University of Glasgow | School of Computing Science

Workplace Assessment 4 Practice Project - Final Report

Engineering Query and Injection Interfaces for an Inherited Autonomous Maritime System

Ronan Peacock (2768673P)

2nd April, 2026.

Executive Summary

This project was conducted within a UK defence autonomy programme subject to export-control and confidentiality constraints. The UK team inherited a containerised autonomous maritime stack and a software-in-the-loop (SIL) environment whose day-to-day use depended heavily on tacit knowledge, fragile workflows, and specialist familiarity with middleware internals. That made it difficult not only to reproduce runs, but also to interrogate runtime state, construct controlled traffic scenarios, and deliver new software with clear engineering evidence.

The project therefore evolved from general SIL familiarisation into a more focused software engineering contribution: building middleware-facing interfaces and supporting delivery practices that make the inherited system more queryable, more controllably testable, and easier to reason about. The central implemented outcomes were a **DDS Query Service**, which exposed selected autonomy-relevant middleware data as structured JSON, and a **Contact Injection Service**, which allowed synthetic contacts to be introduced into the autonomy-facing data stream for repeatable engineering checks and scenario setup. These service-level outcomes were reinforced by **automated test suites**, **mock modes** that removed dependence on a live LAVA/DDS runtime for routine validation, and a **containerised CI/CD path** using Bamboo, Docker, and Harbor.

The project also retained a smaller but still important enabling strand: documenting a repeatable SIL workflow, clarifying configuration layering, and identifying which runtime signals mattered most when positioning the services inside the inherited stack. Those supporting artefacts reduced integration risk and made the service work reproducible for other engineers.

The report argues that the project's main value lies not in new autonomy algorithms but in stronger engineering interfaces around an inherited autonomy system. Within safe disclosure limits, the evidence shows easier interrogation of middleware state, more deliberate synthetic scenario control, and more credible service delivery through testability and CI. Bamboo feature-branch builds reported 113 DDS Query Service tests passing at 63% coverage and 137 Contact Injection Service tests passing at 65% coverage.

Education Use Consent

Contents

1 Introduction

This project was conducted within the Leidos Autonomous Vessel Architecture (LAVA), Leidos' maritime autonomy software baseline for uncrewed surface vessels. In practical terms, LAVA is not a single program but a wider autonomy stack comprising simulation and operator tooling, middleware-connected autonomy services, and vessel-facing control components used to plan missions, interpret sensor and contact data, and execute vessel behaviour. The UK team inherited access to this stack and its SIL environment, but many routine engineering tasks still depended on specialist knowledge and fragile workflows.

The UK team inherited LAVA, an autonomous maritime software stack and software-in-the-loop (SIL) simulation environment, under export-control constraints, but did not inherit a development experience that made the system easy to interrogate or extend. Reproducing runs required knowledge of specific workflow steps and restart conditions; understanding runtime state required familiarity with middleware internals; and constructing useful test scenarios often meant editing configuration-heavy artefacts whose effect on the autonomy stack was not always obvious. These were not only convenience issues. They directly affected diagnosability, reproducibility, and confidence in software change.

The work reported here addressed that problem as a practice-based software engineering project. Rather than implementing new autonomy behaviours, I focused on creating *middleware-facing engineering interfaces* and supporting artefacts that made an inherited system easier to interrogate, easier to stimulate deliberately, and more testable. The central contribution was a pair of microservices: a **DDS Query Service** for reading selected runtime state from the middleware layer and presenting it through a cleaner API, and a **Contact Injection Service** for writing synthetic contact data into the autonomy-facing side of the stack so that controlled traffic conditions could be exercised in simulation. Those services were then strengthened through automated testing, mock-mode execution, and a branch-driven CI/CD path.

This framing is important. The wider architecture also contained upstream components for spoken-command handling, intent mapping, API brokering, and broker-based integration. Those elements are included in this report only as *system context*. My direct software contribution begins at the microservice layer and includes the surrounding engineering needed to run, test, and deliver those services credibly.

The rest of this report is structured as follows. Section 2 provides the brief background needed to justify the design choices. Section 3 defines the problem and requirements. Section 4 summarises the development process and how the project evolved. Section 5 presents the design and implementation work, with particular emphasis on the two microservices and their test and delivery path. Section 6 evaluates the outcomes against the requirements using bounded, confidentiality-safe evidence. Section 7 concludes and sets out recommendations.

Artefact	Type	Purpose in this report
DDS Query Service	Implemented microservice	Provides a reusable interface for querying selected autonomy-relevant DDS state, reducing direct dependence on middleware expertise.
Contact Injection Service	Implemented microservice	Provides a reusable path for injecting synthetic contacts into the autonomy-facing data stream for controlled testing and scenario construction.
Automated tests and mock harnesses	Test software	Allow both services to be exercised without a live LAVA/DDS runtime, improving local development repeatability and CI viability.
Bamboo/Docker/Harbor delivery path	CI/CD configuration + scripts	Moves service changes through repeatable build, test, and image publication steps, improving engineering credibility and maintainability.
SIL workflow, architecture, and configuration artefacts	Supporting documentation + analysis	Preserve the strongest workflow and inherited-system analysis from earlier project phases so that the services can be run and interpreted consistently.

Table 1: Delivered artefacts and their role in the overall project narrative.

1.1 Direct contribution and delivered artefacts

The direct contributions delivered in this project are summarised in Table 1. The table makes a deliberate distinction between the project's principal engineering outcomes and the supporting artefacts that helped position and validate them.

1.2 Relationship to the separately submitted WPS

This report should be read alongside, but not as a continuation of, the separately submitted Workplace Project Supplement (WPS). Both pieces arise from the same broader LAVA/SIL setting, so some system context and some underlying runtime concepts are necessarily shared. However, the principal software outcome of WPS was a browser-based situational-awareness dashboard concerned with human-readable runtime interpretation and lightweight usability evaluation. The principal software outcome of WPA4 is different: middleware-facing query and injection services, together with the test and delivery discipline needed to make them reusable engineering artefacts.

Where dashboard-style visualisation, operator-facing presentation, or usability questions are relevant background, they are intentionally not expanded here because they belong to WPS rather than to the software product claimed in this report. Conversely, this report concentrates on service boundaries, bounded contracts, repeatable validation, and delivery practices at the microservice layer.

1.3 Boundaries, confidentiality, and evidence strategy

The system discussed in this report sits inside a defence-adjacent engineering context with export-control and commercial confidentiality constraints. As a result, the report deliberately avoids raw code, full configuration files, exact schemas, hostnames, storage locations, vessel identifiers, or other operationally specific details. Architectural descriptions are kept abstract, and runtime evidence is presented only in sanitised or summarised form.

These constraints do not remove the need for engineering evidence, but they do shape what kind of evidence can be shown safely. Throughout the report, I therefore rely on four bounded evidence forms: inspectable service responsibilities and interfaces; supporting workflow and configuration artefacts; sanitised descriptions of runtime behaviour; and explicit requirement-closure discussion that states both what the evidence supports and what it does *not* prove. Where quantitative evidence materially strengthens the report, I use sanitised values taken from Bamboo build summaries and logs rather than disclosing raw internal artefacts.

2 Background

This project sits at the intersection of distributed middleware, SIL simulation, and practice-based software engineering in inherited systems. The background here is intentionally brief and only covers ideas that directly shaped the design choices in Section 5.

2.1 Engineering enablement as a software outcome

In inherited systems, valuable software engineering work often takes the form of *interfaces, tooling, and delivery improvements* rather than new end-user features. Continuous-delivery literature emphasises that reliable build, test, and deployment paths are part of the software product, not separate from it [2, 8]. In the same spirit, a service that turns difficult-to-access middleware state into a bounded, reusable interface can be a meaningful software outcome because it changes how the wider team can interrogate and evolve the system.

2.2 Service boundaries over message-based systems

Observability is commonly framed as the ability to infer internal state from externally visible signals such as logs, metrics, and traces [1, 5]. In publish/subscribe systems, the message surface itself becomes an important part of that signal. This is highly relevant to an autonomy stack built around DDS-style middleware, because useful engineering questions often concern which classes of state are present, how often they

update, and whether selected parts of that state can be surfaced safely to downstream engineering tools without exposing the whole internal implementation.

In this report, the practical implication is not to build a human-facing dashboard but to create *bounded developer-facing service boundaries* over selected message families. That framing motivated two design choices. First, I treated middleware-facing services as a legitimate engineering mechanism for interrogating selected state rather than assuming that direct DDS inspection was the only correct path. Second, I treated earlier lightweight DDS inspection work as a stepping stone toward a cleaner service contract instead of as the final outcome in its own right.

2.3 Controlled experimentation and scenario injection

SIL simulation is useful only when it supports *controlled* experimentation. In both robotics and maritime autonomy research, scenario families and repeatable encounter setups are used to probe how systems behave under specific conditions [3, 9, 4, 6, 7]. For a practice-based project, the relevant lesson is not to reproduce academic evaluation wholesale, but to recognise that repeatable synthetic stimuli are essential when diagnosing an autonomy stack.

