Workplace Project Supplement - Final Report

Complex Telemetry to Human-Readable Situations: A Lightweight Maritime Observability Dashboard

Ronan Peacock (2768673P)

13th March, 2026.

Executive Summary

This Workplace Project Supplement (WPS) reports on the design, implementation, and evaluation of a lightweight, *real-time situational awareness dashboard* that supports development and debugging during software-in-the-loop (SIL) runs for a UK maritime autonomy programme. During SIL sessions, developers and test engineers need rapid answers to simple but critical questions - whether telemetry is flowing, the current own-ship state, what nearby contacts (other nearby vessels in the traffic stream) are present, and whether information is fresh - without relying solely on deep log inspection or specialist tooling.

The solution consists of (i) **runtime listener components** that subscribe to selected middleware telemetry streams, decode/normalise values into human-readable units, and shape a minimal, stable payload, and (ii) a **browser-based dashboard** that presents own-ship state and nearby contacts with explicit trust cues. Key features include rate-limited UI updates, explicit connection/data-flow status, timestamping and staleness detection, graceful handling of missing data, and lightweight diagnostics. An extension adds **synthetic contact injection controls** for repeatable scenarios in local development, demonstrations, and evaluation.

The dashboard was developed iteratively through successive usable releases, refined based on colleague feedback and SIL constraints (e.g. delayed publishers, bursty message rates, intermittent data). Evaluation used a task-based walkthrough aligned to requirements, complemented by SUS and brief open-ended feedback. Results show core monitoring tasks could be completed successfully and usability was broadly positive, with minor improvement opportunities around making certain UI states (e.g. injection active and stale vs disconnected) more visually unambiguous.

Due to export-control, commercial confidentiality, and security constraints, this report avoids raw code, schemas, identifiers, and operational details, using generic labels and abstracted examples where necessary. Overall, it presents a practical observability tool intended for controlled SIL/development environments, complementing broader SIL reporting with a focused end-to-end software engineering contribution.

Education Use Consent

Contents

1 Introduction

This report is written in the context of Leidos' UK maritime autonomy work and focuses on the Leidos Autonomous Vehicle Architecture (LAVA), an inherited maritime autonomy software stack used in software-in-the-loop (SIL) development and testing. In practical terms, LAVA is a wider runtime environment of middleware-connected autonomy components, simulation support, and engineering tooling, within which developers need to understand what the system believes about own-ship state, nearby contacts, and live data flow during runs. One publicly announced example of the wider application domain is Sea Dagger, a next-generation Commando Insertion Craft concept developed for the Royal Navy [3].

For an organisation working in this domain, a recurring engineering challenge is understanding and explaining what complex autonomy-enabled systems are doing at runtime, particularly during simulation and test. Multiple subsystems rapidly exchange information via message-based middleware, and stakeholders (developers, test engineers, and technical leads) often need quick answers to questions like: "is telemetry flowing?", "what is the current own-ship state?", "what nearby contacts are present?". Improving this "*at-a-glance*" situational awareness increases engineering throughput, supports safer testing, and reduces reliance on deep log inspection during collaborative sessions.

This report addresses that specific software problem through a focused strand of my work: designing and implementing a lightweight, real-time situational awareness dashboard that subscribes to abstracted middleware telemetry and presents a small set of high-value signals (own-ship state and nearby contacts) in a browser, with explicit indicators of connection state and data freshness. The work was carried out primarily independently, but iteratively refined through feedback from colleagues and shaped by operational constraints of the SIL (software-in-the-loop) environment.

The software contribution is a working dashboard that has been trialled as an engineering aid during shared SIL sessions, plus an extension that supports injecting synthetic contacts to create repeatable test and demonstration scenarios. Section 2 provides project background and constraints. Section 3 defines requirements and how they were validated with stakeholders. Section 4 describes the development process and iterative releases. Section 5 details the design and implementation decisions and challenges. Section 6 reports the results of my usability evaluation. Section 7 concludes with limitations and recommendations.

1.1 Confidentiality and abstraction

The work described in this report is based on a UK operational defence project subject to export-control, commercial confidentiality, and security constraints. As university markers do not hold the necessary clearances, I have deliberately:

- Avoided including any raw source code, configuration files, message schemas,

vessel identifiers, or any similar artefact that could reveal sensitive implementation details.

- Replaced proprietary system and process names with generic role-based labels (for example, “autonomy core”, “navigation service”, “middleware subscriber”, etc.).
- Focused on architectural patterns, design decisions and my own contributions, rather than on low-level implementation specifics.

These constraints inevitably limit how much of the underlying system can be shown directly. Where relevant, I reflect on their impact on design choices and evaluation, while aiming to provide as much technical insight as possible within these boundaries.

2 Background

2.1 Project context

The dashboard work described in this WPS sits within a wider maritime autonomy and defence engineering context at Leidos UK & Europe. Public communications describe Sea Dagger as a Royal Navy-aligned Commando Insertion Craft (CIC) concept developed under the UK Commando Force (UKCF) programme, combining modular mission systems with autonomy-related technology and operational resilience considerations [3]. While this report does not describe platform internals, this context helps explain why runtime situational awareness and test-time observability are important: stakeholders need confidence that complex, autonomy-enabled systems are behaving plausibly and that the right information is flowing at the right time.

More broadly, maritime navigation is governed by COLREGs, which formalise obligations around safe conduct and collision avoidance, reinforcing the value of timely situation appraisal [2].

2.2 Stakeholders and working environment

The dashboard is intended to support stakeholders involved in SIL (software-in-the-loop) development and test, including:

- autonomy developers debugging behaviour during runs,
- test and evaluation engineers observing and diagnosing changes in runtime behaviour,
- technical leads using the dashboard during demonstrations or reviews.

The SIL environment is characterised by multiple components communicating over message-based middleware. This introduces practical realities that shaped the work:

- Publishers may start late or stop intermittently.
- Message rates can be bursty (too fast for direct human consumption).
- Partial data may appear during startup, transitions, or fault conditions.

2.3 Why a dashboard and why "abstracted telemetry"

This WPS complements the associated WPA4 report. WPA4 covers the broader SIL workflows and end-to-end data pathways in the autonomy stack. This report focuses on a narrower strand: engineering a standalone runtime observability tool (middleware listeners + web dashboard) as a complete software engineering deliverable, including requirements, iterative development, design decisions, and evaluation.

Because of confidentiality constraints (detailed in Section 1.1), the dashboard is designed around a minimal and stable presentation model: it exposes only user-relevant, abstracted fields to the UI (e.g. own-ship state, contacts in a relative/operator-friendly form, and trust cues such as freshness/staleness). This helps the tool remain useful to stakeholders while reducing risk of exposing sensitive internal structures and reducing brittleness if upstream schemas evolve.

