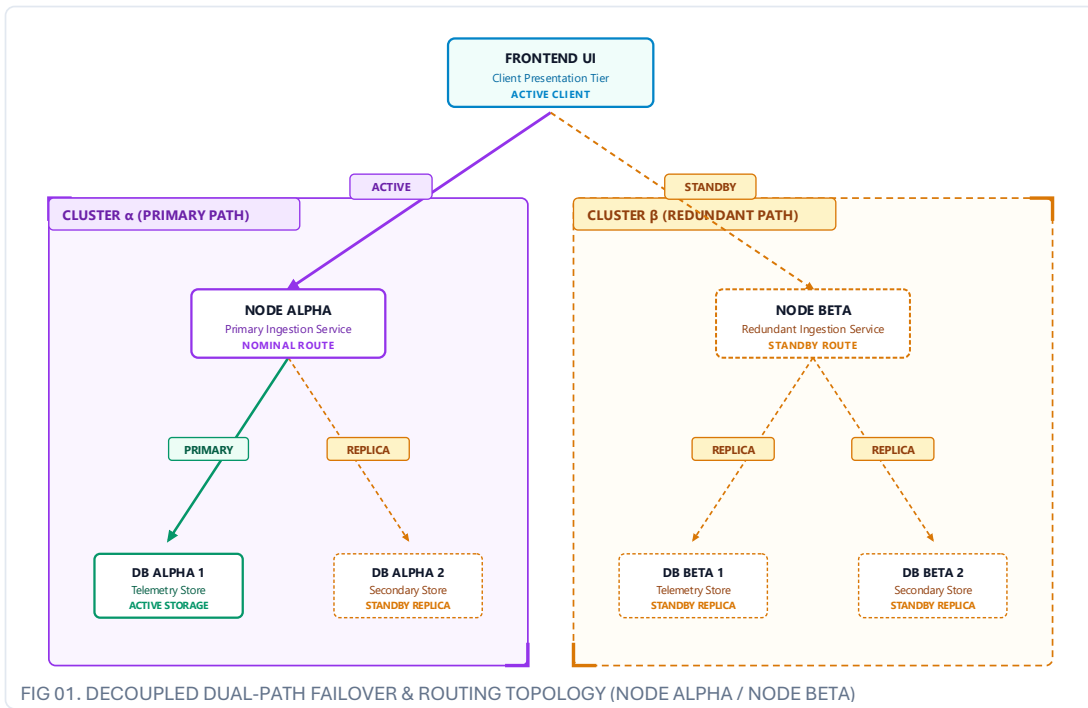


O.R.V.A.L.

Orchestrator for Remote Visualization and Analysis of Logged Telemetry

DETERMINISTIC, FAULT-TOLERANT TELEMETRY VISUALIZATION FOR SPACE TESTING & OPERATIONAL ENVIRONMENTS

AUTONOMOUS FAILURE DETECTION, DETERMINISTIC FAILOVER, AND
COORDINATED RETURN-TO-HOME RECOVERY IN REDWIRE GROUND
OPERATIONS.



AUTHOR / CANDIDATE:

Jelle Smet

Applied Data Intelligence

Thomas More University of Applied Sciences

Academic Coordinator: Elke Boonen

INDUSTRY PARTNER & CONTEXT:

Redwire Europe

Ground Segment Engineering Division

Mentors: Șerban Anghel, Stijn Ilse

Submitted for Consideration: Vlaamse Scriptieprijs

2025-2026

Abstract

Reliable access to telemetry data is essential during satellite testing and in-orbit operations, where engineers depend on visualization systems to monitor spacecraft behaviour and evaluate test results. When the backend service supporting such a system becomes unavailable, however, telemetry access can be interrupted, and recovery may depend on manual intervention. This thesis investigates how a deterministic, fault-tolerant telemetry visualization framework can be designed and implemented for a space testing environment, using ORVAL (Orchestrator for Remote Visualization and Analysis of Logged telemetry) as a practical implementation developed for Redwire.

ORVAL integrates interactive telemetry retrieval and visualization with autonomous failure detection, deterministic failover, recovery monitoring, and coordinated service routing. The architecture uses redundant backend and database paths and applies predefined decision rules to determine when a service should be considered unavailable and when the preferred service path can safely be restored. This approach embeds fault tolerance directly into the visualization workflow rather than treating service failures as a separate operational procedure.

The resulting framework was deployed and evaluated in the intended space ground-segment environment through controlled failure scenarios involving backend and database services. The validation demonstrated that ORVAL could autonomously transition to redundant service paths, maintain the intended routing behaviour across concurrent users, and automatically return to the preferred path after verified recovery. Across repeated failure test scenarios, the transition to the redundant backend consistently executed within the configured 10-25 second detection window, with a single instrumented trace completing failover in approximately 12 seconds. The system was developed for Redwire as a replacement for the existing WebMUST-based telemetry visualization workflow. While validation was performed within testing environments, the framework's autonomous fault tolerance and deterministic failover are specifically designed to meet the rigorous availability requirements of live satellite mission operations.

The results demonstrate that deterministic failure handling, redundant service paths, autonomous recovery, and coordinated routing can be integrated into a telemetry visualization framework to improve service continuity without requiring users to manually initiate service failover. This capability is particularly significant in satellite operations, where communication with a spacecraft is inherently intermittent, and contact windows are often limited to only a few minutes at a time, typically around 10 minutes every few hours. During these short windows, reliable access to telemetry is essential for monitoring spacecraft status and supporting operational decision-making. A temporary loss of access can therefore have consequences beyond a conventional service interruption, as the next opportunity to retrieve or assess critical data may not occur for several hours. By enabling the system to detect failures and autonomously transition to an alternative service path, the proposed approach reduces reliance on manual intervention precisely when access to telemetry is most constrained. Beyond the specific aerospace application, the separation between these mechanisms and the underlying data-access layer provides a basis for adapting the approach to other structured operational data sources.

Acknowledgements

I would like to express my sincere gratitude to everyone who contributed to the development of this thesis and to the internship experience on which it was built.

First and foremost, I would like to thank Redwire for giving me the opportunity to complete my internship within a space engineering environment and for providing me with the opportunity to develop ORVAL as a project with a genuine operational purpose. The experience allowed me to work on a real engineering challenge and to develop and evaluate the system in the environment for which it was intended.

I am especially grateful to Șerban Anghel, my internship mentor, for his guidance, technical insight, and continued support throughout the project. His trust and willingness to give me the freedom to develop ORVAL independently while providing guidance when needed played an important role in the development of the project. I would also like to thank Stijn Ilsen for giving me the opportunity to join Redwire, for his involvement throughout the internship, and for his continued technical feedback. His participation in the project's key meetings and milestones provided valuable perspectives throughout the development of ORVAL.

I would also like to thank the members of the Redwire engineering team for their willingness to answer my questions, share their expertise, and provide feedback throughout the development and testing of the system. Their practical input helped shape ORVAL into a solution grounded in the needs of its intended users.

From Thomas More, I would particularly like to thank Elke Boonen, coordinator of the Applied Data Intelligence programme, for helping establish the connection with Redwire that made this internship possible. I am also grateful for her continued support throughout the project and for taking the time to attend the official presentation of ORVAL.

Finally, I would like to thank Thomas More for the education and opportunities that provided the foundation for this project and for the knowledge and skills developed throughout the Applied Data Intelligence programme.

To everyone who contributed to this journey, whether through technical guidance, professional support, or simply being there along the way, thank you.