This directly motivated the Contact Injection Service. If realistic external traffic cannot be relied upon, then the engineering stack needs a bounded way to create synthetic contacts intentionally, so that developers can test scenario-dependent behaviour without waiting for the "right" conditions to appear accidentally.

2.4 Evaluating under confidentiality constraints

Confidentiality changes the form of acceptable evidence. It is often unsafe to publish full schemas, logs, or raw middleware captures, yet it remains necessary to show that the software is real, testable, and bounded honestly. The evaluation strategy in this report therefore prioritises requirement closure, service responsibility, testability, and delivery discipline over disclosure-heavy artefacts. This is not ideal from a traditional experimental perspective, but it is appropriate to the context and keeps the claims proportionate.

2.5 Inherited-system risk and bounded interfaces

One lesson from the earlier SIL-centred phase of the project was that inherited systems accumulate *diagnostic friction* at their boundaries. In this case, useful runtime state existed, but it was trapped behind middleware-specific knowledge; useful scenario control existed, but it was entangled with heavier simulation and configuration edits; and useful evidence of software quality existed, but it was harder to present convincingly without a more normal build-and-test path. Those are all examples of engineering risk created by weak interfaces rather than by missing algorithms.

That framing influenced the eventual solution shape. A query interface was preferable to asking each developer to become a DDS specialist. An injection interface was preferable to treating every controlled-contact experiment as a bespoke mission-editing exercise. Mockable services were preferable to making the full SIL the only validation environment. In other words, the project's core software contribution was not to replace the inherited stack, but to add *bounded interfaces around the most painful parts of it*. That is why the report focuses so heavily on interface responsibility, controlled scope, and delivery discipline in Section 5.

3 Problem Definition and Requirements

3.1 Problem statement

The central problem addressed by this project was that the inherited autonomy stack was difficult to interrogate and difficult to exercise deliberately. The problem was framed as an engineering-interface problem rather than a human-facing presentation problem: the concern here was not how to display runtime state in a browser, but how to expose and stimulate selected middleware state through reusable service boundaries. Three pain points mattered most.

First, selected runtime state was trapped behind middleware expertise. Engineers who wanted to inspect own-ship position, nearby contacts, or related autonomy-facing state had to understand the relevant DDS surfaces directly, which slowed local debugging and made downstream integration work harder than it needed to be.

Second, controlled traffic stimulation was awkward. Useful autonomy tests often depended on the presence of nearby contacts, but creating those conditions repeatably could require editing simulation and configuration artefacts in ways that were not well aligned with rapid experimentation.

Third, even where useful software had been built, the engineering evidence around it needed strengthening. In a practice-based year-4 project, it is not enough to describe understanding and workflow improvement alone. The project also had to show implemented software, testability, and credible delivery practices.

3.2 Stakeholders

The main stakeholder groups were:

- **UK autonomy developers**, who needed to answer routine engineering questions such as "Where is own-ship now?" and "What contacts are nearby?" without falling back to DDS-only workflows.

- **Technical leads and architects**, who needed reusable service boundaries and build paths rather than one-off personal scripts or undocumented specialist knowledge.
- **Test and integration engineers**, who needed a quicker way to create deliberate nearby-contact conditions and to re-run the same checks across comparable SIL setups.
- **Future maintainers and new team members**, who needed enough workflow and configuration context to position and interpret the services without inheriting tacit runtime knowledge first.

These requirements were derived from repeated engineering tasks carried out during SIL familiarisation, debugging, and integration work rather than from a single formal elicitation workshop. The same bottlenecks recurred across those tasks: selected runtime state was awkward to interrogate without DDS-specific knowledge, deliberate nearby-contact conditions were slow to create through heavier scenario editing alone, and useful service work would not be credible unless it could also be tested and delivered repeatably. Alongside the functional requirements in Table 2, four non-functional concerns shaped the solution throughout: *boundedness* (expose only the smallest safe and useful contract), *reproducibility* (support repeatable service-backed checks), *testability* (exercise most logic without a full live runtime), and *maintainability* (keep the outward contract cleaner than the inherited middleware surface).

3.3 Requirements

The problem statement was refined into the five requirements shown in Table 2. Together they define a project that remains about reproducibility, diagnosability, and controlled experimentation, while still presenting a stronger software engineering outcome. The contribution weighting is deliberate: **R1** and **R2** are the main product outcomes, expressed as read-side and write-side service boundaries rather than user-interface features; **R3** and **R4** are the testability and delivery enablers that make those services credible; and **R5** retains the minimum inherited-system context needed to position and interpret the work coherently.

The prioritisation was also intentional. The highest-value slice of the inherited-system problem was not full exposure of the entire middleware surface or universal scenario authoring; it was a smaller set of bounded read-side and write-side capabilities that answered the most recurrent engineering questions, plus the test and delivery support needed to make those capabilities usable by others.

3.4 Acceptance view

I treated success as *repeatable engineering leverage* that could still be inspected under confidentiality constraints. In practical terms, that meant the project should leave behind software and artefacts that other engineers could use without depending on

Req.	Requirement	Stakeholder task / pain point	Bounded evidence later shown
R1	Deliver a reusable runtime interrogation interface over selected DDS state.	Answer recurring debugging and integration questions about own-ship state and nearby contacts without routine direct DDS inspection.	Bounded query families, sanitised contract examples, and an explicit validation case in Section 6.
R2	Deliver a reusable contact injection interface for synthetic scenario control.	Create a nearby-contact condition deliberately for SIL checks or debugging without repeating heavier mission or configuration edits.	Bounded injection capabilities, sanitised contract examples, and an explicit validation case in Section 6.
R3	Support both services with automated tests that can run without a live LAVA/DDS runtime.	Recheck request handling, validation, and bounded failure behaviour when the full runtime is unavailable or too costly for routine iteration.	Automated suites, mock-mode execution, container smoke checks, and Bamboo test/coverage summaries.
R4	Integrate the service work into a repeatable, branch-driven CI/CD path using containerised build and test steps.	Move service changes through a team-usable build, test, and packaging path rather than local-only execution.	Bamboo branch builds, containerised test execution, and branch-conditioned publication behaviour.
R5	Preserve the strongest inherited-system workflow and configuration artefacts needed to position, run, and interpret the services coherently.	Let engineers exercise and interpret the services without relearning hidden workflow and configuration knowledge first.	Retained workflow/configuration artefacts, architectural placement, and explicit scope boundaries.

Table 2: Requirements, stakeholder tasks, and bounded evidence forms derived from the inherited-system problem framing.

Req.	Marker-visible closure signal	Bounded evidence form later shown
R1	A developer-facing request for selected own-ship or nearby-contact state returns a bounded JSON-style view that is sufficient for comparison, logging, or downstream hand-off.	DDS Query Service contract examples and the R1 validation case in Section 6.
R2	A bounded synthetic-contact request produces a deliberate nearby-contact condition that can be recreated or varied through the same service path.	Contact Injection Service contract examples and the R2 validation case in Section 6.
R3	Both services can be exercised repeatedly in mock mode and container smoke checks, with Bamboo-reported totals and coverage for routine validation.	Test-strategy discussion plus the quantitative Bamboo summary in Section 6.
R4	Service changes pass through branch-triggered build/test steps, with image publication gated to the main branch.	CI/CD pipeline table and Bamboo-derived branch/timing evidence.
R5	Retained workflow and configuration artefacts are sufficient to place the services in the correct runtime context and to interpret observed behaviour coherently.	Supporting workflow/configuration figures and the retained context discussion in Section 5.4.

Table 3: Marker-visible closure signals used to interpret requirement satisfaction.

undocumented specialist knowledge. For **R1** and **R2**, success meant not only that the services existed, but that the report could show a bounded task, a bounded request/result family, and a visible engineering use for that result. For **R3** and **R4**, success meant that routine service change could be checked and delivered through repeatable mechanisms rather than only through ad hoc live runs. For **R5**, success meant that enough SIL and configuration context remained for the service evidence to stay intelligible and reproducible.

3.5 Observable closure signals

Table 3 states the concrete observations that would count as closure for each requirement. This does not replace the later evaluation; it makes the intended acceptance view easier to inspect by showing what a marker can actually look for in the report.

4 Software Development Process

The project did not begin as a green-field microservice build. It began as an inherited-system familiarisation exercise inside a difficult SIL environment and then evolved toward a service-centred solution as the recurring engineering pain points became clearer.

4.1 Phase 1: inherited-system familiarisation

The earliest work focused on bringing up the SIL reliably, understanding the containerised runtime, tracing configuration layering, and learning how missions and contact data moved through the stack. This phase produced artefacts that were valuable in their own right: workflow notes, mission experiments, configuration models, and lightweight DDS observation techniques. More importantly, it exposed a repeated pattern. The team's practical bottlenecks were not only "How do we run the SIL?" but increasingly "How do we interrogate selected state cleanly?" and "How do we create the traffic conditions we need without excessive manual setup?"