3 Requirements Analysis

3.1 High-level objectives and motivation

The high-level objective of this project is to improve day-to-day observability during software-in-the-loop (SIL) runs by giving engineers a lightweight, real-time situational-awareness view of selected LAVA runtime state. Here, LAVA refers to the inherited maritime autonomy stack and middleware-connected SIL environment within which own-ship and contact telemetry are produced and consumed. In practice, colleagues reported that existing approaches (e.g. log streams and cumbersome tooling) made it difficult to quickly answer basic questions such as "what is the system doing right now?", "is data still flowing?", and "what does the autonomy believe is happening around it?". The dashboard therefore aims to reduce the time and friction required to understand and explain runtime behaviour during development, test, and demonstration sessions.

This document reports on the design, implementation, and evaluation of a lightweight, real-time situational awareness dashboard that subscribes to abstracted middleware data and presents own-ship state and nearby contacts in a form that supports developer understanding and debugging during SIL runs.

Note: the broader SIL environment and autonomy system context (and why observability matters in general) is covered in my WPA4 report and is not repeated here except where needed to justify requirements.

3.2 From objectives to detailed requirements

The initial objective ("*make runtime state easier to understand during mission runs*") was refined into specific requirements through iterative engineering work and stakeholder feedback:

- **Informal stakeholder input** was used to identify the highest-value signals and the common tasks users needed to perform during SIL runs.
- **Prototype use in shared SIL sessions** revealed usability issues (e.g. raw values being hard to interpret) and failure-mode risks (e.g. silent data loss), which were converted into explicit requirements (e.g. derived values, staleness indicators).
- **Operational constraints** (security restrictions, deployment environment complexity, intermittent data availability) were treated as requirements drivers rather than afterthoughts.
- Subsequent extension work was considered only if it directly supported repeatable testing and evaluation scenarios and remained consistent with confidentiality constraints.

The resulting requirements are expressed as **observable behaviours** so they can be verified during SIL runs and evaluated by colleagues.

3.3 Stakeholders and users

1. **Primary users**
    - **Autonomy developers** working in the SIL who need a quick, trustworthy view of runtime state without conducting deep dives into logs.
    - **Test engineers** who observe mission behaviour and need to recognise changes, regressions, or interruptions.
2. **Secondary users**
    - **Technical leads and architects** who may use the dashboard during reviews, demonstrations, or incident triage discussions.

3.4 User needs and core tasks

The dashboard is designed around a small set of high-value tasks that recur during SIL development and test sessions:

- View own-ship state (position, heading, speed).
- See nearby contacts (other vessels) and their relationship to own-ship (expressed in an operator-friendly form rather than raw internal values).
- Determine whether the displayed data is current or stale.
- Gain confidence that the autonomy stack is "alive" without inspecting log streams.
- Use the dashboard to explain system behaviour to others.

These tasks form the basis for the functional requirements and the evaluation tasks used later in the report.

3.5 Functional requirements

The requirements below are written as observable behaviours, so they can be verified during SIL sessions and evaluated by colleagues.

FR1 Own-ship live state display
The dashboard shall display own-ship state (including a position estimate, heading, and speed) and update it in near real time as new middleware samples arrive.

FR2 Nearby contacts display (relative form)
The dashboard shall display a set of nearby contacts and present their relationship to own-ship using operator-friendly relative information (e.g., bearing/range or equivalent), rather than raw internal structures.
Rationale: Relative geometry (e.g. distance and azimuth/bearing) is a practical operator-friendly choice for traffic interpretation [5].

FR3 Data freshness and timestamping
The dashboard shall show the time of the most recent update for each displayed data source (own-ship and contacts) so users can assess freshness.

FR4 Connection and data-flow status
The dashboard shall provide an explicit status indicator for whether it is currently receiving data updates from the middleware pipeline.

FR5 Staleness detection as first-class state
The dashboard shall detect when updates have not been received within a defined threshold (default $\approx$ 10 seconds, configurable) and clearly indicate that the displayed data may no longer be trustworthy.

FR6 Robust behaviour under missing/partial data
If some expected fields are missing or a contact list is temporarily absent, the dashboard shall degrade gracefully (e.g., show blanks/unknowns and status indicators) rather than crash or mislead.

FR7 Rate-limited user interface updates
The dashboard shall rate-limit updates to the browser to a human-readable cadence (configurable), while continuing to ingest backend messages at native rates.

FR8 Lightweight diagnostic indicators
The dashboard shall provide basic health counters (e.g., messages received and UI updates published) to support debugging and confidence-building during runs.

FR9 Browser-based access
The dashboard shall be accessible via a standard web browser to minimise setup overhead for users.

3.5.1 Optional extension (iteration 2) - Synthetic contact injection for testing

If enabled, the tool shall allow users with access to the controlled SIL environment to inject synthetic contact-like updates via the UI (parameterised and/or preset scenarios), clearly distinguish injected contacts from non-injected ones, and support start/stop and clear/remove actions.

The feature is intended for local testing and demonstration and is not designed as an exposed operational interface. This extension will improve local development workflow and enable repeatable evaluation scenarios without relying on external runtime conditions.

3.6 Non-functional requirements

NFR1
Usability and learnability
A *new* user should be able to complete core tasks (e.g., find heading/speed, assess freshness, recognise interrupted data flow) with minimal or no instruction.

NFR2
Reliability and fault tolerance
The dashboard should remain responsive under high incoming message rates, intermittent updates, and transient middleware errors.

NFR3
Performance
The dashboard should present updates with low perceived latency while preventing browser overload via rate limiting.

NFR4

Maintainability and extensibility
The codebase should support adding new fields/widgets with minimal changes, and preserve a stable, minimal data model between backend and UI.

NFR5

Deployability
The tool should be easy to run in typical developer environments (including local development setups), with clear run instructions and minimal operational dependencies where feasible.

3.7 Validation: why these requirements are valuable

The stated requirements are considered validated at a practical level because they directly address stakeholder pain points and were refined based on observed use:

- **Developers and test engineers** need rapid answers to "what is happening now?" during SIL runs; requirements around live own-ship/contact display (FR1, FR2) and browser access (FR9) support this.
- **Trustworthiness** is essential during debugging; explicit freshness and staleness handling (FR3-FR5) supports interruption detection and trust judgement (fresh/stale/connected), with explicit "zero contacts" distinction identified as a future refinement.
- **SIL data can be intermittent**; robustness and rate limiting (FR6, FR7) reflect real operational conditions and prevent the UI from becoming unusable under load.
- **Colleague feedback indicated interpretability matters**; derived, operator-friendly presentation and lightweight health indicators (FR2, FR8) reduce reliance on raw logs while supporting confidence and debugging.

Later in the report, these requirements are further validated through a structured usability evaluation with colleagues (task completion plus a recognised usability instrument), providing evidence that the requirements are not only technically feasible but also useful to stakeholders in realistic SIL workflows.