Table of Contents

ABSTRACT	2
ACKNOWLEDGEMENTS	4
TABLE OF CONTENTS	5
LIST OF FIGURES	9
LIST OF TABLES	10
1. INTRODUCTION	11
1.1. BACKGROUND	11
1.2. PROBLEM STATEMENT	12
1.3. RESEARCH OBJECTIVE	13
1.4. RESEARCH QUESTION	14
1.5. RESEARCH SUBQUESTIONS	14
1.6. SCOPE	15
1.7. THESIS STRUCTURE	17
2. CONTEXT & PROBLEM ANALYSIS	19
2.1. TELEMETRY IN SPACE SYSTEMS	19
2.2. TELEMETRY DATA VISUALIZATION	20
2.3. EXISTING INFRASTRUCTURE	23
2.4. EXISTING FAILURE SCENARIOS	25
2.5. LIMITATIONS OF THE EXISTING APPROACH	28
2.6. OPERATIONAL REQUIREMENTS	31
2.7. PROBLEM DEFINITION	35

3. THEORETICAL BACKGROUND & STATE OF THE ART	37
3.1. TELEMETRY VISUALIZATION	37
3.2. HIGH AVAILABILITY	38
3.3. FAULT TOLERANCE	38
3.4. REDUNDANCY	39
3.5. HEALTH MONITORING	40
3.6. HEARTBEAT MECHANISMS	41
3.7. FAILOVER STRATEGIES	42
3.8. STATE-BASED FAILURE DETECTION	43
3.9. DETERMINISTIC DECISION-MAKING	45
3.10. EXISTING APPROACHES	46
3.11. LIMITATIONS IN THE SPECIFIC CONTEXT	49
3.12. RESEARCH GAP & DESIGN IMPLICATIONS	51
4. REQUIREMENTS & SYSTEM ANALYSIS	54
4.1. REQUIREMENTS ELICITATION AND REFINEMENT	54
4.2. FUNCTIONAL REQUIREMENTS	56
4.3. NON-FUNCTIONAL REQUIREMENTS	59
4.4. AVAILABILITY AND FAULT-TOLERANCE REQUIREMENTS	61
4.5. CONSISTENCY AND MULTI-USER REQUIREMENTS	63
4.6. REQUIREMENT PRIORITIZATION	65
4.7. REQUIREMENT TRACEABILITY	68
5. ORVAL ARCHITECTURE & DESIGN	72
5.1. DESIGN GOALS	72
5.2. OVERALL SYSTEM ARCHITECTURE	74
5.3. FRONTEND ARCHITECTURE	76
5.4. BACKEND ARCHITECTURE	77
5.5. DATABASE ARCHITECTURE	78
5.6. HEALTH MONITORING ARCHITECTURE	80
5.7. FAILURE DETECTION	81
5.8. AUTONOMOUS FAILOVER	83
5.9. RETURN-TO-HOME MECHANISM	84
5.10. USER ROUTING & CONSISTENCY	86
5.11. STATE MANAGEMENT	87
5.12. DESIGN DECISIONS & TRADE-OFFS	88

6. IMPLEMENTATION	92
6.1. TECHNOLOGY STACK	92
6.2. DATABASE IMPLEMENTATION	94
6.3. BACKEND IMPLEMENTATION	96
6.4. FRONTEND IMPLEMENTATION	97
6.5. HEALTH MONITORING IMPLEMENTATION	98
6.6. DETERMINISTIC FAILURE DETECTION	100
6.7. AUTONOMOUS FAILOVER AND ROUTING IMPLEMENTATION	101
6.8. AUTONOMOUS RETURN-TO-HOME IMPLEMENTATION	102
6.9. LOGGING AND OBSERVABILITY	103
6.10. DEPLOYMENT ENVIRONMENT	104
6.11. SECURITY AND OPERATIONAL CONSIDERATIONS	105
7. ITERATIVE DEVELOPMENT & DESIGN EVOLUTION	107
7.1. INITIAL PROTOTYPE	107
7.2. DOMAIN EXPERT FEEDBACK & IDENTIFIED PROBLEMS	109
7.3. DESIGN ADAPTATIONS	111
7.4. FINAL ITERATION	115
7.5. LESSONS LEARNED	116
8. VALIDATION & RESULTS	118
8.1. VALIDATION METHODOLOGY	118
8.2. NORMAL OPERATION	122
8.3. BACKEND FAILURE & AUTONOMOUS FAILOVER	123
8.4. BACKEND RECOVERY & RETURN-TO-HOME	125
8.5. MULTIPLE-USER CONSISTENCY	126
8.6. REPEATED FAILURE SCENARIOS	127
8.7. PERFORMANCE AND TIMING ANALYSIS	129
8.8. REQUIREMENT VALIDATION	132
8.9. RESULTS SUMMARY	136
9. DISCUSSION	138
9.1. INTERPRETATION OF RESULTS	138
9.2. STRENGTHS AND TRADE-OFFS	140
9.3. LIMITATIONS	141

10. CONCLUSION	144
10.1. ANSWER TO THE RESEARCH QUESTION	144
10.2. MAIN FINDINGS	146
10.3. CONTRIBUTION	147
10.4. FINAL PERSPECTIVE	148
REFERENCES	150

List of Figures

<i>Figure 1: Simplified MUST/WebMUST architecture relevant to the telemetry visualization workflow (adapted from Baumgartner et al., 2005; European Space Agency (ESA), 2024)</i>	23
<i>Figure 2: Finite state machine representation of the fault detection and component recovery lifecycle.</i>	44
<i>Figure 3: Iterative requirements engineering lifecycle and formal CDR validation gating.</i>	56
<i>Figure 4: High-level reference architecture of ORVAL.</i>	74
<i>Figure 5: Autonomous failover flow from the primary service path to the redundant service path.</i>	83
<i>Figure 6: Autonomous recovery process from the redundant service path back to the primary service path</i>	85
<i>Figure 7: ORVAL frontend state lifecycle and corresponding route state.</i>	88
<i>Figure 8: ORVAL dashboard displaying multiple telemetry visualizations during nominal operation.</i>	122
<i>Figure 9: Observed Alpha-to-Beta failover sequence during controlled backend failure testing.</i>	124

List of Tables

Table 1: Mapping of existing approaches to operational conditions.	48
Table 2: Comparative analysis of standard fault-tolerance approaches and their specific limitations within the ORVAL framework.	50
Table 3: Key architectural requirements and functional objectives of the fault-tolerant telemetry framework.	52
Table 4: Functional requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.	56
Table 5: Non-functional requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.	59
Table 6: Availability and fault-tolerant requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.	61
Table 7: Consistency and multi-user requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.	63
Table 8: MoSCoW prioritization of ORVAL functional, non-functional, availability, and consistency requirements.	65
Table 9: Traceability between ORVAL requirements and subsequent system development and validation activities.	68
Table 10: Overview of the main architectural decisions and trade-offs in ORVAL	89
Table 11: ORVAL technology stack and implementation roles	92
Table 12: End-to-end system validation scenarios and verification objectives across nominal and degraded states.	119
Table 13: Summary of the repeated scenarios and their observed outcomes.	127
Table 14: Summary of configured and observed timing metrics across failover and recovery operations.	129
Table 15: Evidence classification and empirical observation profile for backend failover testing.	130
Table 16: Validation of functional, availability, fault-tolerance, and consistency requirements.	132
Table 17: Validation of non-functional requirements.	134

1. Introduction

1.1. Background

Modern space systems rely heavily on telemetry data to monitor the behaviour, status, and performance of spacecraft and their subsystems (Fortescue, Swinerd, & Stark, 2011). During development and testing, engineers use telemetry to verify system behaviour, identify anomalies, and evaluate whether a system performs according to its intended specifications. As space systems become increasingly complex, the amount and variety of telemetry data that must be processed and interpreted also increase. Reliable access to this information is therefore a critical requirement, not only during the engineering and testing process, but ultimately during live satellite operations where continuous monitoring is vital for mission success.

Telemetry data is typically presented to engineers through specialized visualization systems that transform large volumes of numerical and technical data into information that can be monitored and interpreted in real time (ECSS, 2016). These systems allow engineers to observe parameters, identify unexpected behaviour, and investigate events occurring within the system under test. Telemetry visualization therefore forms an important interface between complex space systems and the engineers responsible for analysing their behaviour.