4.2 Phase 2: service opportunity identification

Once those bottlenecks were visible, the project narrowed into two interface opportunities. The first was a query service that could read selected middleware state and present it in a cleaner, safer form for developers and higher-level integrations. The second was an injection service that could introduce synthetic contacts into the autonomy-facing path for controlled testing. This reframing strengthened the project because it moved the centre of gravity from workflow enablement alone toward implemented software with clearer lifecycle coverage: analysis, design, implementation, testing, and evaluation.

4.3 Phase 3: testability and delivery hardening

A key lesson from working in the inherited environment was that "it runs on my SIL" was not strong enough as evidence. Access to a live LAVA/DDS runtime was constrained, startup cost was non-trivial, and any service that depended on a full environment for routine validation would be painful to maintain. I therefore treated testability as a first-class design concern. The service work was accompanied by automated tests, mock-mode execution paths, and a containerised CI/CD flow that moved the project closer to normal software engineering practice.

4.4 Narrowing from exploration to implementable scope

The project only became a stronger WPA4 candidate once I narrowed it deliberately. Early on, several broader directions were plausible: a larger observability toolset over the middleware surface, heavier scenario-authoring support around mission/configuration artefacts, or a more human-facing runtime interpretation layer. Those directions were not valueless, but they each risked either spreading the work too thinly or overlapping too heavily with the separate WPS strand.

The service-centred scope was therefore chosen as the highest-value implementable slice. It let me carry forward the most useful lessons from the earlier exploration

phase while still presenting a clearly bounded software product. That narrowing also improved the report's evidential position: a smaller number of service responsibilities, each backed by tests, mock mode, and CI/CD, was easier to justify credibly than a broader but less inspectable collection of loosely connected workflow improvements.

This was not only a scoping convenience. It was a technical judgement about where the inherited-system friction was most worth addressing first. Recurrent engineering work kept converging on two needs: selected state had to become easier to interrogate, and controlled nearby-contact conditions had to become easier to create deliberately. The final scope reflects that repeated convergence.

4.5 Collaboration and autonomy

The project was carried out inside a wider engineering team, but the work reported here is my direct contribution. The surrounding system architecture, upstream voice-command chain, autonomy middleware, and many baseline runtime components were inherited. My responsibility was to identify specific engineering problems within that context, propose and implement reusable service-level solutions, and then make those solutions more credible through testing, CI, and supporting artefacts.

What changed over time was not only the code I wrote, but the way I framed the engineering problem. Early work on the SIL workflow, mission variants, configuration layering, and DDS observation gave me enough command of the inherited system to make more deliberate software choices later. For example, the decision to expose BPP-style own-ship state and nearby-contact information through the DDS Query Service was informed by repeated earlier investigations into which signals developers most often needed when reasoning about missions. Similarly, the Contact Injection Service was not chosen in the abstract; it emerged because controlled traffic conditions were repeatedly useful during SIL work, but awkward to create quickly through heavier mission and configuration editing alone.

That history matters to the report's coherence. The microservices were not two detached side projects. They were the software-engineering response that became visible once the earlier workflow and observability work made the inherited system's recurring pain points concrete.

5 Design and Implementation

This section presents the main engineering contribution: the two microservices and the surrounding practices needed to make them useful and maintainable.

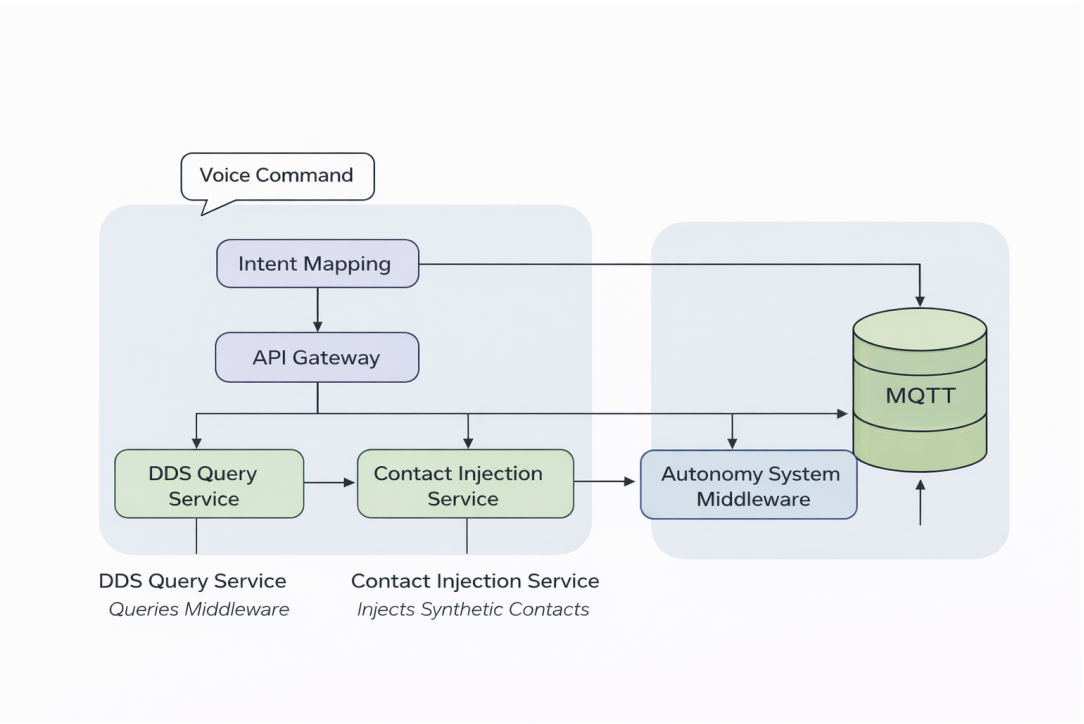

Figure 1: Contextual placement of the two project microservices. The voice-command, intent-mapping, API-gateway, MQTT, and autonomy-middleware elements are wider system context. The student's direct contributions in this report are the DDS Query Service, the Contact Injection Service, and the associated testing and delivery pipeline.

5.1 System context and service placement

The services sit within a wider autonomy stack whose upstream and downstream components were largely pre-existing. Figure 1 is suitable for inclusion because it gives a readable architectural placement view, but it only becomes safe and accurate when explained carefully. The figure is *not* a claim that I built the whole voice-command or broker-based integration chain. Instead, it shows where the two services sit relative to that wider system context.

The upstream context was a spoken-command and API-mediated integration chain. The downstream context remained the autonomy middleware and the wider LAVA runtime. The project's microservices therefore acted as *engineering interfaces at the middleware boundary*. One read selected state out of that boundary. The other wrote controlled synthetic contact data into it.

Table 4 makes those boundaries explicit.

5.1.1 Inherited runtime and data flow

Although my direct contribution begins at the microservice layer, understanding the inherited runtime was still essential to designing those boundaries well. At runtime,

Architectural element	Role	Relationship to this project
Voice command, intent mapping, API gateway	Upstream command-processing context	Included only to explain why a clean microservice boundary was useful; not implemented as part of this project.
MQTT/brokered integration	Wider integration context	Shows how service calls and runtime events fit into a larger distributed system, but is not claimed as the core project outcome.
DDS Query Service	Read-side middleware interface	Direct project contribution. Makes selected autonomy-relevant state queryable as structured JSON.
Contact Injection Service	Write-side middleware interface	Direct project contribution. Makes synthetic contact creation available through a bounded service interface.
Autonomy middleware and wider LAVA runtime	System-facing backbone	Inherited environment that the services had to read from, write to, and coexist with safely.

Table 4: Ownership and scope boundaries for the architectural elements shown in Figure 1.

the autonomy stack executes as a set of long-running containerised services connected over a shared publish/subscribe backbone. In practical terms, those services cover responsibilities such as world-state formation, planning, control, chart/environment handling, and communications. The important engineering point for this project was that useful state already existed on that middleware surface; the problem was that it was not yet exposed in a way that was convenient for day-to-day developer use.

The simulation-autonomy-control loop also explains why the two services were useful complements. The simulator and bridge processes publish own-ship and contact-related state into the middleware layer; upstream autonomy services consume that state to form a world model and plan behaviour; and control outputs are then fed back into the simulator. A read-side query service therefore gives developers a cleaner way to inspect what the stack currently appears to believe. A write-side injection service gives them a cleaner way to influence what the stack will believe next, by introducing controlled synthetic contacts at the same architectural boundary.

This view also clarifies why the services are valuable without overclaiming them. They do not replace the autonomy stack, the simulation, or the middleware itself. Instead, they provide two carefully chosen engineering surfaces around the inherited backbone: one for interrogating selected state, and one for creating selected stimuli.