3.8 Requirements traceability to user tasks

Table 1: Traceability from user tasks to supporting requirements.

User task	Key supporting requirements
View own-ship state during a run	FR1, FR3, FR7
See nearby contacts in a usable form	FR2, FR3, FR7
Decide if data is fresh or stale	FR3, FR5, FR4
Detect mission/system state change	FR4, FR5, FR8
Gain confidence without logs	FR4, FR8, NFR1
Explain behaviour to others live	FR9, NFR1, FR2

3.9 Acceptance criteria (verifiable checks)

- **AC1:** During a SIL run with telemetry available, the dashboard updates own-ship state and contacts without manual refresh.
- **AC2:** Users can identify heading and speed correctly from the dashboard within a short time window.
- **AC3:** If data flow stops, the dashboard transitions to a clear "no recent data / stale" state within ≈ 10 seconds (configurable).
- **AC4:** Under high incoming message rates, the UI remains responsive and updates at the configured human-readable cadence.
- **AC5:** If contacts are genuinely absent, the dashboard indicates "zero contacts" distinctly from "no data received."
- **AC6 (if injection enabled):** Users can inject and remove synthetic contacts, injected items are clearly distinguishable, and clearing removes all injected contacts from the display state.

4 Software Development Process

This project was developed iteratively in parallel with other ongoing development and test activity. Rather than attempting a single "big bang" delivery, I produced successive releases that could be used during real SIL sessions, then refined the implementation based on stakeholder feedback and the operational constraints encountered in practice.

4.1 Development steps and working approach

Although the project was lightweight in scope, the work followed a clear sequence: I framed the problem around "rapid situational awareness during SIL runs" and agreed

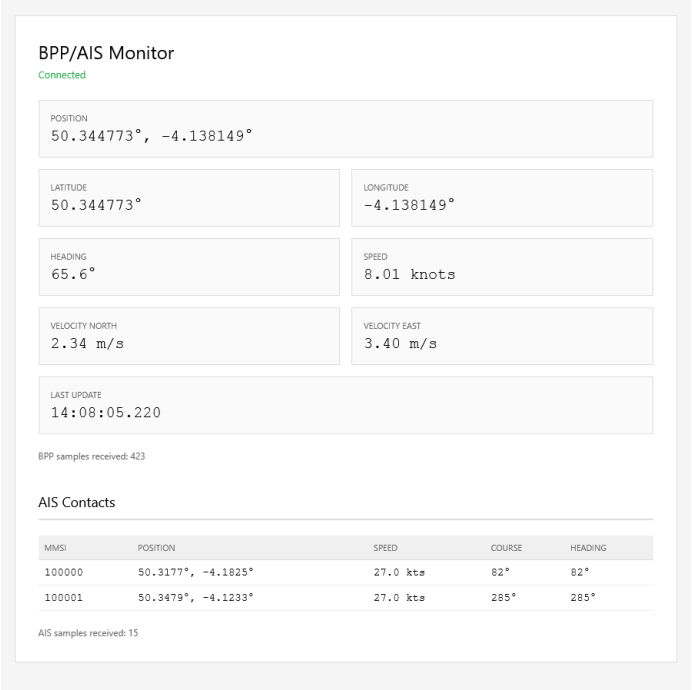

Figure 1: Initial dashboard prototype showing read-only own-ship and contact values

the minimum set of high-value signals; I then built an end-to-end MVP (middleware subscription through to browser display) to validate feasibility; next I trialled the tool in context during SIL sessions to observe interpretation and failure modes; and finally I iterated on trust cues, robustness and usability (staleness signalling, rate limiting, graceful handling of missing data), before adding the injection extension to enable repeatable scenarios for development and evaluation.

4.2 Iterative releases and how the design evolved

The work progressed through successive, usable releases:

4.2.1 Release 1 - Read-only MVP (end-to-end pipeline)

The first release focused on proving the pipeline: consuming runtime middleware data, converting it into meaningful units, and presenting it in a browser. The primary outcome was a minimal interface that demonstrated feasibility and immediate utility. Figure 1 shows this initial prototype. The key elements are the read-only own-ship display and the contact list, intended primarily to confirm that telemetry is flowing.

4.2.2 Release 2 - Interpretability and trust signals

Feedback indicated that raw values were hard to interpret quickly and that it was not always obvious whether displayed values were current. This led to: (a) derived display values with clearer labels, and (b) explicit connection state and staleness indicators so users could judge trustworthiness.

4.2.3 Release 3 - Robustness and operationalism

As the dashboard was used more frequently, the focus shifted to handling real operational conditions: delayed discovery, intermittent publishers, transient read errors, and high incoming message rates. Improvements included defensive handling of incomplete payloads, stable UI update rates, and lightweight health indicators.

4.2.4 Release 4 - Synthetic contact injection (testing and evaluation support)

To enable repeatable test scenarios - particularly for local development and structured evaluation - I added a synthetic contact injection capability with start/stop, preset scenarios, and clear/remove controls. Figure 3 (included in Section 5) shows the current interface including these controls and the visual distinction between injected and observed contacts.

The progression from Figure 1 to Figure 3 provides concrete evidence of iterative development: each iteration responded to stakeholder needs and observed issues encountered during SIL use.

4.3 Stakeholder engagement and feedback format

Stakeholder engagement occurred through short, informal interactions embedded in day-to-day engineering work rather than formal workshops. Typically, I gave brief demonstrations to developers and test engineers to confirm the dashboard answered the questions they would otherwise rely on logs for. I also used the dashboard in shared SIL sessions, which enabled immediate "in the moment" feedback (e.g., clarifying whether a value was interpretable and how to recognise when data had stopped flowing). Finally, I held targeted discussions with engineers familiar with autonomy testing priorities to decide which signals were genuinely useful to expose while keeping the UI focused and safe.

4.4 How the work was embedded in the wider organisation

The dashboard was developed as a supporting tool within a broader SIL environment. This influenced both priorities and constraints:

- **Integration constraints:** the tool had to interoperate with an established middleware ecosystem and run within security constraints (e.g. no external services and no persistent storage).
- **Coordination:** signal selection and interpretation required alignment with colleagues (autonomy developers and test engineers) to avoid incorrect assumptions about what different telemetry represented.
- **Operational relevance:** because SIL sessions are collaborative and time-limited, the tool needed to be quick to launch, easy to understand, and resilient to missing or delayed data.

Overall, the organisational embedding helped ensure relevance: repeated exposure to realistic SIL conditions surfaced practical issues early (discovery timing, update rates, stale data ambiguity) and guided design decisions.