However, in a space environment, providing access to information is only part of the challenge. The systems used to monitor and test space hardware must themselves be designed with reliability and fault tolerance in mind. The availability of telemetry visualization therefore depends not only on the availability of the underlying data, but also on the reliability of the software infrastructure through which engineers access and interpret that data.

Redundancy is therefore an important principle within reliable space systems. By providing alternative components that can take over when a primary component becomes unavailable, redundant architectures can reduce the impact of individual failures and increase system availability (Tanenbaum & Van Steen, 2017). The same principle can be applied to the software infrastructure supporting telemetry visualization. Rather than treating visualization and infrastructure reliability as separate concerns, a robust telemetry system can combine both: engineers require not only a clear representation of telemetry data, but also confidence that this representation remains available when individual components fail.

Nevertheless, redundancy alone does not guarantee continuous operation. A redundant system must be able to determine when a component has become unavailable, decide when a transition should occur, and redirect users to an operational component. Recovery introduces an additional challenge. When a failed primary component becomes available again, the system must determine how users should be routed and how the overall system should return to a consistent state. This becomes particularly relevant when multiple users are connected simultaneously. If some users remain connected to a redundant component while others are redirected to a recovered primary component, different users may interact with different instances of the system.

These considerations illustrate that telemetry visualization, availability, and redundancy are closely interconnected within a space testing environment. The visualization layer provides engineers with access to the information required to analyse a system, while the underlying infrastructure must ensure that this access remains dependable in the presence of failures. Designing such a system consequently involves more than displaying telemetry data: it also requires mechanisms for health monitoring, failure detection, autonomous failover, and coordinated recovery.

This thesis investigates the challenges of maintaining telemetry visualization, system availability, and deterministic redundancy within the context of space testing and satellite operations at Redwire. Specifically, it presents the design, implementation, and empirical validation of ORVAL, a telemetry visualization framework engineered to provide continuous telemetry access across redundant backend infrastructures, coupling autonomous failure detection and deterministic failover with a coordinated return-to-home recovery mechanism that reliably restores the primary service path once component health is verified.

The development of such a system raises questions concerning the requirements that a resilient telemetry visualization system must satisfy, how these requirements can be translated into appropriate architecture, and whether the resulting system can maintain its intended functionality, availability, and consistency under failure conditions. These questions form the basis of the research presented in this thesis.

1.2. Problem Statement

Providing engineers with continuous access to telemetry data requires a distributed software architecture whose components must operate dependably under rigorous operational constraints. In such an environment, the visualization interface, backend ingestion services, and underlying database clusters form an interconnected pipeline where the availability of the monitoring interface is inherently vulnerable to the failure of any underlying component. Designing an architecture that ensures seamless, real-time telemetry access therefore requires robust fault-handling mechanisms to prevent individual component failures from disrupting engineering oversight.

Within such an architecture, individual components can represent potential points of failure. A failure of a backend service, for example, can prevent users from retrieving telemetry data even when the underlying system under test continues to operate normally. Similarly, a loss of connectivity between a backend and its database can make the corresponding visualization functionality unavailable. As a result, a failure in the supporting software infrastructure can effectively interrupt access to telemetry without necessarily indicating a failure of the space system being monitored.

The consequences of such an infrastructure failure extend beyond the affected software component itself. When engineers lose access to telemetry visualization during testing or active flight operations, they may temporarily lose the ability to monitor critical system parameters, identify unexpected behaviour, or analyse ongoing events. In a live mission scenario, even a temporary loss of visibility poses significant operational risks. Restoring access may therefore become necessary before testing and monitoring can continue effectively. In an environment where telemetry is used to support engineering decisions, even a temporary loss of visibility can disrupt the testing process.

In the absence of an autonomous recovery mechanism, responding to such failures may require manual intervention. An operator may need to identify the unavailable component, determine the cause of the failure, restore the affected service or connection, and subsequently verify that users can access the telemetry visualization system again. This introduces additional recovery time and creates a dependency on human intervention. Furthermore, when multiple users are connected simultaneously, manually recovering individual components does not necessarily guarantee that all users are returned to the same operational state.

The problem addressed by this thesis is therefore the difficulty of maintaining reliable and consistent access to telemetry visualization when components of the underlying software infrastructure become unavailable. Addressing this problem requires more than simply providing redundant components. The system must be capable of detecting component failures, determining when a failover should occur, redirecting users to available infrastructure, and coordinating the recovery of the primary system when it becomes available again. The challenge consequently lies in combining telemetry visualization with an infrastructure architecture capable of autonomously handling failures while maintaining system availability and consistent user access.

1.3. Research Objective

The objective of this thesis is to design and implement a deterministic, fault-tolerant telemetry visualization framework capable of maintaining reliable and consistent access to telemetry data in the presence of backend failures. To achieve this objective, this thesis presents the design and development of ORVAL, a framework that combines telemetry visualization with redundant software infrastructure and autonomous failure-handling mechanisms.

The development of ORVAL focuses on translating the requirements of a space testing environment into an architecture that can monitor the health of its backend components, detect failures, autonomously perform failover, and coordinate the recovery of the primary system. Particular attention is given to maintaining consistent user access during both failure and recovery scenarios, ensuring that multiple users are not unintentionally routed to different system states.

The resulting framework is evaluated through controlled failure scenarios to determine whether the implemented mechanisms provide the intended levels of availability and consistency. The objective is therefore not only to produce a functional telemetry visualization framework, but also to assess whether a deterministic and fault-tolerant architecture can reliably maintain access to telemetry data when backend failures occur.

1.4. Research Question

The problem and research objective defined in the previous sections lead to a central research question concerning the design of a resilient telemetry visualization framework. The question focuses on how telemetry visualization can remain reliable and consistent when failures occur within the backend infrastructure, while taking into account the deterministic behaviour required to manage failure and recovery scenarios.

The central research question of this thesis is therefore:

Research Question

How can a deterministic, fault-tolerant telemetry visualization framework be designed and implemented to provide reliable and consistent access to telemetry data in the presence of backend failures?

Answering this question requires consideration of both the architectural and operational aspects of the framework. This includes determining the requirements for failure detection and failover, designing mechanisms for maintaining consistent user access during failures and recovery, implementing the resulting architecture, and evaluating its behaviour under controlled failure scenarios.

1.5. Research Subquestions

The central research question encompasses several aspects of the design and evaluation of a fault-tolerant telemetry visualization framework. To structure the research and ensure that each aspect of the problem is addressed systematically, the central research question is divided into three research subquestions. These questions respectively address the requirements that the framework must satisfy, the architectural approach used to satisfy these requirements, and the extent to which the resulting system meets the identified requirements.

1.5.1. Subquestion 1: Requirements

What functional, reliability, availability, redundancy, and consistency requirements must a fault-tolerant telemetry visualization system satisfy in a space testing environment?

This question establishes the requirements that form the basis for the subsequent design and implementation of the framework. It considers both the functional capabilities required by the telemetry visualization environment and the reliability-related properties necessary to maintain access to telemetry data during component failures.

1.5.2. Subquestion 2: Architecture

How can an architecture be designed to provide autonomous failure detection, deterministic failover, and coordinated recovery while satisfying these requirements?

This question focuses on translating the identified requirements into an appropriate system architecture. Particular attention is given to the mechanisms required to monitor backend health, detect failures, perform deterministic failover, maintain consistent user routing, and coordinate the recovery of the primary system.

1.5.3. Subquestion 3: Validation

To what extent does the resulting system satisfy the identified requirements under controlled failure scenarios?

This question evaluates the resulting implementation against the requirements established in research subquestion 1. Controlled failure scenarios are used to assess whether the framework maintains its intended functionality, availability, and consistency during backend failures and subsequent recovery.

Together, these research subquestions provide a structured approach to answering the central research question. The first question establishes what the system must achieve, the second investigates how these requirements can be addressed through system architecture and implementation, and the third evaluates whether the resulting framework achieves the intended properties in practice.