5.1.2 Why services rather than direct tooling or one-off scripts

There were plausible alternatives to the final design. One option would have been to rely on direct DDS tooling and personal scripts for both state inspection and synthetic-contact generation. That would have been fast for a single specialist devel-

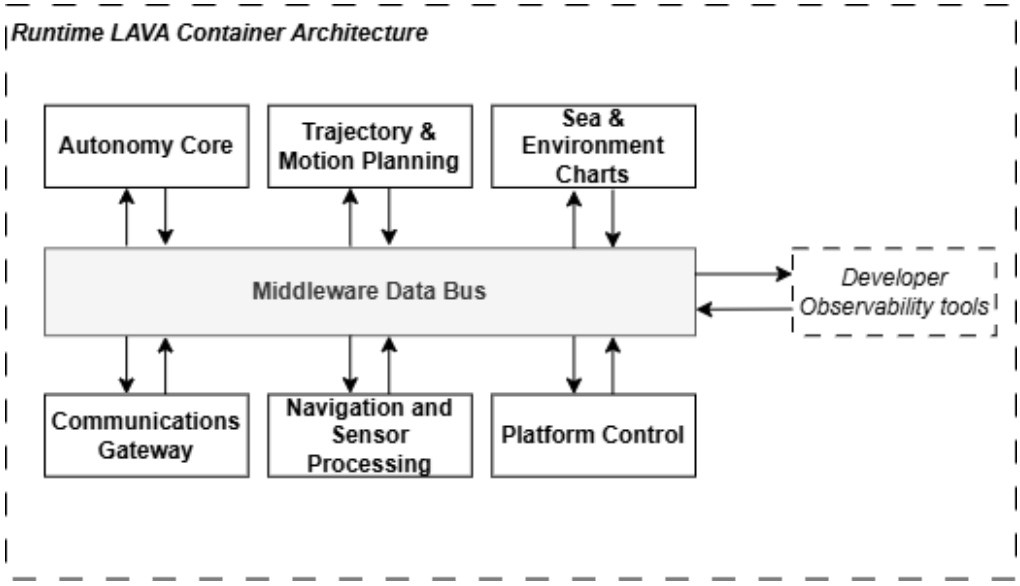

Figure 2: Abstract inherited runtime view retained from the earlier SIL-centred phase. It shows why the middleware bus was the natural boundary for both selected-state querying and controlled synthetic-contact injection.

oper, but it would have preserved the main inherited-system problem: useful capabilities remaining tied to middleware expertise and informal local knowledge. Another option would have been to push all controlled-scenario work into mission and simulation files alone. That remained useful for broader scenario structure, but it was comparatively heavy-weight when the engineering task was to inspect a selected class of runtime state or create a specific nearby-contact condition quickly.

The chosen microservice approach therefore represented a deliberate compromise. It sacrificed some theoretical flexibility in exchange for bounded, reusable interfaces that other engineers could understand, test, and deliver more easily. Compared with direct DDS scripts or mission-file-only control, the services created a narrower and more inspectable intervention point at the middleware boundary. Separate read-side and write-side services also made the contracts easier to reason about than a single catch-all "middleware utility" would have been.

5.1.3 Key design decisions at the architecture boundary

Several design decisions follow directly from the placement shown above.

First, I kept the read-side and write-side responsibilities separate. Querying selected state and injecting synthetic contacts are related engineering needs, but they are not the same concern. Separate services kept each contract narrower, made failure modes easier to reason about, and avoided producing one over-broad "middleware utility" whose scope would be harder to test and harder to justify.

Second, I exposed *selected* state rather than trying to surface the whole DDS universe.

This bounded approach was better aligned with confidentiality, easier to mock in tests, and more likely to remain stable for downstream consumers.

Third, I treated deployment and testing as part of the design. A service boundary that only works against a live, fully brought-up autonomy runtime is much weaker in practice than one that can be built, started, and exercised in isolation. That is why mockability and container execution appear repeatedly throughout the implementation.

5.2 DDS Query Service

5.2.1 Purpose and design rationale

The DDS Query Service addressed a simple but important problem: selected runtime state was useful to many engineers, but the direct DDS surface was too specialised to be a convenient day-to-day interface. The service therefore provided a bounded read-side abstraction over the middleware layer. Instead of asking each consumer to understand topic families, message types, and runtime dependencies directly, the service exposed selected information through structured JSON responses.

This was a design choice about *abstraction*, not about replacing DDS. DDS remained the underlying backbone. The value of the service was that it turned selected DDS-backed state into an engineering interface that could be queried more easily, tested more easily, and integrated more safely.

5.2.2 Scope of exposed state

The service scope was deliberately selective. At minimum, the service surfaced best-present-position (BPP) style state and nearby-contact information. Those data families were valuable because they answered two recurring developer questions: "Where does the autonomy stack believe own-ship is?" and "What contacts does it currently believe are nearby?" They were prioritised because those were the most recurrent engineering questions during SIL-backed debugging and integration work.

Because the full service schema cannot be disclosed safely, Table 5 shows the outward contract only at the bounded family level needed to inspect the service's engineering use.

No broader query surface is claimed here beyond BPP-style own-ship state and nearby-contact information. Keeping the exposed families narrow is part of how the report keeps the service scope aligned with what is later closed explicitly in evaluation.

Engineering question / task	Bounded request family	Bounded response family	Why this was sufficient
Inspect current own-ship state during a SIL check	Request current own-ship / BPP-style state.	Structured JSON-style own-ship view.	Answers a routine diagnostic question and gives downstream code a cleaner hand-off than raw middleware traffic.
Inspect nearby contacts before or after a controlled action	Request current nearby-contact state family.	Bounded nearby-contact view suitable for comparison or logging.	Makes the surrounding traffic picture easier to interrogate without disclosing raw topic details.
Reuse the same query path in tests or higher-level integrations	Request the same bounded state families through the service boundary or mock mode.	Stable contract response or bounded unavailable/error result.	Keeps consumers tied to a reusable service contract rather than to direct DDS knowledge.

Table 5: Sanitised outward contract examples for the DDS Query Service.

5.2.3 Implementation shape

The service separated two concerns. One part interacted with the middleware-facing side of the system and acquired the selected runtime data. The other part shaped that data into a stable JSON contract suitable for an HTTP-style service boundary. That separation mattered for maintainability: it kept protocol-specific or runtime-specific concerns away from the outward-facing API contract.

It also mattered for testing. Because the middleware-facing part could not be assumed to be available in every development or CI environment, the implementation supported a *mock mode*. In mock mode, the query layer returned deterministic fixture-like data rather than depending on a live DDS/LAVA runtime. This made it possible to test request handling, response shaping, error cases, and contract behaviour without paying the cost of a full SIL bring-up. The same outward contract could then be exercised again in a SIL-backed check when a developer needed a bounded view of current own-ship or nearby-contact state without falling back to direct DDS inspection as the default path.

An earlier lightweight DDS snapshot-and-diff helper, produced during the inherited-system investigation phase, informed which signal families were most worth surfacing. In that sense, the service subsumed the strongest part of the earlier observability strand into a cleaner software artefact.

DDS Query Service lifecycle (sanitised view)
External request received → query family validated → live adapter or mock adapter selected → selected state acquired and normalised → structured JSON result or bounded unavailable/error response returned.

Figure 3: Sanitised lifecycle view for the DDS Query Service. The value of the service lies in keeping validation, adapter choice, and response shaping inside a reusable boundary rather than exposing those concerns directly to each consumer.

5.2.4 Request lifecycle and bounded failure handling

A typical request lifecycle was conceptually straightforward: receive an external request, validate the requested query family, acquire the relevant middleware-backed state through the service's internal adapter layer, normalise the result, and return a structured JSON response. The key design point was that validation and response shaping happened *before* any consumer had to care about middleware specifics. That pushed complexity inward and kept the outward contract more stable.

Bounded failure handling was also important. In an inherited distributed system, many failures are environmental rather than purely logical: missing middleware connectivity, stale services, or incomplete runtime bring-up. A useful query service therefore has to fail in a way that helps engineering diagnosis rather than simply propagating raw internal behaviour outward. The mock-mode path supported this by separating basic service correctness from environment-specific failure modes.

5.2.5 Key implementation challenges and resolutions

The report is stronger if it also states the main technical problems encountered while making the service usable. Table 6 summarises the most important ones.

5.2.6 Why the abstraction was engineering-relevant

The practical value of the DDS Query Service was not only that it returned JSON. Its deeper value was that it changed the *unit of understanding*. Instead of every downstream engineer or tool reasoning directly about DDS access, they could reason about a smaller set of service-level questions: "Can I request current own-ship state? Can I request current nearby contacts? What does the service return when that information is unavailable?" That is a more reusable engineering surface, and it is exactly the kind of outcome that fits a practice-based software project.

Table 5 makes that point more inspectable. The service's outward use can be described in terms of bounded engineering questions, bounded request families, and bounded JSON-style results rather than raw middleware traffic. That is enough to support the claim that selected state became easier to interrogate and easier to hand to other tools without overstating total runtime coverage.