4.5 Independence vs teamwork

The implementation work was primarily carried out independently (designing and building the subscriber/decoding/web components), but it was not isolated. Decisions were shaped by regular interaction with the wider team:

- Colleagues influenced **what information was valuable** to display and how it should be presented to support debugging and explanation.
- Constraints discussed with senior engineers influenced **how the tool could be deployed** and what could be recorded or shown.
- Using the tool during joint SIL sessions provided immediate feedback that informed iterations.

This "independent build, team-informed refinement" model gave clear ownership while still embedding the project in the realities of team workflows.

4.6 What helped and what hindered

Several factors helped the dashboard progress quickly. Early access to realistic SIL sessions meant usability and robustness issues surfaced in context, and feedback from developers and test engineers ensured the dashboard supported multiple stakeholder tasks rather than reflecting a single viewpoint. The confidentiality constraints were also helpful in practice: they forced a minimal, stable presentation model and reduced the risk of over-exposing internals.

The main hindrances were environmental and organisational. Intermittent publishers and complex runtime conditions made purely offline testing difficult, increasing

reliance on defensive coding and synthetic scenarios. Security and deployment constraints limited tooling choices and required additional effort to keep the solution self-contained. Finally, stakeholder time was limited, so feedback had to be gathered opportunistically through concise demonstrations and focused questions.

5 Design and Implementation

This section describes the key designs developed for the project (architecture and user experience), how they evolved through stakeholder feedback, the technologies used to implement the solution, and the principal design decisions and implementation challenges encountered.

5.1 Design overview and evolution

The dashboard began as a minimal, read-only prototype intended to answer a small number of high-value questions during simulation runs: "Is the system alive?", "What is the current own-ship state?", and "What contacts are present?". Early feedback during informal demonstrations and shared SIL sessions indicated that while the prototype was useful, raw numeric outputs were often hard to interpret quickly, and silent failure modes (e.g. data stopping without an obvious indicator) risked misleading users.

As a result, the design evolved in two main ways:

- **Interpretability improvements:** values were normalised into consistent real-world units and simple derived values were introduced (e.g. a user-friendly speed derived from underlying velocity components), reducing the need for users to mentally transform or cross-check numbers against logs.
- **Trust and failure-state signalling:** connection state and staleness were treated as first-class UI states, allowing users to distinguish "connected but not receiving updates" from "disconnected".

A further iteration introduced a **synthetic contact injection** capability. This was driven by the practical need for repeatable scenarios during development and demonstrations (and later, evaluation).

5.2 Technologies used

The implementation is intentionally lightweight:

- **Backend:** Python-based web service using a microframework to serve the UI and provide a runtime endpoint for browser clients.

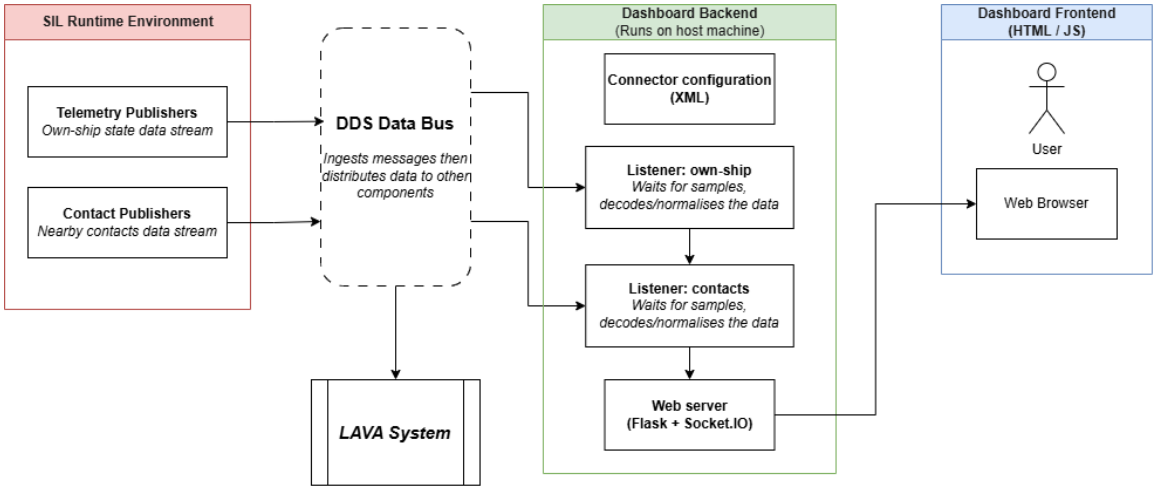

Figure 2: High-level architecture of the situational awareness dashboard

- **Real-time updates:** WebSocket-style communication (via a Socket.IO library) used to push updates from the backend to all connected clients without browser polling.
- **Middleware integration:** a vendor-provided library to join the runtime discovery domain and read/write selected telemetry streams. This connector is an organisational constraint (i.e. an imposed integration mechanism), but how it is utilised (threading, rate-limiting, data shaping) is my own design choice.
- **Frontend:** a single-page HTML user interface styled with CSS and lightweight JavaScript attached that subscribes to WebSocket events and updates the DOM.

Where local development is required without access to the full SIL environment, the implementation also supports a **stand-alone/demo mode** that emits synthetic updates to exercise the UI and server pipeline.

5.3 Architecture design

Figure 2 illustrates the high-level architecture of the dashboard pipeline. The system is structured into three concerns:

1. **Ingestion:** runtime telemetry is acquired from message-based middleware via a connector interface. Separate readers are created for (a) own-ship telemetry and (b) nearby contacts.
2. **Data normalisation and shaping:** samples are converted into a minimal, stable internal representation suitable for UI display (consistent units, derived values, and defensive handling of missing fields).
3. **Presentation:** the backend broadcasts updates to all browser clients over WebSockets, and the browser renders the data as UI elements, with explicit freshness and status indicators.

The main loop in this architecture is the continuous ingestion/normalisation loop for each telemetry stream. Each loop reads samples, drops invalid payloads, converts data into display-ready values, and periodically emits a consolidated update to the browser layer.

This architecture was chosen to keep the browser thin and to concentrate domain-specific encoding and safety checks in the backend, where behaviour is easier to test and standardise.

5.4 Middleware listener layer

The dashboard is underpinned by a middleware listener layer implemented in Python using RTI's Connext ecosystem [4]. DDS (Data Distribution Service) is an open, OMG-managed standard for secure real-time information exchange in distributed systems, and RTI provides a widely used implementation and supporting tooling. For rapid integration, I used RTI Connector for Python, which loads an XML configuration describing the DDS system (domain participant, message topics/types, readers/writers, quality of service rules) and exposes a compact "wait → take samples → iterate" API.

XML configuration and integration boundary. The XML configuration defines two inbound message families (own-ship telemetry and contact lists) and registers them within a single middleware domain. It instantiates a subscriber with two readers (one per data stream) and a publisher with a writer used for synthetic contact injection. The configuration also expresses QoS (Quality of Service) policies (e.g. message reliability/durability and bounded history/resource limits) and transport/discovery constraints appropriate for a controlled SIL networking environment. In this report, I avoid quoting any configuration literals, but the key design point is that the XML acts as a stable integration contract between middleware and the Python application.