1.6. Scope

This thesis focuses on the design, implementation, and evaluation of ORVAL as a fault-tolerant telemetry visualization framework for use in a space testing environment. The scope of the project covers the software infrastructure required to provide engineers with access to telemetry data and to maintain this access when individual backend components become unavailable. This includes the visualization interface, backend services, database connectivity, backend health monitoring, failure detection, autonomous failover, coordinated recovery, and user routing.

The research specifically focuses on strategies and architectural mechanisms to counteract and manage failures occurring within the software infrastructure supporting the telemetry visualization system. The framework is designed to detect the unavailability of backend components, autonomously redirect users to redundant infrastructure, and coordinate the return to the primary system when the failed component becomes available again. Particular attention is given to deterministic failure-handling behaviour and to maintaining consistent user access during both failure and recovery scenarios.

The evaluation of ORVAL is performed through controlled failure scenarios within a development and testing environment. These scenarios are intended to reproduce relevant backend and infrastructure failures in a controlled and repeatable manner. The evaluation therefore assesses the behaviour of the implemented framework under defined failure conditions rather than claiming reliability under all possible operational conditions. Although ORVAL is intended for use within Redwire's space testing environment, this thesis focuses on the software architecture and behaviour of the framework rather than prescribing the exact physical infrastructure and production configuration in which it must be deployed.

The scope of the thesis is also limited to the software-level resilience mechanisms implemented within ORVAL. The project does not attempt to address the reliability or fault-tolerance of the spacecraft, its physical subsystems, sensors, communication hardware, or other hardware components. Similarly, ORVAL does not constitute a complete solution for space system reliability. Its purpose within this research is to provide a resilient software layer for telemetry visualization and access within the defined testing environment.

Database behaviour is considered only insofar as it affects the availability and operation of the telemetry visualization framework. The thesis does not investigate database replication strategies, distributed database consistency, or database-level fault tolerance as independent research topics. Where redundant databases are used as part of the evaluation environment, they are treated as infrastructure components available to the framework rather than as the subject of the research itself.

Finally, the research does not provide a comprehensive assessment of network reliability, cybersecurity, or the security properties of authentication and authorization mechanisms. Authentication and authorization functionality implemented within ORVAL may be described where relevant to the system architecture and user interaction, but these mechanisms are not evaluated as independent security topics. Similarly, network behaviour is considered only insofar as it affects the operation of the framework and its failure-handling mechanisms.

1.7. Thesis Structure

The remainder of this thesis is structured into nine chapters, each addressing a specific aspect of the research and contributing to answering the central research question.

1.7.1. Context & Problem Analysis

Chapter 2 – Context & Problem Analysis describes the space testing context in which ORVAL is situated. It introduces telemetry and its visualization, describes the existing software infrastructure, and identifies the failure scenarios and limitations that motivate the development of a fault-tolerant solution. The chapter concludes by defining the specific problem addressed by the thesis.

1.7.2. Theoretical Background & State of the Art

Chapter 3 – Theoretical Background & State of the Art establishes the theoretical foundation of the research. It discusses concepts including telemetry visualization, high availability, fault tolerance, redundancy, health monitoring, heartbeat mechanisms, failover strategies, and deterministic decision-making. Existing approaches are examined and compared to identify their applicability and limitations within the specific context of this thesis. The chapter concludes by deriving the research gap and the resulting design implications for ORVAL.

1.7.3. Requirements & System Analysis

Chapter 4 – Requirements & System Analysis establishes the requirements that the framework must satisfy. Requirements are derived from the space testing environment, existing infrastructure, domain expertise, technical analysis, and relevant literature. The requirements are categorized and prioritized according to their functional and non-functional properties, with particular attention to availability, reliability, redundancy, failover, and consistency. A requirements traceability matrix establishes the relationship between requirements, design decisions, implementation, and subsequent validation.

1.7.4. ORVAL Architecture & Design

Chapter 5 – ORVAL Architecture & Design presents the architecture and design of the proposed framework. The chapter describes the overall system architecture and the individual frontend, backend, and database components. It further details the health monitoring, failure detection, deterministic failover, return-to-home, user routing, and state management mechanisms. Major architectural decisions and their associated trade-offs are also discussed.

1.7.5. Implementation

Chapter 6 – Implementation describes how the proposed architecture was realized in software. The technology stack and implementation of the database, backend, frontend, health monitoring, failure detection, failover, recovery, routing, and logging mechanisms are presented. The chapter also describes the deployment environment and relevant operational considerations.

1.7.6. Iterative Development & Design Evolution

Chapter 7 – Iterative Development & Design Evolution describes how ORVAL evolved throughout the development process. Initial prototypes, feedback from domain experts, identified problems, and subsequent design adaptations are presented. The chapter demonstrates how practical findings and expert feedback influenced the architectural and implementation decisions leading to the final version of the framework.

1.7.7. Validation & Results

Chapter 8 – Validation & Results evaluates the implemented framework against the requirements established in Chapter 4. Controlled failure scenarios are used to assess normal operation, failure detection, deterministic failover, backend recovery, return-to-home behaviour, and multi-user consistency. Performance measurements and repeated test scenarios are included where applicable. The chapter concludes with a systematic validation of the identified requirements.

1.7.8. Discussion

Chapter 9 – Discussion interprets the results obtained during the validation and discusses their implications for the research. The strengths and limitations of the resulting framework are examined, together with relevant architectural trade-offs and operational considerations. The chapter also addresses the extent to which the research questions have been answered and identifies meaningful opportunities for future improvement and broader application.

1.7.9. Conclusion

Chapter 10 – Conclusion provides the final answer to the central research question and summarizes the main findings and contributions of the thesis. The chapter concludes by reflecting on the significance of the developed framework within the context of fault-tolerant telemetry visualization.

Together, these chapters follow the progression from problem identification and theoretical analysis, through requirements and system design, to implementation, validation, and critical evaluation. This structure provides a systematic approach to investigating the research question and assessing the resulting ORVAL framework.

2. Context & Problem Analysis

This chapter examines the context in which ORVAL is developed and defines the problem addressed by this thesis. It first introduces telemetry and its role within space systems, followed by an examination of telemetry visualization and the existing software infrastructure. The chapter then analyses relevant failure scenarios and limitations of the existing approach before introducing the operational requirements that guide the development of the proposed framework. Finally, the chapter concludes with a precise definition of the problem addressed by the research.

2.1. Telemetry in Space Systems

2.1.1. Definition and Purpose of Telemetry

Telemetry is a fundamental source of information for monitoring spacecraft and their subsystems. It consists of measurements and status information generated by a spacecraft and transmitted to ground-based systems, where the information can be displayed, distributed, stored, and analysed. Telemetry therefore provides engineers with information about the operational condition and behaviour of spacecraft systems during both testing and operation (Fortescue, Swinerd, & Stark, 2011; NASA, 2024).

2.1.2. Telemetry Parameters

Telemetry consists of numerous individual parameters that describe different aspects of spacecraft and subsystem behaviour. Depending on the system being monitored, these parameters can represent physical measurements such as temperature, pressure, voltage, and current, as well as discrete subsystem states and other operational information. The collection of these parameters provides engineers with a representation of the current state of the spacecraft and allows deviations from expected behaviour to be identified (Fortescue, Swinerd, & Stark, 2011).

2.1.3. Raw and Calibrated Telemetry Values

Telemetry measurements may require processing before they can be interpreted directly by engineers. Raw telemetry values represent measurements in their transmitted or unprocessed representation, whereas calibrated or converted values express these measurements in engineering units that are meaningful for analysis. Telemetry processing can therefore involve converting transmitted data into physical quantities such as temperature, voltage, or pressure, allowing engineers to interpret the behaviour represented by the original measurement (ECSS, 2016).

2.1.4. Telemetry During Development and Testing

Telemetry is not limited to operational spacecraft missions but is also used extensively during spacecraft development and testing. During these activities, telemetry allows engineers to monitor hardware and subsystems, verify expected behaviour, and investigate unexpected events. Telemetry can be generated by spacecraft hardware as well as by ground equipment and simulation environments, providing a common source of information for analysing system behaviour throughout different stages of development and verification (ECSS, 2016).