Challenge	Why it mattered	Resolution adopted and remaining compromise
Keeping the outward API cleaner than the DDS-facing internals	A raw middleware-shaped interface would have pushed protocol knowledge and state-shaping complexity onto every consumer.	I separated middleware acquisition from outward response shaping, so the service could present a narrower JSON contract. The compromise is that only the most useful state families were exposed, not the whole underlying surface.
Supporting routine development without depending on a live runtime	A service that only worked against a fully brought-up SIL would have been slow to iterate on and weak in CI.	I added mock-mode execution so request handling, response shaping, and bounded error cases could be exercised against deterministic fixtures. Live SIL checks still remained necessary for final adapter verification.
Handling environmental failures without making diagnosis harder	Missing connectivity, stale services, or incomplete bring-up can look similar to logic faults if surfaced poorly.	I treated validation and bounded failure handling as part of the service contract, so consumers received more useful unavailable/error behaviour than raw middleware leakage. The compromise is that some live-runtime nuance remains hidden behind the abstraction.

Table 6: Most important DDS Query Service implementation challenges and how they were addressed.

5.3 Contact Injection Service

5.3.1 Purpose and design rationale

The Contact Injection Service addressed the write-side equivalent of the query problem. Autonomy behaviour often depends on nearby-contact conditions, but creating those conditions repeatably can be awkward when the only route is manual simulation editing or direct specialist interaction with middleware internals. The service therefore provided a cleaner engineering path for introducing synthetic contacts into the autonomy-facing side of the stack.

The design objective was not to fake the whole environment. It was to create a reusable, bounded interface for *controlled traffic stimulation*. That made the service useful for scenario construction, testing, and demonstration work where the team needed the autonomy stack to "believe" that a nearby contact existed under known conditions.

5.3.2 Injection model

At a high level, the service accepted a synthetic-contact request, translated it into the middleware-facing form expected by the autonomy stack, and published it onto

Setup task	Bounded injection request family	Bounded observed effect	Why this was useful
Create a nearby-contact condition for a SIL check	Submit a synthetic-contact request using relative position, heading, speed, and related bounded contact attributes.	Autonomy-facing side receives a deliberate nearby-contact condition in the surrounding traffic picture.	Creates controlled traffic conditions without waiting for accidental contacts or rebuilding a heavier scenario pack.
Vary the same condition across repeated runs	Adjust the same bounded motion-related fields across reruns.	Traffic picture can be recreated or deliberately varied through the same service path.	Supports controlled experimentation rather than one-off handcrafted scenario edits.
Reuse the same path in testing or integration support	Submit requests through a stable service boundary rather than manual mission-file-only edits.	The same injection path can support repeated setup while live runtime checks confirm the final effect.	Turns scenario control into a software-facing capability that can be tested and delivered more credibly.

Table 7: Sanitised outward contract examples for the Contact Injection Service.

Contact Injection Service lifecycle (sanitised view)
Injection request received → bounded motion/contact inputs validated → synthetic contact translated into the middleware-facing form → publish step executed through the service boundary → nearby-contact condition becomes available for live confirmation in the autonomy-facing traffic picture.

Figure 4: Sanitised lifecycle view for the Contact Injection Service. The important design feature is that deliberate nearby-contact setup becomes a reusable service path rather than a bespoke one-off editing exercise.

the relevant path so that the wider system could consume it as though it were an observed contact. This design aligned well with the inherited architecture because it preserved the autonomy middleware as the central system-facing backbone while giving developers a more accessible write-side tool.

The main controlled variables were the kinds of contact properties that matter for scenario setup: relative position, heading, speed, and related contact attributes. These properties allowed developers to create repeatable encounter conditions without relying solely on external traffic or on repeatedly editing larger mission artefacts by hand. They were prioritised because motion-related fields were the minimum useful set for creating deliberate nearby-contact conditions without exposing or supporting the full middleware contact surface.

Because the full request schema cannot be disclosed safely, Table 7 shows the outward service contract only at the bounded family level needed to inspect the engineering value of the injection path.

5.3.3 Relationship to existing workflow and mission artefacts

The Contact Injection Service did not make the earlier SIL and mission work irrelevant. Instead, it changed its role. The workflow and mission artefacts became *supporting context* for when and how to exercise the service, while the service itself became the more substantial engineering outcome. The older artefacts remained important for bringing up the right environment, understanding which vessel or overlay was active, and interpreting what the autonomy stack did in response to the injected contact.

5.3.4 Service-level trade-offs

The main trade-off in the Contact Injection Service was between flexibility and boundedness. A very flexible injection tool could expose a large fraction of the underlying contact model, but that would make the outward contract harder to validate and would increase the risk of drifting into an unsafe or disclosure-heavy description. I therefore frame the service as a bounded engineering interface for common synthetic-contact use rather than as a universal scenario authoring environment.

A second trade-off concerned how the service complemented simulation-side mission files. Mission files remained useful for broad scenario structure, but they were comparatively heavy-weight when the engineer's immediate need was "place a nearby contact under known conditions and observe the response". The injection service filled that gap by supporting quicker controlled setup at the middleware boundary.

5.3.5 Key implementation challenges and resolutions

The Contact Injection Service also involved several non-trivial engineering choices. Table 8 summarises the most important ones.

5.3.6 Expected validation path

For this service, the validation path naturally had two layers. The first layer was service-level: request validation, command construction, and mock-mode behaviour. The second layer was live integration: confirming that, once published, the synthetic contact became visible to the autonomy-facing side of the inherited stack in the way the engineer expected.

Table 7 makes that outward contract inspectable at the same bounded level used later in evaluation. It shows the important engineering point without over-disclosing the underlying contact model: a developer could submit a small family of motion-related parameters and use the same path to recreate or vary a deliberate nearby-contact condition.

Challenge	Why it mattered	Resolution adopted and remaining compromise
Choosing a minimal but useful input family	A very broad synthetic-contact model would have been harder to validate, explain, and keep stable for common engineering use.	I centred the service on relative position, heading, speed, and related bounded contact attributes. The compromise is that the service supports common nearby-contact setup rather than universal scenario authoring.
Complementing rather than replacing mission/configuration artefacts	The service had to accelerate recurring setup tasks without pretending that wider scenario structure no longer mattered.	I treated the service as a middleware-boundary tool for deliberate contact creation and kept mission/configuration artefacts as the integration scaffold. The compromise is that some scenario work still belongs outside the service.
Verifying live effect separately from request acceptance	A request can be well-formed without guaranteeing that the surrounding runtime is brought up or interpreted correctly.	I adopted a two-layer validation model: service-level validation for request construction and mock behaviour, followed by live SIL confirmation that the nearby-contact condition became visible on the autonomy-facing side. The compromise is that full proof still depends on bounded live checks.

Table 8: Most important Contact Injection Service implementation challenges and how they were addressed.

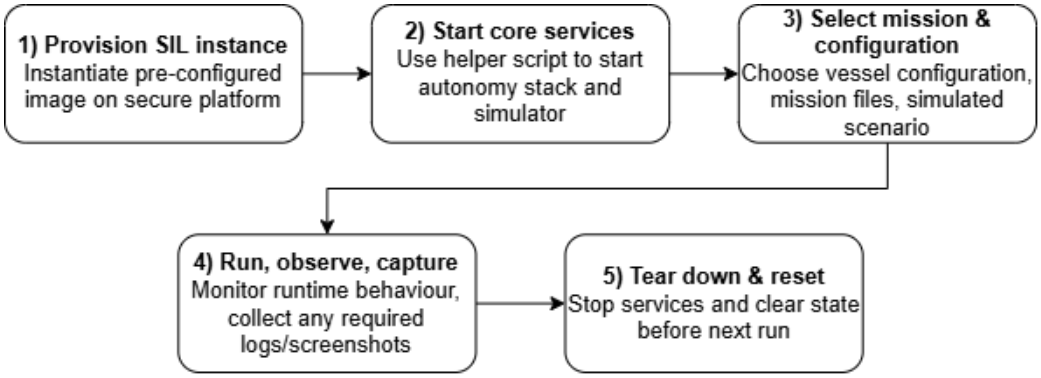

Figure 5: Supporting SIL workflow retained from the earlier project phase. In this microservice-centred report it explains how the services are positioned and exercised rather than serving as the main contribution on its own.

5.4 Supporting workflow, mission-pack, configuration, and runtime interpretation

The report retains a supporting strand from the earlier SIL-centred phase because the microservices did not emerge in isolation. They were shaped by repeated work on workflow repeatability, controlled mission execution, and configuration interpretation, and they still depend on those artefacts when moved into a live inherited environment.

First, a repeatable SIL workflow remained necessary. The services existed inside a containerised autonomy environment with non-trivial bring-up, mission-selection, restart, and teardown behaviour. Figure 5 therefore remains useful as a concise view of the developer-facing lifecycle around a run.

