after a configurable delay. The `hasReceivedData` flag prevents spurious reconnection logging on startup before any messages have arrived, distinguishing between a connection that never succeeded and a previously operational connection that subsequently failed.

An additional challenge emerged related to preventing duplicate WebSocket connections when reconnection logic runs concurrently with manual restarts during development. Without proper guards, multiple reconnection timers could queue simultaneously, each attempting to create a new WebSocket instance when their delay expires. This resulted in multiple parallel WebSocket connections consuming redundant bandwidth and processing duplicate AIS messages. The solution involved two safeguards: first, a guard condition at the beginning of `startAISStream()` that returns early if a WebSocket connection already exists in the OPEN state; second, storing the reconnection timeout handle in a module-scoped variable and clearing any existing timeout before scheduling a new one. These measures keep only a single WebSocket connection alive at any given time, regardless of how many reconnection attempts or manual restart commands are issued.

6.6 Marker Clustering Performance Optimization

Early testing with vessel counts exceeding 20-30 revealed severe performance degradation in the map rendering pipeline, manifesting as 30-second browser freezes during each vessel data update cycle. Profiling with Chrome DevTools identified the bottleneck as excessive DOM manipulation triggered by recreating all map markers on every update rather than performing incremental updates. The naive implementation cleared the entire marker layer and rebuilt all markers from scratch each time new vessel data arrived, causing the browser to destroy and recreate dozens of DOM elements, recalculate layout, and repaint the map canvas 3-4 times per second. This thrashing behavior was particularly visible when vessel icons transitioned between clustered and individual states as the operator zoomed the map.

The solution involved two complementary optimizations. First, integrating the `leaflet.markercluster` plugin with configuration tuned for vessel tracking use cases: `maxClusterRadius` set to 40 pixels to prevent excessive clustering at medium zoom levels, `disableClusteringAtZoom` set to 12 so individual vessel icons appear when the operator zooms to operational detail levels, and `chunkedLoading` enabled

to defer marker rendering across multiple animation frames rather than blocking the main thread. Second, implementing the marker diffing algorithm described earlier that compares the new vessel snapshot against existing markers by MMSI, updating only the markers that have changed position or properties while leaving unchanged markers in place. These optimizations reduced update cycle time from 30 seconds to under 100 milliseconds even with 1,400+ vessels tracked simultaneously, eliminating the browser freeze behavior and restoring smooth map interaction.

6.7 Open-Meteo as a NOAA Replacement

My original spec called for the NOAA Marine Weather API, and that's what I started building against. The reality of NOAA's programmatic offerings was harder than I expected. The NWS forecast API lacks a bulk endpoint and has incomplete offshore coverage. The ERDDAP WMS endpoint I'd planned to use for tile data is effectively dead — it now returns 302/404 chains. The NDBC buoy distribution model is built around individual text files per station, so any meaningful coverage would require dozens of sequential HTTP requests and an initial load of 15+ seconds. Switching to Open-Meteo solved all three problems with a single migration. The lesson here is that vendor selection in real projects sometimes has to follow service availability rather than the spec, and being able to make that switch quickly with clear written justification is more valuable than rigidly sticking to a plan that's no longer feasible.

6.8 Heatmap Canvas Clipping

The wave height layer started life as a `leaflet.heat` heatmap and produced a visually jarring quarter-circle artifact whenever the map was centered over a region without points. Tracing that down to `leaflet.heat`'s viewport-bound canvas placement was a useful reminder that just because a library is "the standard" doesn't mean it's the right tool for every overlay use case. `L.circle` is the simpler primitive, and switching to it not only fixed the bug but produced a cleaner-looking overlay at every zoom level. I now reach for the simplest geometric primitive that satisfies the requirement before reaching for a specialized rendering library.

7. Conclusion

The Naval Intelligence Platform delivered a full-stack maritime surveillance application that meets the complexity and operational demands of real-world defense industry systems. With all four milestones complete, the platform tracks roughly 1,400 vessels across North American waters in real time, layers marine and atmospheric weather overlays from Open-Meteo on top of the live map, exposes a System Monitor panel for live observability into backend services, and computes ten sections of fleet analytics in the browser from the underlying data streams. Each milestone was built directly on the infrastructure of the one before it, and the final integrated build runs the full stack locally with sub-100-millisecond marker update cycles even at peak vessel counts.