2.2. Telemetry Data Visualization

2.2.1. Existing Visualization Workflow

Telemetry data generated during spacecraft testing and operations must be made accessible to engineers in a form that supports monitoring and analysis. This requires more than simply storing telemetry measurements: engineers need mechanisms to retrieve relevant parameters, select appropriate time intervals, and visualize the resulting data. The ESA Mission Utility and Support Tools (MUST) ecosystem, for example, provides infrastructure for importing, processing, storing, and accessing spacecraft telemetry and related mission data, while client applications provide functionality for visualization, analysis, and monitoring (European Space Agency (ESA), 2024).

Within the MUST ecosystem, WebMUST provides a web-based interface through which spacecraft data can be interactively explored and visualized. The application allows users to plot and export spacecraft data through a web browser and provides access to time-series data stored by the underlying MUST infrastructure. Its web-based architecture also allows users to access spacecraft data remotely without requiring a dedicated rich-client application (Solenix, 2026).

A typical telemetry visualization workflow therefore involves selecting the parameters of interest, defining the relevant time interval, retrieving the corresponding data, and presenting the resulting information in a suitable visualization. WebMUST supports this workflow through interactive plotting of spacecraft data and provides additional representations for numerical time series and other mission data, including alphanumeric and tabular displays (Solenix, 2026).

The separation between the visualization layer and the underlying data infrastructure is an important characteristic of this type of system. In MUST, the server infrastructure is responsible for data import, processing, and storage, while client applications retrieve the required information through standardized interfaces. This architecture allows visualization and analysis functionality to be provided independently of the underlying mechanisms used to store and process the telemetry data (European Space Agency (ESA), 2024).

2.2.2. Real-Time Monitoring

Telemetry visualization can be used to monitor the current behaviour and operational state of spacecraft systems. The ECSS standard for telemetry and telecommand packet utilization identifies telemetry as a mechanism for remote monitoring and control of spacecraft subsystems and payloads during spacecraft integration, testing, and flight operations (European Cooperation for Space Standardization (ECSS), 2016).

Real-time or near-real-time visualization allows engineers to observe changes in telemetry parameters as new measurements become available. This provides direct visibility into the behaviour of the system being monitored and can support the identification of unexpected changes or events. The MUST ecosystem also provides monitoring functionality in addition to its visualization for operational monitoring activities (European Space Agency (ESA), 2024).

The ability to monitor telemetry continuously is particularly relevant during testing, where engineers may need to observe system behaviour while a test is being executed. In such situations, visualization provides an interface through which changes in relevant parameters can be observed and interpreted without requiring engineers to inspect individual telemetry records manually.

2.2.3. Historical Telemetry Data

Telemetry visualization is not limited to data that is currently being received. Historical telemetry allows engineers to investigate events that have already occurred, examine parameter behaviour over extended periods, and compare measurements from different points in time. The MUST ecosystem was specifically developed to provide efficient access to spacecraft parameter data and supports both visualization and analysis of stored time-series data (Solenix, 2026).

WebMUST supports the visualization of large amounts of historical spacecraft data. Its Viewpoint Resolution Aggregation technique enables efficient plotting of extensive datasets, including telemetry parameter behaviour across time ranges that would otherwise contain very large numbers of individual measurements (Solenix, 2026).

Historical telemetry is particularly relevant when investigating unexpected behaviour or anomalies. By examining the evolution of parameters before, during, and after an event, engineers can identify changes in system behaviour and investigate relationships between different measurements. ESA's telemetry analysis technologies similarly use historical telemetry data to identify anomalous behaviour and investigate relationships between parameters involved in an event (European Space Agency (ESA), 2026).

2.2.4. Visualization of Multiple Parameters

Spacecraft telemetry consists of large numbers of parameters representing different aspects of system behaviour. Visualization systems must therefore support the analysis of multiple parameters rather than restricting engineers to isolated measurements. MUST provides access to large collections of spacecraft parameter data and includes functionality for visualizing and analysing time-series information (Solenix, 2026).

Displaying multiple parameters within a common temporal context allows engineers to compare their behaviour and identify relationships between measurements. WebMUST supports comparison of telemetry data, while related tools within the MUST ecosystem, such as DrMUST, provide additional capabilities for identifying correlations between telemetry parameters during anomaly investigation (Solenix, 2026).

The number of parameters and measurements involved can also create challenges for telemetry visualization systems. Large telemetry archives may contain substantial numbers of data points, particularly when parameters are sampled at high frequencies or examined over long periods. WebMUST therefore incorporates mechanisms designed to efficiently visualize large quantities of telemetry data while maintaining an interpretable representation of the underlying time series (Solenix, 2026).

2.2.5. User Interaction

Telemetry visualization systems provide engineers with interactive mechanisms for exploring and analysing data rather than presenting telemetry as static information. Users may need to select parameters, define time ranges, inspect values, compare data series, and choose appropriate representations depending on the analysis being performed. WebMUST was designed as an interactive web client and supports interactive plotting and exporting of spacecraft data through a web browser (Solenix, 2026).

User interaction can also extend beyond basic graph creation. WebMUST provides different representations of time-series data, including graphical plots and alphanumeric displays, while other functionality within the MUST ecosystem supports reporting, parameter comparison, anomaly investigation, and monitoring. The capabilities demonstrate that telemetry visualization forms part of a broader workflow in which engineers actively explore and analyse spacecraft data (Solenix, 2026).

Within the context of this thesis, these characteristics establish the functional environment in which ORVAL operates. ORVAL similarly provides engineers with an interface through which telemetry parameters can be retrieved and visualized over time. However, the focus of the research extends beyond the visualization functionality itself. The software infrastructure supporting visualization must also remain available when individual backend components become unavailable. The reliability of the visualization workflow is therefore dependent not only on the functionality presented to the user, but also on the availability and behaviour of the infrastructure providing the underlying data.

2.3. Existing Infrastructure

The telemetry visualization environment considered in this thesis is based on the Mission Utility and Support Tools (MUST) ecosystem developed by the European Space Agency (ESA). MUST provides a software infrastructure for accessing, processing, storing, visualizing, and analysing telemetry and other mission data. The architecture follows a client-server model in which a central MUST server is responsible for importing and processing mission data and storing it in a repository, while client applications retrieve the processed data through standardized interfaces (European Space Agency (ESA), 2024; Baumgartner, Martínez-Heras, Donati, & Quintana, 2005).

2.3.1. MUST Architecture

The MUST architecture separates the processing and storage of mission data from the applications used to access and visualize that data. Data originating from spacecraft and other mission data providers is imported into the MUST infrastructure, where it can be processed and converted into a common representation. The resulting data is stored in the MUST repository, allowing client applications to retrieve the information without repeatedly accessing the original data providers (Baumgartner, Martínez-Heras, Donati, & Quintana, 2005).

The MUST server therefore acts as an intermediary between the sources of mission data and the applications used by engineers. According to ESA, the server handles data importation, processing, and storage, while MUST clients retrieve the resulting data through standardized interfaces over TCP/IP. This separation allows multiple client applications to access the same underlying repository and supports remote access to mission data (European Space Agency (ESA), 2024).

The architecture can consequently be represented at a high level as a sequence of data sources, the MUST server and repository, and one or more client applications. The client layer is not limited to visualization: the MUST ecosystem includes applications for data analysis, monitoring, anomaly investigation, reporting, and other forms of interaction with mission data (European Space Agency (ESA), 2024).

Figure 1 presents a simplified representation of the MUST architecture relevant to the context of this thesis.



Figure 1: Simplified MUST/WebMUST architecture relevant to the telemetry visualization workflow (adapted from Baumgartner et al., 2005; European Space Agency (ESA), 2024)

2.3.2. WebMUST as the Visualization Layer

WebMUST functions as a web-based client within the MUST ecosystem. It provides engineers with browser-based access to spacecraft data and supports interactive visualization and export of telemetry and other mission information. WebMUST can retrieve time-series data from the underlying MUST infrastructure and present it through graphical and other data representations (Solenix, 2026).