Discovery, timeouts, and retry behaviour. DDS discovery is dynamic: publishers may appear late or temporarily disappear. To handle this, the application includes a brief startup pause to allow discovery to converge and then uses a "wait for publications" step per reader with a bounded timeout. If no matching publishers are detected within the timeout, the tool logs a warning and continues rather than exiting, allowing delayed publishers to be discovered later. Connector's API explicitly supports bounded waits for both data arrival and publication matching, which informed this design.

Acquisition loop and resilience. Each reader runs a poll-and-take loop: it waits for data, takes available samples, drops invalid samples, and continues. Timeout and transient middleware errors are treated as expected operational conditions rather than fatal failures: exceptions are caught, logged, and the loop retries after a short delay. This makes the dashboard suitable for long SIL sessions where intermittent publications or transient errors would otherwise cause fragile stop-start behaviour.

Threading model and rate limiting. In the integrated web service, inbound

readers run as background threads (one per stream) alongside a separate publisher thread for synthetic contact injection, while the Flask server runs in the main thread. Because the connector library is treated as non-thread-safe in this design, access is protected with a shared lock that is held only around short "wait/take/write" sections. Updates are emitted to browser clients at a low, human-readable cadence (≈1 Hz) even when incoming middleware sample rates are higher, preserving UI responsiveness without losing semantic information.

Decoding and normalisation. Own-ship telemetry uses packed numeric encodings for position/velocity/altitude. The listener converts these to real-world units (degrees and m/s), normalises angular wraparound, and derives speed from horizontal velocity components. Missing fields are tolerated and treated as absent so the UI can display blanks rather than failing.

Contact semantics and stale-data avoidance. Contact updates are treated as authoritative lists per source. The listener maintains a cache keyed by contact identifiers. On each update from a given source it clears cached contacts from that same source before inserting the new list. This ensures that empty updates correctly clear contacts and prevents "ghost" entries from lingering in the UI - an important correctness property for situational awareness displays used during debugging.

5.5 UX design and interaction design

The user experience was designed around the core SIL tasks defined in the Requirements section. The interface is deliberately simple:

- **Own-ship panel:** compact numeric tiles showing position estimate, heading and speed, intended to be readable at a glance.
- **Contacts table:** a live list of nearby contacts with a stable ordering to reduce visual "jumping" during updates.
- **Status indicators:** a clear connection indicator and a separate freshness/staleness indicator so users can judge trustworthiness of the displayed values.
- **Injection panel (extension):** controls to add and remove synthetic contacts, load preset scenarios, start/stop publishing, and clear all injected contacts.

Figure 3 shows the current interface: the status and freshness indicators are more prominent, values are presented in normalised units, and an injection panel has been added to support repeatable test scenarios.

5.6 Key design decisions and justification

This subsection focuses on genuine design choices (where there was freedom to choose), rather than organisationally imposed elements such as the underlying middleware connector.

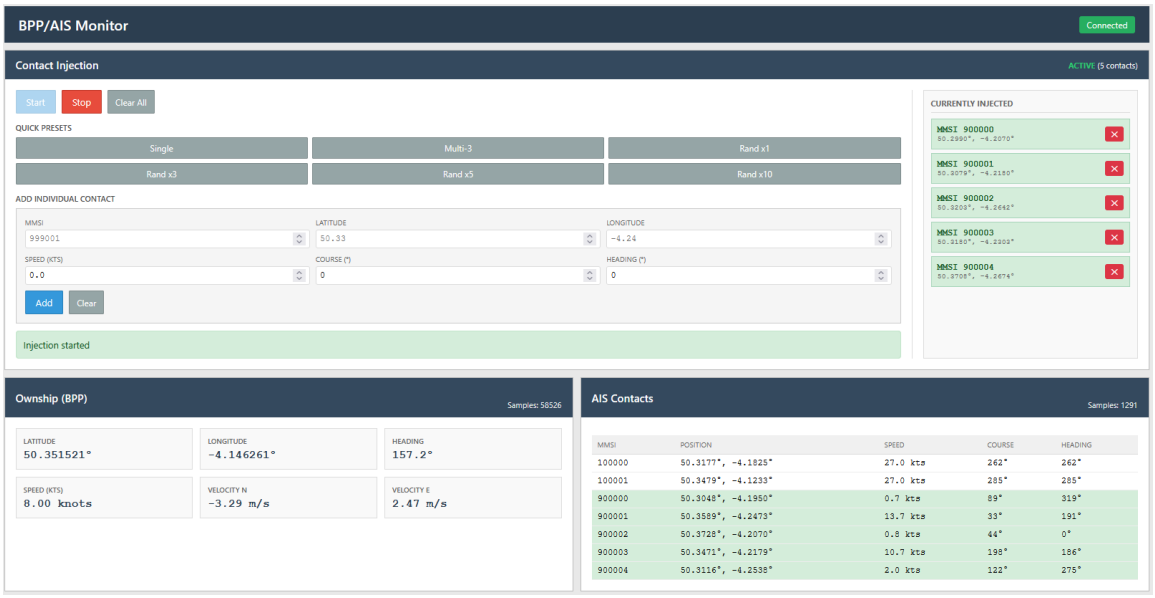

Figure 3: Current dashboard interface including synthetic contact injection controls

5.6.1 XML-defined middleware integration boundary and QoS strategy

- **Decision:** The DDS participant, readers (own-ship and contacts), and writer (synthetic contact injector) are all defined in an XML Connector configuration, with QoS set at this boundary [4].
- **Alternatives considered:** Hard-code entities/QoS in Python; rely on default QoS; use a lower-level DDS API instead of RTI Connector.
- **Justification:** An XML-defined boundary keeps middleware integration explicit and stable, and allows QoS tuning for SIL conditions without invasive code changes.

5.6.2 Threading model and connector safety

- **Decision:** Background reader threads are used for own-ship and contacts, with a shared lock around connector access. The web server remains responsive in the main thread.
- **Alternatives considered:** Single-threaded polling loop; asynchronous event loop; separate process per stream.
- **Justification:** The connector API is *not* thread-safe. A single lock around short critical sections keeps concurrency manageable while still allowing independent reader loops and a responsive UI service.

5.6.3 Server push vs polling

- **Decision:** WebSocket-style server push is used to deliver updates to the browser.

- **Alternatives considered:** Browser polling of REST endpoints at fixed intervals.
- **Justification:** Push-based updates reduce unnecessary load and provide lower latency and clearer semantics for when "data has changed", particularly when upstream update rates are bursty.