Three engineering decisions during development stand out as examples of pragmatic judgment under real constraints: cutting historical route tracking when the AISStream.io free tier could not support it, replacing NOAA weather APIs with Open-Meteo when NOAA's three programmatic endpoints proved unusable, and switching from a heatmap library to L.circle when the heatmap clipped at canvas boundaries. Each of these required making a clear-eyed call to deviate from the original spec in favor of what could actually be delivered. The patterns practiced across the project — event-driven WebSocket ingestion, multi-source asynchronous data fusion, rendering performance optimization at scale, and in-process telemetry feeding a live operations dashboard — map directly to the kind of work done in command center systems at aerospace and defense contractors, positioning the platform as a relevant portfolio artifact for that transition.

8. References

1. AISStream.io, "Real-time AIS data API," 2026. [Online]. Available: <https://aisstream.io/>
2. U.S. Coast Guard Navigation Center, "Automatic Identification System overview," 2026. [Online]. Available: <https://www.navcen.uscg.gov/automatic-identification-system-overview>
3. MarineTraffic, "Vessel tracking documentation," 2026. [Online]. Available: <https://www.marinetraffic.com/>
4. VesselFinder, "AIS vessel tracking services," 2026. [Online]. Available: <https://www.vesselfinder.com/>
5. Open-Meteo, "Marine weather API documentation," 2026. [Online]. Available: <https://open-meteo.com/en/docs/marine-weather-api>
6. Open-Meteo, "Atmospheric forecast API documentation," 2026. [Online]. Available: <https://open-meteo.com/en/docs>
7. National Oceanic and Atmospheric Administration, "National Weather Service: Marine weather forecasting," 2026. [Online]. Available: <https://www.weather.gov/marine/>
8. National Oceanic and Atmospheric Administration, "National Data Buoy Center," 2026. [Online]. Available: <https://www.ndbc.noaa.gov/>
9. National Oceanic and Atmospheric Administration, "Ocean Prediction Center: Wave height and sea state data," 2026. [Online]. Available: <https://ocean.weather.gov/>
10. V. Agafonkin, "Leaflet: An open-source JavaScript library for mobile-friendly interactive maps," 2026. [Online]. Available: <https://leafletjs.com/>
11. react-leaflet contributors, "react-leaflet: React components for Leaflet maps," 2026. [Online]. Available: <https://react-leaflet.js.org/>
12. Leaflet.markercluster contributors, "Leaflet.markercluster: Marker clustering plugin for Leaflet," 2026. [Online]. Available: <https://github.com/Leaflet/Leaflet.markercluster>
13. OpenStreetMap Foundation, "OpenStreetMap," 2026. [Online]. Available: <https://www.openstreetmap.org/>
14. CARTO, "Basemaps: Dark Matter and Voyager," 2026. [Online]. Available: <https://carto.com/basemaps/>
15. Meta Open Source, "React: The library for web and native user interfaces," 2026. [Online]. Available: <https://react.dev/>
16. OpenJS Foundation, "Node.js: JavaScript runtime built on Chrome's V8 engine," 2026. [Online]. Available: <https://nodejs.org/>
17. OpenJS Foundation, "Express: Fast, unopinionated, minimalist web framework for Node.js," 2026. [Online]. Available: <https://expressjs.com/>
18. ws contributors, "ws: A Node.js WebSocket library," 2026. [Online]. Available: <https://github.com/websockets/ws>
19. motdotla, "dotenv: Loads environment variables from a .env file into process.env," 2026. [Online]. Available: <https://github.com/motdotla/dotenv>
20. Mermaid contributors, "Mermaid: JavaScript-based diagramming and charting tool," 2026. [Online]. Available: <https://mermaid.js.org/>

21. GitHub, Inc., "GitHub: Where the world builds software," 2026. [Online]. Available: <https://github.com/>