Unlike a standalone visualization application that directly manages the underlying telemetry data, WebMUST relies on the services and data repositories provided by the underlying infrastructure. The client therefore forms the user-facing layer of a larger software architecture in which data acquisition, processing, storage, and visualization are separated into distinct components (European Space Agency (ESA), 2024; Solenix, 2026).

This separation is relevant to the problem investigated in this thesis because the availability of the visualization interface depends on the availability of the software components that provide the required data. A failure in the infrastructure between the user and the telemetry repository can therefore affect the engineer's ability to access and visualize telemetry, even when the underlying telemetry source itself remains operational.

2.3.3. Data Flow

At a high level, telemetry data flows from the spacecraft or another mission data provider into the MUST infrastructure. The data is imported and processed by the server before being stored in the MUST repository. WebMUST subsequently retrieves the required data through the available interfaces and presents it to the user for visualization and analysis (European Space Agency (ESA), 2024; Baumgartner, Martínez-Heras, Donati, & Quintana, 2005).

This architecture allows the visualization client to remain focused on retrieving and presenting mission data while the underlying infrastructure handles data processing and storage. The approach also allows multiple client applications to access a common repository, supporting different forms of analysis and monitoring without requiring each client to implement its own telemetry processing pipeline (Baumgartner, Martínez-Heras, Donati, & Quintana, 2005).

For the purposes of this thesis, this data flow can be simplified to four main stages: data acquisition, backend processing and storage, data retrieval, and visualization. ORVAL operates primarily around the latter stages of this flow, where reliable access to the backend services becomes important for maintaining telemetry visualization.

2.3.4. Infrastructure Dependency

The architecture described above introduces dependencies between the visualization client and the services responsible for providing telemetry data. WebMUST itself does not represent the complete telemetry infrastructure; it relies on the underlying MUST services and repository to retrieve the information required by the user. Consequently, the availability of the visualization workflow depends on the availability of the supporting infrastructure (European Space Agency (ESA), 2024; Solenix, 2026).

This dependency is particularly relevant in an environment where engineers require continuous access to telemetry during testing. If an infrastructure component responsible for serving telemetry becomes unavailable, the visualization layer may no longer be able to provide the required information. The resulting problem is therefore not necessarily a failure of telemetry generation itself, but a failure in the software path through which engineers access that telemetry.

The original architecture consequently provides the functional foundation for telemetry visualization but does not, by itself, address the specific fault-tolerance mechanisms investigated in this thesis. The following sections examine the failure scenarios that can occur within this infrastructure and the limitations of the existing approach.

2.4. Existing Failure Scenarios

The architecture described in the previous section depends on several software and infrastructure components to provide engineers with access to telemetry data. Consequently, the availability of the visualization workflow is affected not only by the availability of the telemetry source itself, but also by the availability of the services, connections, and data repositories through which the telemetry is accessed. Ground systems used for spacecraft operations are themselves subject to failures, including failures in software, hardware, and communication links (European Space Agency (ESA), 2024; European Space Agency (ESA), 2007).

Within the context of this thesis, five failure scenarios are particularly relevant: backend failure, database failure, network connection failure, service restart, and backend recovery. These scenarios represent failures that can interrupt the software path between the user and the telemetry data and therefore provide the basis for analysing the limitations of the existing approach.

2.4.1. Backend Failure

A backend failure occurs when a software service responsible for providing telemetry data to the visualization layer becomes unavailable or stops responding. In the MUST architecture, client applications retrieve mission data from the server infrastructure through standardized interfaces. Consequently, an unavailable server-side component can prevent a client from retrieving the data required for visualization, even when the underlying telemetry data itself remains available (European Space Agency (ESA), 2024).

From the perspective of the visualization workflow, such a failure represents a loss of access to the telemetry service rather than necessarily a failure of the spacecraft or the telemetry source. The engineer may therefore remain unable to retrieve or visualize telemetry even though the system being monitored continues to produce data.

Ground-segment failures are recognized as an operational concern in spacecraft systems. ESA describes unexpected events affecting ground systems and discusses recovery procedures for failures involving ground equipment and software. Backup equipment can be used to restore normal operations following such failures (European Space Agency (ESA), 2007).

For the architecture investigated in this thesis, a backend failure therefore represents a relevant failure condition because the visualization layer depends on backend services to provide access to the underlying telemetry data.

2.4.2. Database Failure

A database failure occurs when the repository containing telemetry data becomes unavailable or cannot be accessed by the services responsible for retrieving the data. MUST uses a repository to store spacecraft parameter data and provides access to this information through its data-access interfaces (European Space Agency (ESA), 2024; Solenix, 2026).

If the database or repository cannot be reached, a visualization client may be unable to retrieve the requested telemetry even if the client application and backend service themselves remain operational. The failure can therefore propagate upwards through the software architecture, ultimately affecting the functionality available to the engineer.

Database failure is considered within this thesis as an infrastructure failure affecting telemetry availability. The thesis does not, however, investigate database replication, distributed database consistency, or database-level fault-tolerance mechanisms as independent research topics. The database is treated as an infrastructure dependency of the telemetry visualization framework.

This distinction is important because the objective of ORVAL is not to make the underlying database fault-tolerant by itself. Instead, ORVAL addresses the availability of the visualization service when one of its supporting backend paths becomes unavailable.

2.4.3. Network Connection Failure

A network connection failure occurs when communication between components of the telemetry visualization infrastructure is interrupted. The MUST architecture relies on network communication between clients and server-side infrastructure, with ESA describing standardized interfaces accessed through Transmission Control Protocol/Internet Protocol (TCP/IP) (European Space Agency (ESA), 2024).

A communication failure can therefore prevent a client from reaching a backend service even when both the client and backend remain operational. From the user's perspective, this may appear similar to a backend failure because the required telemetry data can no longer be retrieved.

Communication failures are also relevant within broader spacecraft ground segments. ESA describes ground systems as distributed environments containing communication networks and other interconnected infrastructure, and identifies data-link failures as possible sources of operational problems (European Space Agency (ESA), 2007).

For the purpose of this thesis, network failures are considered at the level required to evaluate the behaviour of the ORVAL software infrastructure. The thesis does not attempt to provide a comprehensive analysis of network reliability or communication infrastructure.

2.4.4. Service Restart

A service restart represents a temporary period during which a backend service is intentionally or unintentionally stopped and subsequently started again. During this interval, requests directed towards the service may fail even though the underlying service is expected to become available again.

Service interruptions can occur as part of maintenance, recovery from software failures, or other operational procedures. Ground-system operations therefore include activities in which software and hardware components may need to be recovered or replaced following failures (European Space Agency (ESA), 2007).

From the perspective of a telemetry visualization system, a service restart is particularly relevant because the system must distinguish between a temporary interruption and a persistent failure. A mechanism that reacts immediately to a single unsuccessful request could unnecessarily switch to redundant infrastructure during a short-lived interruption.

This consideration motivates the use of explicit failure-detection logic within ORVAL. Rather than treating every unsuccessful communication attempt as a definitive backend failure, the framework incorporates repeated health checks before initiating a failover decision.

2.4.5. Backend Recovery

Backend recovery occurs when a previously unavailable backend becomes operational again. Recovery introduces an additional architectural consideration because restoring the failed component does not automatically determine how existing users should be routed.

In a system containing primary and redundant backend components, users may have been redirected to the redundant component while the primary component was unavailable. If the primary component becomes available again without coordinated routing, users may subsequently connect to different backend instances depending on when they initiate or refresh their connections.

This creates a potential consistency problem at the system level. The issue is therefore not simply whether the failed backend can be restarted, but whether the overall visualization environment can return to a known and consistent state after recovery.