5.6.4 Rate-limited UI updates

- **Decision:** Outbound updates to the browser are throttled to a low, human-readable cadence (approximately 1 Hz, configurable), even if middleware samples arrive faster.
- **Alternatives considered:** Emit every sample; adaptive rates based on client load; skip every N samples.
- **Justification:** The dashboard is human-facing. Throttling provides a stable, readable interface and avoids flooding the browser and WebSocket layer.

5.6.5 Staleness detection strategy

- **Decision:** Staleness is tracked client-side using the time since the last update (stale after $\approx$ 10 seconds, configurable) to distinguish "connected but stale" from "disconnected".
- **Alternatives considered:** Server-side heartbeat events; explicit "no data" messages; relying on WebSocket ping/pong alone.
- **Justification:** Client-side timing is lightweight, provides immediate user feedback, and catches silent upstream failures even when the socket connection remains active.

5.6.6 Data model shaping and derived values

- **Decision:** Samples are normalised server-side into a compact payload containing only display-relevant fields, including derived values (e.g. speed derived from velocity components).
- **Alternatives considered:** Passing raw middleware structures to the UI; doing all derivations in JavaScript; storing full historical sample sequences.
- **Justification:** Server-side normalisation keeps the UI simple, ensures consistent interpretation of units, supports confidentiality constraints by avoiding exposure of full message structures, and reduces the risk of UI breakage if upstream schemas evolve.

5.6.7 Contact list maintenance (preventing "ghost contacts")

- **Decision:** Contact updates are treated as authoritative lists per source. When a new list arrives, cached contacts from that source are cleared before repopulating, so empty updates correctly remove contacts.
- **Alternatives considered:** Merge data without clearing; time-to-live expiry rules; per-contact delta semantics.
- **Justification:** Authoritative-list semantics provide deterministic behaviour without requiring timers and reduce operator confusion from stale entries persisting on screen.

5.7 Synthetic contact injection (Iteration 2)

A key extension to the dashboard is the ability to inject synthetic contacts into the middleware for controlled testing and demonstration.

Implementation. Injection is performed via a middleware data writer defined in the same XML Connector configuration as the inbound readers, so publishing occurs within the same domain participation and transport/discovery constraints as subscription.
Scope note: Injection is intended for controlled SIL/development use (local or trusted lab network) and is not intended to be exposed on untrusted networks.

User workflow. The UI allows a user to start/stop injection, add a contact via a form, load preset scenarios, remove individual injected contacts, and clear all injected contacts. The injected set is reflected in the UI and injected rows are visually distinguished from observed contacts to avoid confusion.

Backend strategy. Injected contacts are stored in memory and a background loop periodically publishes a synthesised contact list while injection is active. This periodic publishing is robust to transient middleware issues and ensures downstream components receive updates consistently. The publisher thread uses the same Connector-based middleware interface as the subscribers (with the same connector-safety approach) to write the synthesised list at a controlled cadence.

Safety and cleanup. "Stop" and "clear all" actions publish an empty contact list to flush cached contacts deterministically. This prevents injected artefacts from persisting after users believe they have been removed.

This feature strengthens the tool as an engineering aid: it supports repeatable scenarios (useful for evaluation and debugging) without requiring changes to the wider system.

5.8 Implementation challenges and mitigations

Several practical challenges were encountered when integrating a real-time dashboard into a complex message-based environment:

- **Delayed or missing publishers:** startup waits for publications use bounded timeouts and log warnings when data is not immediately available. Reader loops wait for publications with bounded timeouts and continue running even when none are found, enabling late publishers to be discovered without restart.
- **No samples arriving (yet):** reader loops track whether any valid samples have been seen and can surface a "waiting for data" state rather than failing silently.
- **Transient connector errors and timeouts:** bounded waits are used; timeouts are handled without treating them as data; exceptions are logged and loops paused briefly before retrying.
- **Malformed or incomplete payloads:** missing fields are handled defensively, resulting in blank/unknown UI fields rather than crashes.
- **Client disconnect vs stale telemetry:** the UI distinguishes transport disconnects from "no new telemetry", ensuring users can interpret whether the issue is the browser link or upstream data flow.
- **High inbound message rates:** the primary mitigation is server-side throttling of UI updates, reducing client load while still allowing backend ingestion to operate on native rates.

Together, these mitigations improve reliability and user trust: the dashboard remains responsive and avoids misleading users when data is absent, delayed, or stale.

6 Evaluation

This section evaluates whether the dashboard meets its usability goals: enabling SIL stakeholders to (i) quickly read own-ship state and nearby contacts, and (ii) correctly interpret whether the displayed information is trustworthy (fresh/stale/-connected) with minimal instruction. The evaluation combines a short task-based usability walkthrough with a recognised questionnaire - *the System Usability Scale (SUS)* - and open-ended feedback questions.

Full details of the evaluation protocol and materials are provided in Appendix A, and the participant responses and scores are provided in Appendix B.

6.1 Evaluation method

Approach. A lightweight usability evaluation was conducted using:

- A **task-based walkthrough** aligned with the core user tasks defined in the Requirements (Section 3).
- The **System Usability Scale (SUS)** [1] as a recognised, standardised usability instrument.
- No personal data was collected; responses were anonymised and recorded only as brief notes for the purposes of this evaluation.

This combination provides (a) direct evidence of whether the dashboard supports its core monitoring tasks and (b) a lightweight usability indicator to complement the task outcomes.

Participant and setting. One colleague (anonymised as P1) completed the evaluation. The session was run as a short guided walkthrough using a running instance of the dashboard. The participant was encouraged to describe what they were doing and why, and the observer recorded task outcomes and notes. The evaluation intentionally avoids capturing sensitive operational details; only abstracted observations and anonymised feedback are reported.

Tasks. The walkthrough tasks (detailed further in Appendix A) covered:

1. Reading own-ship heading and speed.
2. Interpreting a contact's relative position to own-ship.
3. Judging freshness/staleness from dashboard indicators.
4. Detecting a run/system state change from UI cues.
5. Describing expected behaviour during data-flow interruption.
6. Using injection controls to add/remove/clear synthetic contacts.

Measures captured. The following metrics were gathered from the evaluation:

- **Task completion outcome** (pass/fail) and related notes.
- **SUS responses and computed score** (out of 100).
- **Direct feedback** on usefulness, confusion points, and suggested improvements through open-ended questions.

Table 2: Summary of usability task outcomes (P1). Full protocol and responses in Appendix A/B.

Task	Result	Key observation
T1	Pass	Heading/speed found in own-ship panel.
T2	Pass	Contact relationship interpreted correctly from table layout.
T3	Pass	Freshness judged using colour + text indicator consistently.
T4	Pass	State change detected via banner + counters + value changes.
T5	Pass	Correctly explained stale vs disconnect implications.
T6	Pass*	Injection workflow completed; *active state could be clearer.