In practice, that earlier workflow work reduced the chance of misinterpreting environment failures as service failures. It clarified the order in which autonomy services were normally started, highlighted common stale-state and restart issues, and gave the team a more repeatable route from "cold SIL instance" to "ready to exercise a service-backed scenario". That remains relevant because even strong service interfaces can appear misleading if they are evaluated in the wrong runtime state.

Second, the earlier mission-pack work still mattered as an integration scaffold. The retained baseline waypoint and station-keeping scenario families were originally designed to create interpretable, repeatable SIL behaviour. In the microservice-centred version of the project, that mission work serves a different but still important role: it provides the kinds of controlled situations in which the services make sense. The DDS Query Service is most useful when developers need to inspect own-ship or nearby-contact state during a recognisable mission. The Contact Injection Service is most useful when those scenario families need to be augmented with deliberate nearby-contact conditions rather than relying only on pre-authored simulation traffic.

Third, configuration layering still mattered. A wrong vessel profile, stale overlay, or misunderstood shared configuration could change the behaviour observed during service integration or SIL validation. Figure 6 summarises the inherited layered con-

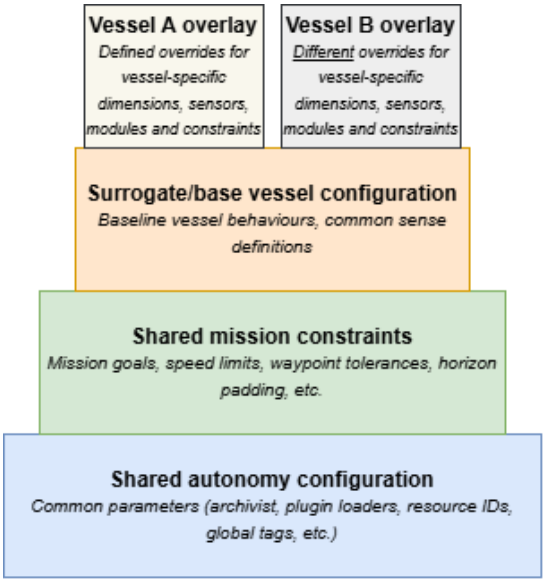

Figure 6: Abstract configuration layering view retained from the earlier project phase. Understanding which layers were shared and which were vessel-specific remained important when validating the services in the inherited stack.

figuration model that remained relevant when positioning the services in the correct runtime context.

The key engineering lesson from that earlier work was that many apparently service-level anomalies can still be explained by environment choice, active overlays, or mission-side setup rather than by defects in the service boundary itself. Preserving the configuration artefacts in the report therefore does more than add context: it explains how the microservice work was kept interpretable and how false diagnoses were reduced.

Finally, earlier lightweight DDS observation work still had value as a diagnostic aid. It is best understood here as a precursor rather than the end point. It helped identify which classes of state were most worth surfacing through the DDS Query Service, and it reinforced the general design principle that developers benefited most from bounded, low-overhead visibility into runtime state rather than from full raw-middleware exposure by default.

5.5 Test strategy and mock-mode execution

5.5.1 Why testability mattered

Testability was a core design concern because depending on a full live LAVA/DDS environment for routine service validation would have made both development and assessment weaker. The environment was costly to bring up, not always available, and unsuitable as the only place to exercise basic contract or validation behaviour. Automated tests and mock-mode execution therefore mattered for three reasons:

Test category	DDS Query Service	Contact Injection Service
API / contract tests	Validate request handling and JSON response shape without live DDS dependencies. The latest documented Bamboo run reported 113 passed tests across the automated DDS Query Service suite.	Validate request handling and synthetic-contact command structure without live middleware dependencies. The latest documented Bamboo run reported 137 passed tests across the automated Contact Injection Service suite.
Validation and error-path tests	Check missing-input, malformed-input, and bounded error cases.	Check invalid or incomplete synthetic-contact requests and bounded failure behaviour.
Mock-mode service tests	Exercise service behaviour against deterministic fixture data instead of a live runtime.	Exercise injection behaviour against mock middleware adapters instead of a live runtime.
Container smoke tests	Confirm the service can build and start correctly inside its container image.	Confirm the service can build and start correctly inside its container image.
Coverage evidence	Bamboo logs reported 63% total line coverage for the DDS Query Service build.	Bamboo logs reported 65% total line coverage for the Contact Injection Service build.

Table 9: Bounded test-strategy view for the two microservices.

they improved local developer feedback, they enabled CI, and they made the report's engineering claims stronger.

5.5.2 Test structure

Both services were accompanied by automated tests and those tests could run without a live environment through mock-mode behaviour. A sensible decomposition, consistent with the stated implementation goals, is shown in Table 9.

5.5.3 Local developer loop versus live SIL loop

The resulting validation model had a practical day-to-day shape. Most iterations could happen in a fast local loop: edit service code, run automated tests in mock mode, rebuild the service container if necessary, and only then move to the heavier SIL-backed check when middleware integration actually needed to be confirmed. That split is important from a software engineering perspective because it reduced friction without pretending that live integration no longer mattered.

This was also a deliberate architectural choice rather than a pure convenience feature. By keeping a stable outward contract and allowing the service internals to select between live adapters and mock adapters, the implementation separated routine service

correctness from runtime-availability concerns. That in turn made the services easier to evolve: developers could change parsing, validation, response shaping, or request construction logic without immediately paying the full integration cost.

That split also made the services more maintainable for engineers other than their author. A developer who wanted to change response shaping, validation behaviour, or request handling did not first need access to a fully operational SIL or a specialist DDS setup. They could work against the mock-backed contract, observe failures quickly, and only pay the cost of live integration when the change genuinely crossed the middleware boundary. In an inherited system with costly bring-up, that is a substantial engineering improvement in itself.

5.5.4 Live versus mock validation

Mock mode did not eliminate the need for live validation. It changed the division of responsibility. Mock-mode tests were appropriate for service contracts, parsing, validation, and much of the service logic. Live SIL checks were still needed for the final read-side and write-side integration with the inherited middleware. This distinction is important because it keeps the claims honest: automated tests support service-level correctness and repeatability, but they do not by themselves prove that every live DDS interaction behaves correctly in all operational circumstances.

Taken together, the numeric Bamboo evidence and the bounded SIL-backed vignettes provide a balanced validation story: most routine correctness is exercised in repeatable test infrastructure, while live checks are used to confirm the final middleware boundary.

5.6 CI/CD pipeline and delivery path

The project also needed a credible delivery story. The service work was integrated into a Bamboo-based CI environment that built and tested containerised services and pushed images to Harbor from the main branch. This was important because it turned the project from a pair of local services into a more maintainable engineering artefact.

This pipeline mattered for more than convenience. It supported repeatability, reduced the chance that service behaviour would be validated only in a single engineer's environment, and strengthened the project's claim to be a practice-based software contribution rather than a one-off prototype.

It also sharpened the project's engineering narrative. A microservice accompanied by automated tests, containerised execution, and controlled publication reads much more clearly as a software product outcome than one accompanied only by ad hoc local runs. In the context of WPA4, that is important because it shows not just that code was written, but that it was developed with an identifiable lifecycle and delivery path.

Pipeline stage	Purpose	Current evidence position
Branch-triggered build	Create a repeatable build path for service changes rather than relying on ad hoc local execution.	On feature/add-harbor-credentials, code changes triggered full service builds (for example DDS Query Service #8 and Contact Injection Service #27), while specs updates appeared separately as testless builds.
Containerised test execution	Run automated tests in a controlled Docker-based environment so local and CI behaviour remain aligned.	Bamboo build logs and summaries showed automated test execution, coverage generation, and later artifact handling inside the containerised path.
Artifact and report capture	Preserve machine-readable outputs such as test reports or coverage summaries for later inspection.	Bamboo exposed Summary, Tests, Logs, Metadata, and Webhooks views, and the logs recorded both per-build coverage summaries and HTML coverage generation to htmlcov.
Main-branch publication to Harbor	Publish deployable images only from the appropriate branch, reducing accidental release paths.	Feature-branch logs explicitly showed Harbor tagging, login, and push being skipped when not on main branch, implying that publication is reserved for the main branch path.
Versioned build logic	Keep build behaviour close to the codebase where feasible so service evolution and delivery evolution remain coupled.	Repeated "Specs configuration updated" entries alongside code-change builds indicate repository-driven Bamboo Specs changes were versioned separately from service code changes.

Table 10: Bounded view of the service CI/CD path.

The staged flow is straightforward but important. A service change first enters a branch-triggered build path, where Bamboo constructs the container image and runs the automated suite inside a controlled Docker-based environment. That arrangement helps keep CI behaviour aligned with the packaged runtime rather than with a particular engineer's workstation. The resulting reports and coverage summaries are then exposed through Bamboo's build views, giving the team a bounded but inspectable evidence trail. Publication is intentionally stricter: the feature-branch evidence shows Harbor publication being skipped outside main, which reinforces the principle that testable development builds and publishable release artefacts are not the same thing.