Recovery procedures generally aim to restore affected systems to a stable operational state and subsequently resume nominal operations where possible (European Space Agency (ESA), 2007). Within the context of this thesis, this introduces an additional consideration: restoring the primary backend is not sufficient by itself if users can remain distributed across different backend instances.

For ORVAL, backend recovery is therefore treated as a separate failure-handling phase rather than simply the inverse of failover. The framework must determine when the primary backend has recovered and coordinate the routing of users so that the system returns to its intended primary state.

2.4.6. Summary of Failure Scenarios

The failure scenarios described above demonstrate that loss of telemetry visualization can originate at different levels of the supporting software infrastructure. A backend failure can make a service unavailable, a database failure can prevent access to stored telemetry, and a network failure can interrupt communication between otherwise operational components. Temporary service interruptions introduce the additional challenge of distinguishing transient failures from persistent ones, while backend recovery introduces the need to coordinate the transition back to the primary system.

These scenarios demonstrate that providing a redundant backend alone is insufficient to guarantee reliable and consistent user access. A fault-tolerant visualization framework must also determine when a component should be considered unavailable, decide when failover should occur, redirect users to an operational component, and coordinate the return to the primary system following recovery. These requirements provide the basis for the limitations analysed in the following section.

2.5. Limitations of the Existing Approach

The MUST and WebMUST ecosystem provides extensive functionality for accessing, visualizing, monitoring, and analysing spacecraft telemetry data. MUST separates data processing and storage from the client applications used by engineers, while WebMUST provides a web-based interface for interactive visualization of spacecraft data (European Space Agency (ESA), 2024; Solenix, 2026). However, when continuous and consistent access to telemetry is considered as a requirement, several limitations become relevant to the existing approach.

2.5.1. Dependency on Backend Availability

The telemetry visualization workflow depends on the availability of the server-side infrastructure that provides access to the underlying telemetry data. Within MUST, the server is responsible for data importation, processing, and storage, while client applications retrieve data from the repository through standardized interfaces (European Space Agency (ESA), 2024).

This architecture creates a dependency between the visualization layer and the services responsible for providing the required data. If a required server-side component becomes unavailable, client applications may no longer be able to retrieve the telemetry required for visualization, even if the underlying telemetry source itself remains operational. The issue is therefore not necessarily a failure of telemetry generation, but a failure in the software path through which engineers access the generated data.

The existence of multiple MUST client applications does not remove this dependency. The clients provide different ways of interacting with mission data, but they still rely on the underlying MUST infrastructure or another compatible data source to retrieve the required information (European Space Agency (ESA), 2024; Solenix, 2026).

For the context of this thesis, this means that providing telemetry visualization functionality alone is insufficient to guarantee continuous access to telemetry. The supporting software infrastructure must also be capable of maintaining access when an individual backend component becomes unavailable.

2.5.2. Lack of an Explicit Autonomous Failover Mechanism

The documented MUST architecture describes a server and suite of client applications through which mission data can be accessed, visualized, analysed, and monitored (European Space Agency (ESA), 2024). WebMUST is described as a client application that uses MUST as its backend in the typical configuration, while also supporting other systems that provide time-series data (Solenix, 2026).

The available documentation for MUST and WebMUST does not describe an autonomous backend failover mechanism of the type investigated in this thesis. In particular, the documented architecture does not specify a mechanism that continuously monitors the availability of a primary visualization backend, determines when that backend has failed, and automatically redirects users to a redundant backend (European Space Agency (ESA), 2024; Solenix, 2026).

This distinction is important because redundancy and fault tolerance are not equivalent. The presence of multiple components or alternative applications does not by itself guarantee that a system can detect a component failure, select an operational alternative, and redirect users to that alternative. These mechanisms must be explicitly designed if continuous availability is a system requirement.

2.5.3. Lack of Infrastructure-Level Failure Detection

MUST provides monitoring and anomaly-investigation functionality for spacecraft and mission data. For example, its ecosystem includes monitoring capabilities and tools such as DrMUST and Novelty Detection for identifying and analysing anomalous telemetry behaviour (European Space Agency (ESA), 2024; Solenix, 2026).

However, monitoring the behaviour of spacecraft telemetry is different from monitoring the availability of the software infrastructure responsible for delivering that telemetry to a visualization client. A telemetry-monitoring mechanism may determine that a spacecraft parameter exhibits unexpected behaviour, but this does not necessarily determine whether the backend service providing access to that parameter is operational.

For the problem addressed in this thesis, an additional infrastructure-level health-monitoring mechanism is therefore required. Such a mechanism must determine whether the backend service remains available and provide the information necessary for a deterministic failure decision. This distinction forms an important part of the problem addressed by ORVAL.

2.5.4. Lack of Coordinated Recovery

Failure handling does not end when a failed backend becomes operational again. In a system containing primary and redundant backend components, users may have been redirected to a secondary component during the failure. Simply restoring the primary component therefore does not automatically determine how users should subsequently be routed.

This creates a distinction between component recovery and system recovery. Component recovery concerns whether the failed backend has become operational again, whereas system recovery concerns whether the overall visualization environment has returned to its intended and consistent state.

The importance of recovery procedures within ground infrastructure is recognized in spacecraft operations. ESA describes ground systems as being subject to failures and notes that recovery from failures in ground equipment and data links may require backup equipment and established recovery procedures (European Space Agency (ESA), 2007).

For the specific problem considered in this thesis, restoring the primary backend is therefore not sufficient if users can remain distributed across different backend instances. A coordinated mechanism is required to determine when the primary system has recovered and to establish a consistent routing state.

This limitation directly motivates the return-to-home mechanism implemented in ORVAL, which coordinates the transition from the redundant backend back to the primary backend after recovery.

2.5.5. Limited Focus on Infrastructure-Level Resilience

The primary purpose of MUST and WebMUST is to provide access to, visualization of, and interaction with mission data. MUST supports data storage, analysis, visualization, monitoring, anomaly investigation, reporting, and related mission-data functions, while WebMUST provides interactive web-based access to spacecraft data (European Space Agency (ESA), 2024; Solenix, 2026).

The limitation identified in this thesis is therefore not a lack of telemetry visualization functionality. The existing ecosystem already provides extensive functionality for engineers to access and analyse telemetry data. Rather, the limitation concerns the availability and resilience of the software infrastructure supporting that visualization workflow.

More specifically, the research addresses the situation in which an engineer is actively using the visualization system while one of the backend components becomes unavailable. In such a situation, maintaining the functionality of the visualization interface alone is insufficient; the underlying services must also support failure detection, failover, and coordinated recovery.

2.5.6. Summary of Limitations

The existing MUST/WebMUST environment provides the functional capabilities required for telemetry access, visualization, monitoring, and analysis. Its documented architecture nevertheless leaves several aspects relevant to the fault-tolerant visualization problem addressed by this thesis.

First, the visualization workflow depends on the availability of the underlying server-side infrastructure. Second, the documented architecture does not describe an autonomous mechanism for detecting failure of a primary backend and redirecting users to redundant infrastructure. Third, telemetry monitoring and anomaly detection do not directly address the availability of the software services providing access to telemetry. Finally, recovery of a failed backend does not inherently establish a coordinated routing state for users who were redirected during the failure.

These limitations demonstrate that providing redundant infrastructure alone is insufficient to guarantee fault-tolerant telemetry visualization. A fault-tolerant framework must, according to defined rules, detect relevant failures, redirect users to an operational backend, and coordinate the return to the primary system following recovery. These considerations form the basis for the operational requirements introduced in the following section.

2.6. Operational Requirements

The operational requirements for ORVAL were derived from the telemetry visualization workflow and software infrastructure used within the space testing environment at Redwire. These requirements were identified through analysis of the existing infrastructure, observation of the operational workflow, discussions with domain experts, and consideration of relevant technical and scientific literature. The purpose of this analysis is to establish what the visualization framework must be capable of providing within its intended operational context. The formal system requirements derived from these operational needs are defined in Chapter 4.