6.2 Results

6.2.1 Task-based outcomes

Stress-check for AC4 (high inbound rates). The dashboard was exercised using the synthetic/demo mode with inbound updates generated at approximately 100 Hz for 10 minutes while outbound browser emissions remained capped at the configured cadence (≈1 Hz). During this run the browser remained responsive (inputs and rendering) and the update counters increased steadily, indicating the rate-limiting strategy functioned as intended under elevated input rates.

Across the tasks, the participant successfully completed the core monitoring and interpretation activities:

- **Own-ship state discovery (T1):** The participant located and read heading and speed, indicating that key information is placed in an obvious and accessible location.
- **Contacts interpretation (T2):** The participant selected a contact and described its relationship to own-ship using the information presented in the table, suggesting that the contact layout and relative presentation supports situational understanding.
- **Freshness/staleness judgement (T3):** The participant used the dashboard indicators to determine that data was fresh, which directly supports the requirement that stale data should be visible rather than silently assumed trustworthy.
- **Data-flow interruption reasoning (T5):** The participant described how the UI would reflect interrupted data flow and differentiated possible causes, suggesting the dashboard provides actionable cues rather than only raw numbers.
- **Injection workflow (T6):** The participant completed add/start/remove/clear operations and valued the visual distinction between injected and observed contacts. A minor usability issue was noted: it was not always visually obvious when injection was active versus merely configured.

A detailed breakdown of the outcomes and selected excerpts is provided in Appendix B.

6.2.2 SUS score

From the SUS questionnaire responses (Appendix B), the participant's computed SUS score was 80/100 [1]. Given the single-participant sample, this result is reported as a supportive indicator of perceived usability in this session, rather than a generalisable comparison across systems.

6.2.3 Qualitative feedback summary

The open-ended responses (Appendix B) reinforce the quantitative and task-based outcomes.

What was most useful:

- **Freshness indicators:** the combination of colour and text helped the participant quickly assess whether the display could be trusted.
- **Contacts table:** described as compact and readable, supporting quick interpretation.
- **Status/banner cues:** helped state changes stand out during the session.

Minor pain point:

- The injection controls would benefit from a clearer "active" state indication (e.g. stronger button state change, clearer label or icon) to reduce uncertainty.

Suggested improvements:

- A small relative-position visualisation (e.g. a minimap) to complement the contacts table.
- Clearer visual distinction between "connected but stale" and "fully disconnected".
- Hover tooltips or small help text for column meanings for first-time users.

Two representative excerpts were: "*The freshness indicator is green... the colour matches the text.*" and "*Start Injection didn't visually change state... wasn't 100% sure it was active...*".

Table 3: Acceptance criteria coverage and evidence pointers.

AC	Status	Evidence
AC1	Met	Task outcomes + live update behaviour (Sec. 6.2).
AC2	Met	T1 pass (Table 2).
AC3	Met	T5 pass + staleness threshold definition (Sec. 3.5/5.6.5).
AC4	Met	Rate-limiting/throttle mitigation (Sec. 5.8) + stress-check observation (Sec. 6.2.1)
AC5	Not met (yet)	Identified as an interpretation risk; addressed as future work (Sec. 7.3).
AC6	Met	T6 pass* + injection design (Sec. 5.7).

6.3 Discussion

Overall, the evaluation provides evidence that the dashboard meets its primary usability intent: it supports rapid "at-a-glance" comprehension of own-ship and contacts and enables correct interpretation of trustworthiness through explicit freshness and connection states. Table 3 directly shows which of the acceptance criteria were partially or fully met.

A key success is the emphasis on **trust signalling**. The participant relied on the freshness indicators and status cues to make decisions, rather than assuming the last value shown was correct. This is important in SIL workflows where intermittent publishers and transient network conditions can otherwise lead to silent failures and misleading displays.

The main improvement opportunity identified relates to the injection extension: while the workflow is usable, the UI should make the "injection active" state unambiguous. This is a relatively small UX change with high impact on confidence during testing and demonstrations.

6.4 Limitations

Due to confidentiality constraints and colleague availability, this evaluation used a small number of participants (n=1), so results should be interpreted as preliminary. The participant was also a colleague, which may introduce familiarity and social-desirability bias (responses may be more favourable than with a genuinely new user).

The goal at this stage was not statistical generalisation, but early validation that the dashboard is usable for representative SIL tasks and that its failure-state signalling is understandable. Future iterations should repeat the same evaluation protocol with additional colleagues from across both developer and test roles to improve confidence in the findings and to capture a broader range of expectations.

7 Conclusions and Recommendations

7.1 Summary of outcomes and final status

This project delivered a lightweight, browser-based situational awareness dashboard that improves observability during SIL runs by transforming abstracted middleware telemetry into a small set of human-readable indicators. The dashboard provides live own-ship state and nearby contact information, makes data trustworthiness explicit via freshness/staleness and connection cues, and includes basic diagnostic signals to support debugging and confidence-building. The work was developed iteratively: an initial read-only prototype validated the concept in real SIL workflows, and subsequent refinements improved interpretability, robustness to missing or intermittent data, and clarity of failure states.

A key extension introduced a synthetic contact injection capability to support repeatable scenarios for development and demonstration. This strengthens the tool's usefulness beyond passive monitoring by enabling controlled test situations without relying on external runtime conditions.

The usability evaluation reported in Section 6 provides initial evidence that the dashboard supports core tasks successfully and is perceived as usable for its intended purpose, with a small number of focused improvement areas.

7.2 Limitations

Several limitations remain:

- **Evaluation scope:** The usability evaluation involved a small number of participants, so results can only provide early validation rather than broad generalisation. Repeating the same protocol with additional colleagues would increase confidence and capture a wider range of expectations.
- **Operational dependency:** Although the dashboard is designed to tolerate intermittent data and delayed publishers, its effectiveness still depends on the availability and quality of middleware telemetry during runs.
- **Information scope:** The dashboard intentionally presents a minimal subset of runtime information to remain interpretable and safe; this means it will not answer every debugging question without deeper tools and logs.
- **UI nuance:** While trust cues exist, the evaluation surfaced that some states (e.g. "connected but stale" vs "fully disconnected", and injection state) could be made more visually unambiguous.

These limitations are consistent with the project's intent: provide a lightweight, low-friction situational view that complements, rather than replaces, specialist debugging tooling.