6 Evaluation

6.1 Evaluation approach and claim boundaries

The evaluation is intentionally structured around requirement closure because that is the clearest way to assess a practice-based project under confidentiality constraints. The aim is not to overstate what can be proven from the available evidence. Instead, the aim is to show what the implemented services and supporting artefacts demonstrably achieved, what those achievements support, and where the evidence remains bounded.

The evidence sources used here are: the implemented service roles and architecture; the sanitised contract examples for each service; Bamboo build summaries and logs reporting test totals, coverage, durations, and branch behaviour; the stated presence of automated tests, mock mode, and CI/CD integration; and the supporting SIL and configuration artefacts retained from the earlier project phase. For **R1** and **R2** in particular, the evaluation below uses small validation-case structures so that setup, action, observed result, supported claim, and remaining uncertainty are explicit rather than left implied.

6.2 Adequacy of the quantitative evidence

The quantitative evidence in Table 11 is useful, but it has to be interpreted carefully. The test totals and coverage values are strong enough to show that the services were more than ad hoc scripts: they were exercised repeatedly through a CI-backed path and their main request, validation, bounded failure, and mock-adapter behaviours were checked routinely. The short build durations also matter because they indicate that the services could move through a practical branch-level feedback loop rather than through occasional heavyweight validation only.

At the same time, these numbers do not justify overclaiming. Coverage in the low-to-mid 60s is meaningful service-level evidence, but it is not exhaustive assurance,

Low-risk evidence item	Value used in this report
Automated tests passed across both service builds	250 total (113 DDS Query Service + 137 Contact Injection Service)
DDS Query Service coverage	63% total coverage on the documented Bamboo build
Contact Injection Service coverage	65% total coverage on the documented Bamboo build
Latest successful feature-branch build durations	26 seconds (DDS Query Service) and 30 seconds (Contact Injection Service)
Branch-delivery evidence	Feature-branch builds ran full service build/test paths, while Harbor publication was skipped when not on main

Table 11: Low-risk quantitative evidence summary derived from the sanitised Bamboo information already discussed in the report.

especially for behaviour that still depends on live middleware connectivity and runtime state. I therefore treat the Bamboo values as evidence that routine correctness and delivery discipline were established, not as evidence that every live DDS interaction, every scenario condition, or every environment-dependent edge case was fully proven. That is why the quantitative evidence is used alongside the bounded R1/R2 validation cases rather than instead of them.

6.3 Requirement-closure table

In Table 12, *Met* denotes claims directly supported by the bounded evidence shown in the report, while *Bounded* denotes materially supported engineering progress that remains intentionally narrower than exhaustive operational proof.

Table 12 summarises the closure position at a glance; the discussion below explains how each requirement is supported and where the evidence remains deliberately bounded.

6.4 Evaluation discussion

The evaluation has an identifiable software centre of gravity. The project is argued through implemented read-side and write-side services, supported by testing and CI/CD discipline, with the older workflow and configuration work retained only where they explain or validate those services.

6.4.1 Overall evidential picture

Taken together, the evidence has a coherent shape. **R1** and **R2** are supported primarily by implemented service responsibilities, bounded interface scope, and quali-

Req.	Strongest evidence in report	Supported claim	What remains unproven	Status
R1	Service-capability evidence: Table 5; Section 5.2; R1 validation case below.	Selected own-ship and nearby-contact state became queryable through a reusable boundary for debugging, comparison, and downstream hand-off.	Full topic coverage, live performance under all runtime conditions, or exhaustive middleware fidelity.	Met
R2	Service-capability evidence: Table 7; Section 5.3; R2 validation case below.	Synthetic nearby-contact conditions could be created more deliberately through a reusable service path.	Exhaustive encounter coverage, parity with all real sensors, or universal scenario authoring.	Met
R3	Engineering-discipline evidence: Table 9; Bamboo totals and coverage in Table 11.	Core request, validation, and bounded failure behaviour can be rechecked repeatedly without a live runtime.	Complete adequacy of test breadth or correctness of every live DDS interaction.	Bounded
R4	Engineering-discipline evidence: Table 10; Bamboo timings and branch-conditioned publication behaviour in Table 11.	Service changes moved through a repeatable build, test, and packaging path with bounded publication controls.	Long-term reliability, operational deployment scale, or adoption breadth.	Bounded
R5	Context/enabling evidence: Figures 5–6 and Section 5.4.	The services were positioned in enough workflow and configuration context to be run and interpreted coherently.	Elimination of inherited-system complexity or complete onboarding coverage.	Met

Table 12: Requirement-closure view with stronger evidence traceability and explicit claim boundaries.

tative integration reasoning. **R3** and **R4** are supported more numerically through Bamboo-derived test counts, coverage, build times, and branch-conditioned publication behaviour. **R5** is supported by continuity with the earlier SIL workflow and configuration work that remained necessary to exercise the services safely. That mix is appropriate to a practice-based report in a confidentiality-constrained environment: some requirements are best evidenced by stable software boundaries, while others are best evidenced by repeatable software-engineering process.

6.4.2 R1 - DDS Query Service

R1 is best read as a bounded validation case for a developer-facing service boundary rather than as a claim of end-user visualisation or total runtime observability.

Setup / precondition. A developer is working in a SIL-backed check, or in an equivalent service-backed diagnostic context, and needs selected own-ship and nearby-contact state without falling back to routine direct DDS inspection.

Action taken. The developer requests BPP-style own-ship state and nearby-contact information through the DDS Query Service boundary described in Table 5.

Observed bounded result. The service returns a structured JSON-style own-ship view together with a bounded nearby-contact view suitable for comparison, logging, or downstream hand-off.

R1 exemplar	
Example engineering task	Check current own-ship state and nearby contacts during a SIL-backed debugging or comparison run.
Bounded request family	Request BPP-style own-ship state together with current nearby-contact state through the service boundary.
Bounded returned field family	Normalised own-ship position/orientation-style values plus a bounded nearby-contact summary suitable for comparison or downstream hand-off.
Practical use in context	Provides one reusable interrogation path for own-ship state and nearby contacts without routine direct DDS inspection.

Claim supported. Selected autonomy-relevant runtime state became inspectable through a reusable service contract, reducing routine dependence on DDS-specific knowledge for common diagnostic tasks.

Compared with the earlier route, the same task no longer had to begin at the raw middleware surface or in a specialist one-off script. That does not mean DDS became irrelevant, but it does mean that a common class of engineering question could now begin from a narrower and more reusable interface. For WPA4 purposes, that is the key improvement: the project produced a software artefact that changed the default path for recurring interrogation work.

Remaining uncertainty. This does not prove complete coverage of every relevant topic family, nor live behaviour under all runtime states or performance conditions.

6.4.3 R2 - Contact Injection Service

R2 is also best read as a bounded validation case focused on deliberate scenario control rather than on universal scenario authoring.

Setup / precondition. A developer needs a nearby-contact condition for a SIL check, debugging session, or controlled experiment, but does not want to rely on accidental traffic or to rebuild a heavier scenario pack for every run.

Action taken. The developer submits a bounded synthetic-contact request through the Contact Injection Service using motion-related parameters such as relative position, heading, and speed, consistent with Table 7.

Observed bounded result. The autonomy-facing side of the inherited stack receives a deliberate nearby-contact condition in the surrounding traffic picture, and the same path can be used to recreate or vary that setup.

R2 exemplar	
Example setup goal	Create a deliberate nearby-contact condition for a repeatable SIL check or debugging session.
Bounded synthetic-contact input family	Submit relative position, heading, speed, and related bounded motion/contact attributes through the Contact Injection Service.
Observed nearby-contact effect	The autonomy-facing traffic picture shows a deliberate nearby contact that can be recreated or varied through the same path.
Practical use in context	Supports service-level scenario stimulation against a known contact setup without rebuilding a heavier mission pack for each run.

Claim supported. Scenario control became more direct and reusable through a service boundary rather than remaining dependent on heavier manual setup alone.

Compared with the earlier route, a developer could now create or vary a nearby-contact condition through one bounded request family rather than treating every such check as a bespoke mission-editing exercise. That is the practical engineering gain evidenced here: not universal scenario authoring, but a faster and more repeatable path for a recurring class of test and debugging setup.

Remaining uncertainty. This does not prove exhaustive encounter coverage, full parity with all real sensors, or that the service replaces broader mission-authoring workflows.

Task	Earlier route	Service-based route	Bounded improvement
Selected runtime interrogation	Direct DDS inspection or one-off specialist-facing tooling for selected own-ship or nearby-contact state.	Reusable DDS Query Service request over bounded state families.	Cleaner, more reusable interrogation path for comparison, logging, or downstream hand-off.
Controlled nearby-contact setup	Heavier mission/configuration edits or dependence on accidental traffic.	Reusable Contact Injection Service request with bounded motion-related inputs.	Quicker, more repeatable scenario setup for debugging and SIL checks without bespoke scenario authoring.

Taken together, the R1 and R2 material gives a more triangulated evaluation than either requirement-closure labels or Bamboo numbers would provide on their own. The contract tables show what the services were designed to do, the bounded exemplars show how those capabilities were exercised in practice, and the earlier-route/service-route comparison clarifies why the change mattered to engineering work rather than merely to architectural neatness.

6.4.4 R3 - Automated tests and mock mode

R3 is where the project becomes more obviously software-engineering-led. Automated tests and mock mode show that the services were designed to be exercised repeatedly, not only during occasional live SIL sessions.

Bamboo evidence grounds this requirement numerically: the DDS Query Service build reported 113 passed tests with 63% total coverage, and the Contact Injection Service build reported 137 passed tests with 65% total coverage. Read alongside the test design, those figures show good coverage of request handling, validation rules, bounded failure behaviour, mock-backed service logic, and basic container-start confidence.

Those coverage figures should be read as *proportionate service-level evidence*, not as a claim of exhaustive assurance. In this report they support the view that the main request paths, validation logic, bounded error handling, mock adapters, and container start-up behaviour were exercised regularly in CI. They do not by themselves prove

every live DDS interaction or environment-dependent edge case, which remain more dependent on bounded live SIL checks.

6.4.5 R4 - CI/CD path

R4 strengthens the project's credibility by showing that the services participate in a repeatable delivery path. Bamboo, Docker, and Harbor move the work beyond local-only experimentation and toward maintainable team artefacts. This matters in a practice-based report because it shows consideration of build, test, packaging, and publication rather than implementation alone. It is also product evidence: the interfaces can be rebuilt, rechecked, and packaged through the same controlled path by the wider team, which reduces integration risk when the services evolve. In other words, the CI/CD path is not a peripheral appendix to the software contribution; it is part of what makes the service boundary work reusable by others.

The pipeline evidence is now inspectable in bounded textual form. On feature/add-harbor-credentials, the latest successful DDS Query Service and Contact Injection Service builds completed in 26 seconds and 30 seconds respectively; branch statistics reported 87% and 84% success, and the logs explicitly showed Harbor publication steps being skipped when the branch was not main.

6.4.6 R5 - Supporting artefacts

R5 remains important because the services do not exist in a vacuum. The earlier SIL and configuration artefacts explain how to bring up the right environment, how to avoid stale-state mistakes, and how to interpret what the services are doing once live integration begins. In this service-centred account these artefacts are no longer the main event, but they are still necessary supporting evidence.

This is also where continuity between the earlier and later project phases matters most. The workflow, mission, and configuration work reduced integration ambiguity before the microservices existed, and they continued to do so once the services were introduced. That does not make them the report's centre of gravity, but it does justify retaining them as part of the evaluated solution rather than as discarded background.

Across R1–R5, the overall pattern is consistent. The strongest evidence supports the claim that the project produced bounded, reusable engineering interfaces around an inherited autonomy stack and then strengthened them through testability, CI discipline, and supporting artefacts. The evaluation is therefore strongest on interrogability, controlled scenario setup, and maintainability. It is intentionally weaker on exhaustive live-runtime proof or long-term organisational adoption, and those limits are stated explicitly rather than hidden. This is also why the report reads differently from a human-facing dashboard evaluation: the centre of gravity here is service capability, repeatable validation, and delivery discipline.

6.5 Engineering benefit supported by the evidence

Within those stated limits, the evidence still supports a meaningful engineering benefit. The DDS Query Service reduced the need for recurring interrogation work to begin directly at the DDS layer. The Contact Injection Service reduced the amount of repeated manual setup needed to create one deliberate nearby-contact condition. Mock mode reduced the cost of routine iteration, while the CI/CD path reduced the risk that the services would remain local-only utilities. None of those changes solve the whole inherited system, but together they do amount to a credible shift in how a bounded set of common engineering tasks can be performed.

That is also why the project outcome is better understood as *engineering leverage* than as raw feature count. The strongest improvement is not that the report lists many capabilities, but that it shows a smaller set of capabilities being made cleaner, more repeatable, and more deliverable than before. For a practice-based software engineering project, that is a worthwhile and appropriately evidenced outcome.

6.6 Limitations

Several limitations cap the strength of the conclusions.

- The report does not include raw code, schemas, or full runtime logs, so some implementation detail necessarily remains abstract.
- Quantitative evidence for tests and CI is bounded to sanitised Bamboo summaries and logs rather than raw exported reports or screenshots.
- The evaluation is strongest at the level of service capability, testability, and delivery practice. It is weaker on long-term adoption or large-scale operational evidence, which would require additional internal data not provided here.
- The services are presented as bounded engineering interfaces around an inherited system, not as proof that the wider autonomy stack is fully solved or fully observable.
- Human-facing runtime presentation and usability questions are intentionally out of scope here because they belong to the separately submitted WPS dashboard report rather than to the WPA4 software product claim.

7 Conclusions and Recommendations

This project began with a genuine inherited-system problem: a complex autonomous maritime stack that was difficult to interrogate, difficult to stimulate deliberately, and too dependent on tacit operational knowledge. The resulting report presents a stronger and more coherent software engineering answer to that problem.

The main outcome was not a new autonomy algorithm. It was bounded engineering leverage around an inherited autonomy stack: a reusable read-side query interface, a reusable write-side injection interface, and the testability and delivery discipline needed to make both maintainable. The DDS Query Service made selected runtime state more accessible for routine engineering questions. The Contact Injection Service made synthetic nearby-contact setup more deliberate and repeatable. Automated tests, mock-mode execution, and Bamboo/Docker/Harbor integration strengthened those interfaces as software artefacts rather than one-off local utilities. Just as importantly, the project resolved several practical implementation problems along the way: keeping outward contracts narrower than the DDS-facing internals, supporting routine development without a full live runtime, and making deliberate nearby-contact setup reusable without pretending to replace broader mission-authoring workflows. The earlier SIL and configuration work remained valuable, but only as enabling context for that service-centred contribution.

Taken together, the evidence supports a clear but bounded claim. The project improved interrogability and controlled experimentation around the inherited stack through reusable query and injection interfaces, and it made those interfaces more credible through testability and delivery discipline. It does not claim autonomy-algorithm novelty, exhaustive live-runtime proof, or large-scale adoption. That narrower claim is also the better evidenced one.

This also clarifies the distinction from my separately submitted WPS work. WPS addressed browser-based, human-readable runtime interpretation through a lightweight dashboard and usability-led evaluation. WPA4 instead addresses reusable service boundaries, repeatable validation, and CI-backed delivery around the inherited autonomy stack.

7.1 Recommendations

The most important next actions are:

- Extend query and injection scope cautiously, preserving the principle of bounded, safe abstractions rather than exposing the entire middleware surface at once.
- Formalise a small set of stable service contracts for the most common engineering tasks (for example, own-ship state inspection, nearby-contact inspection, and repeatable synthetic-contact recipes) so downstream tooling can depend on them more confidently.
- Keep workflow, configuration, and build artefacts versioned alongside the service code where feasible, so the supporting engineering knowledge evolves with the implementation rather than drifting away from it.
- Use the existing mission and SIL artefacts as a repeatable integration scaffold when extending either service, so that new interface functionality is checked against recognisable scenarios rather than only through isolated local testing.

7.2 Reflection

The most useful lesson from this project was that in inherited systems, strong software engineering often means building better *interfaces around complexity* rather than replacing the complexity outright. Under confidentiality constraints, that lesson becomes even sharper: the artefacts that matter most are often the ones that improve reproducibility, diagnosability, and safe delivery without requiring the whole internal system to be disclosed.

A second lesson was that supporting artefacts are most valuable when they are designed to survive abstraction. The earlier SIL workflow, mission scaffolds, and configuration summaries were initially useful because they helped me work in the inherited environment; they became more valuable once they also helped other engineers position and interpret the new services. That changed how I think about engineering documentation in constrained environments: not as an afterthought to implementation, but as part of the interface through which later engineers will understand what a piece of software is really doing.

References

[1] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[2] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[3] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2149–2154, 2004.

[4] Thomas Nakken Larsen, Amalie Heiberg, Eivind Meyer, Adil Rasheed, Omer Sand, and Damiano Varagnolo. Risk-based implementation of COLREGs for autonomous surface vehicles using deep reinforcement learning. *arXiv preprint*, 2021.

[5] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. In *Proceedings of the 2010 ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2010.

[6] A. Tsolakis, D. Benders, O. de Groot, R. R. Negenborn, V. Reppa, and L. Ferranti. COLREGs-aware trajectory optimization for autonomous surface vessels. *IFAC PapersOnLine*, 55(31):269–274, 2022.

[7] Guan Wei and Wang Kuo. COLREGs-compliant multi-ship collision avoidance based on multi-agent reinforcement learning technique. *Journal of Marine Science and Engineering*, 10(10):1431, 2022.

[8] Adam Wiggins. *The Twelve-Factor App*. Web publication, 2011. Available: https://12factor.net/.

[9] 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.