7.3 Recommendations and future work

Based on development experience and evaluation feedback, the following future work is recommended:

1. **AC5 mitigation** (UI transparency): explicitly distinguish (i) *No contact samples received yet* ("Waiting for contact data…"), (ii) *Contact data received but list empty* ("0 contacts"), and (iii) *Contact data stale* ("Contacts stale; last update at …"), using the existing timestamp + staleness mechanism.
2. **Improve injection status signalling:** Make "injection active" unambiguous (e.g. a persistent ON/OFF indicator and clearer button state changes), and provide brief inline helper text to reduce uncertainty for first-time users.
3. **Add lightweight UI guidance:** Tooltips or short help text for contact table columns and status indicators to improve learnability without increasing clutter.
4. **Extend evaluation:** Repeat the same task protocol with additional stakeholders across development and test roles, and re-run after UI refinements to measure improvement.
5. **Strengthen local development support:** Continue improving a repeatable local test harness (stub publisher / synthetic scenario mode) to enable safe refactoring and extension work without relying on SIL environment availability.
6. **Incremental feature growth:** If additional telemetry fields are added, preserve the principle of minimal, stakeholder-driven information and maintain a stable, abstracted data model between backend and UI.

Overall, the work demonstrates an end-to-end software engineering contribution: a valuable observability tool that is practically usable within controlled SIL/development environments, shaped by stakeholder feedback and evaluated using a structured usability approach.

References

[1] John Brooke. Sus: A “quick and dirty” usability scale. In Patrick W. Jordan, Bruce Thomas, Ian L. McClelland, and Bernard Weerdmeester, editors, *Usability Evaluation in Industry*, pages 189–194. Taylor & Francis, London, UK, 1996.

[2] International Maritime Organization. Convention on the international regulations for preventing collisions at sea, 1972 (colregs). https://www.imo.org/en/about/conventions/pages/colreg.aspx, 1972. Overview and status page.

[3] Leidos UK & Europe. Leidos unveils sea dagger design to advance uk maritime autonomy capabilities. https://www.leidos.com/insights/leidos-unveils-sea-dagger-design-advance-uk-maritime-autonomy-capabilities, 2025. Press release, 4 Sep 2025.

[4] Real-Time Innovations (RTI). Dds standard. https://www.rti.com/products/dds-standard, 2026. Accessed 9 Jan 2026.

[5] Xinyu Zhang, Chengbo Wang, Yuanchang Liu, and Xiang Chen. Decision-making for the autonomous navigation of maritime autonomous surface ships based on scene division and deep reinforcement learning. *Sensors*, 19(18):4055, 2019. Received 19 Aug 2019; accepted 17 Sep 2019; published 19 Sep 2019.

Appendix A: Usability evaluation framework and protocol

A.1 Aim

Evaluate whether the situational awareness dashboard enables SIL stakeholders to: (i) quickly read own-ship state and contacts, and (ii) correctly interpret data trustworthiness (fresh/stale/connected), with minimal instruction.

A.2 Method summary

- Method: Task-based usability walkthrough + System Usability Scale (SUS) + short free-text prompts.
- Participants: 1 colleague (P1), role: developer/test stakeholder (anonymised).
- Setting: Guided session using the running dashboard instance (values treated as synthetic/abstracted for reporting).
- Data captured: task outcomes + qualitative comments + SUS responses. No personal data recorded.

A.3 Tasks and success criteria

See Table 4.

A.4 Measures

- Task completion: Pass/Fail per task + brief qualitative notes.
- SUS: 10-item SUS questionnaire, scored out of 100.
- Free-text prompts: usefulness, confusion points, desired additions, and one top priority change.

Appendix B: Usability evaluation results (participant P1)

B.1 Task outcomes and participant responses

See Table 5.

B.2 SUS questionnaire responses and score

See Table 6.

Table 4: Task-based usability evaluation protocol.

Task	**Participant prompt**	**Success criteria (observer)**
T1	Identify the current own-ship heading and speed.	Finds and correctly reads heading and speed, or clearly explains inability to locate them.
T2	Choose one contact and describe its relationship to own-ship (relative bearing/range or equivalent).	Selects a contact row and correctly interprets the relationship in the terms shown by the UI.
T3	Decide whether contact data is fresh or stale, and explain how you know.	Uses freshness/staleness indicators (timestamp/status) to justify judgement.
T4	Detect a run/system state change using the dashboard alone.	References observable dashboard cues (banner/status, counters, value changes) rather than assumptions.
T5	Explain what the dashboard indicates during a loss of data flow.	Distinguishes "disconnected" vs "connected but stale/no data" and describes implications.
T6 (optional)	Use injection controls to add or load a preset, start injection, remove one injected contact, then clear all.	Completes workflow; injected items are recognised as visually distinct.

Table 5: Task-based usability results (P1).

Task	**Result**	**Summary of response**	**Selected excerpt (25 words)**
T1	Pass	P1 quickly located the own-ship panel and read heading/speed successfully; noted the values are prominently placed.	"Heading and speed... I see them in the top-left panel... That was easy to find."
T2	Pass	P1 selected a specific contact row and correctly interpreted relative bearing and range; found the table layout straightforward.	"The table layout makes this pretty straightforward."
T3	Pass	P1 judged the data as fresh using the indicator colour and liked the consistency of colour + text for reducing misreads.	"The freshness indicator is green... the colour matches the text."
T4	Pass	P1 detected a state change using multiple cues: top banner state change, counter increment resuming, and small changes in own-ship values.	"Status banner... switched... update counter started incrementing again."
T5	Pass	P1 described expected behaviour under interrupted data flow (freshness degrading, values stopping) and distinguished likely causes (transport vs upstream feed).	"I'd check whether it's a network issue or a sensor feed issue..."
T6	Pass (minor issue)	P1 successfully used injection controls to add/start/remove/clear injected contacts and appreciated visual distinction. Minor pain point: start button state not visually obvious.	"'Start Injection' didn't visually change state... wasn't 100% sure it was active..."

Table 6: SUS responses (P1) and computed score.

SUS item	**Rating (1–5)**
1. I think that I would like to use this system frequently.	5
2. I found the system had a simple design.	3
3. I thought the system was easy to use.	4
4. I do not think that I would need the support of a technical person to be able to use this system.	4
5. I found the various functions in this system were well integrated.	4
6. I thought there was consistency in this system.	4
7. I would imagine that most people would learn to use this system very quickly.	4
8. I found the system very easy to use.	5
9. I felt very confident using the system.	4
10. I did not need to learn a lot of things before I could get going with this system.	3
Computed SUS score (P1)	**80 / 100**

B.3 Open-ended feedback (P1)

Most useful aspects

- Freshness indicators: colour + timestamp combination supported quick trust judgement.
- Contacts table: compact, readable, and relative bearing/range presentation felt intuitive.
- Status banner: made state changes obvious.

Confusing or unclear

- Injection controls: not always obvious when injection was "active" versus merely configured/ready.

Suggested additions

- A small minimap / relative-position visualisation to complement the contacts table.
- Clearer visual distinction between "connected but stale" and "fully disconnected" (e.g., iconography).
- Hover tooltips explaining column headers for first-time users.

Highest priority change

> Improve the visual distinction between stale and disconnected states, as it is a critical operational distinction.