The requirements can broadly be divided into two complementary areas. First, ORVAL must provide the functionality required by engineers to access and analyse telemetry data. Second, the infrastructure supporting this functionality must remain available and maintain a consistent user state when individual backend components become unavailable. Both aspects are necessary because a fault-tolerant system that cannot provide the required visualization functionality would not satisfy the operational needs of its users.

2.6.1. Requirements Elicitation and Operational Context

The requirements considered in this thesis originate primarily from the existing telemetry visualization environment at Redwire. The analysis of the existing infrastructure established how telemetry data is currently accessed, processed, and visualized, while discussions and feedback from domain experts provided insight into the practical needs of engineers working with the system. These observations were complemented by the capabilities and limitations identified in the existing MUST and WebMUST infrastructure described in the preceding sections.

The distinction between operational requirements and implementation decisions is important. The operational environment establishes what the system needs to achieve, whereas the architecture and implementation determine how those objectives are achieved. For example, maintaining access to telemetry during a backend failure constitutes an operational requirement, whereas the three-attempt mechanism used by ORVAL to detect failure is an implementation decision described in Chapters 5 and 6.

The requirements identified in this section therefore describe the desired behaviour and capabilities of the framework without prematurely prescribing the technical solution. This allows alternative architectural and implementation approaches to be considered before the final ORVAL design is justified.

2.6.2. Telemetry Visualization Requirements

The first group of requirements concerns the core telemetry visualization functionality expected by engineers. The operational environment requires reliable access to relevant telemetry parameters and the presentation of the resulting data in a form that supports monitoring and analysis. This requirement reflects the role of telemetry visualization within spacecraft testing, where telemetry is used to monitor subsystem behaviour and investigate system events (ECSS, 2016).

ORVAL should support the selection and retrieval of individual telemetry parameters relevant to a particular test or analysis. Spacecraft systems can generate large numbers of parameters describing different aspects of system behaviour, making the ability to select relevant parameters an important part of the visualization workflow (Fortescue, Swinerd, & Stark, 2011).

The framework should also support multiple visualization representations. A single visualization type is not necessarily suitable for every analysis task. Depending on the information being investigated, engineers may need graphical time-series plots, numerical representations, or other forms of data presentation. Existing tools within the MUST ecosystem similarly provide different ways of accessing and representing mission data (European Space Agency (ESA), 2024; Solenix, 2026). ORVAL should therefore not be limited to a simple plot type where different representations are operationally useful.

A further requirement is real-time or near-real-time visualization. During spacecraft testing, engineers may need to observe telemetry while a test is being executed in order to monitor the behaviour of hardware and subsystems. Telemetry standards explicitly support monitoring during spacecraft integration, testing, and operations (ECSS, 2016). ORVAL should consequently provide sufficient timely access to newly available telemetry data to support this workflow.

ORVAL should additionally support historical telemetry visualization. Engineers may need to examine data generated before the current point in time in order to investigate previous events, compare system behaviour, or analyse anomalies. The existing MUST ecosystem provides access to stored spacecraft parameter data and supports the visualization and analysis of historical telemetry (European Space Agency (ESA), 2024; Solenix, 2026). Historical data access is therefore considered part of the required visualization functionality rather than an optional extension.

Finally, the framework should provide interactive user functionality. Engineers should be able to select parameters, adjust the relevant time interval, inspect measurements, and interact with the resulting visualizations according to the analysis being performed. This reflects the interactive nature of existing web-based telemetry visualization tools such as WebMUST, which provides users with browser-based access to spacecraft data and interactive plotting functionality (Solenix, 2026).

2.6.3. Availability and Redundancy Requirements

In addition to providing telemetry visualization functionality, ORVAL must address the availability limitations identified in the existing Redwire environment. The visualization workflow depends on backend services to retrieve and provide telemetry data. Consequently, the failure of an individual backend component should not result in unnecessary loss of access to the visualization system when an operational alternative is available.

The framework should therefore support backend redundancy. At least one alternative backend must be available to provide the required service when the primary backend becomes unavailable. The purpose of this redundancy is not to improve the performance of the visualization system, but to provide an alternative service path during component failure.

Redundancy alone, however, is insufficient to satisfy the operational requirement. The framework must also be able to monitor backend availability and detect failures autonomously. This is particularly important in the Redwire testing environment because relying on an engineer to manually identify a backend failure and redirect users would introduce additional delay and operational dependency.

The framework should consequently preserve access to telemetry visualization with minimal manual intervention when a supported backend failure occurs. The system must be capable of determining that the primary backend is no longer available and redirecting users to an operationally redundant backend according to predefined system behaviour.

2.6.4. Recovery and User Consistency Requirements

The availability requirement extends beyond the initial failure event. Once a failed backend becomes operational again, the framework must also determine how the recovered component should be reintegrated into the system.

ORVAL should therefore support automatic detection of backend recovery. The system must be able to determine when the primary backend has returned to an operational state rather than relying exclusively on manual intervention by an engineer.

Recovery must also be coordinated at the level of user routing. During a primary backend failure, users may have been redirected to a redundant backend. If the primary backend subsequently becomes available without coordinated routing, different users could potentially remain connected to different backend instances. This would result in an inconsistent system state and could complicate the management and monitoring of active users.

The operational environment therefore requires a coordinated mechanism for returning users to the primary backend after recovery. The objective is not merely to confirm that the primary backend is operational again, but to ensure that the visualization environment returns to a known and consistent state.

This requirement is particularly relevant to ORVAL because the system is intended to provide a shared visualization environment rather than an isolated connection for a single user. Maintaining consistent routing across users is consequently considered part of the availability and reliability requirements of the framework.

2.6.5. Operational Constraints

The identified requirements must be interpreted within the scope of the Redwire space testing environment. ORVAL is intended to improve the availability and resilience of the telemetry visualization infrastructure rather than replace the complete spacecraft ground segment.

The framework therefore does not attempt to make every component of the underlying infrastructure fault-tolerant. In particular, database replication, distributed database consistency, network infrastructure reliability, and spacecraft-level fault tolerance are outside the primary scope of the research. The database and network are instead treated as infrastructure dependencies whose failures may affect the availability of the visualization service.

Similarly, the framework is evaluated within a controlled testing environment rather than through deployment on an operational spacecraft. Consequently, the requirements defined in this thesis concern the behaviour of the software infrastructure under the controlled failure scenarios relevant to the Redwire testing environment. The extent to which the resulting architecture could be transferred to other operational environments is discussed later in this thesis.

2.6.6. Summary of Operational Requirements

The operational analysis identifies two complementary groups of requirements for ORVAL. The first concerns the functionality required to support telemetry visualization, including telemetry parameter retrieval, multiple visualization representations, real-time monitoring, historical data access, and interactive user operation. These requirements reflect the existing telemetry visualization workflow and the needs of engineers working with spacecraft test data (ECSS, 2016; European Space Agency (ESA), 2024; Solenix, 2026).

The second concerns the reliability and availability of the infrastructure supporting that visualization. ORVAL must provide backend redundancy, autonomous infrastructure-level failure detection, automatic failover, recovery detection, and coordinated user routing following backend recovery. These requirements directly address the limitations identified in the existing approach and the operational needs observed within the Redwire testing environment.

Together, these operational requirements define the capabilities that the proposed framework must provide without prescribing the specific technical mechanisms used to achieve them. Chapter 4 formalizes these requirements into functional and non-functional system requirements and establishes their relationship with the subsequent architecture, implementation, and validation activities.

2.7. Problem Definition

2.7.1. Definition of the Problem

The analysis presented in this chapter has established that telemetry visualization is an important part of spacecraft testing and that engineers depend on software infrastructure to access, process, and visualize telemetry data. The existing MUST and WebMUST ecosystem provides extensive functionality for these purposes, including telemetry access, visualization, monitoring, analysis, and remote interaction (European Space Agency (ESA), 2024; Solenix, 2026). However, the availability of these functions remains dependent on the availability of the underlying software infrastructure.

Within the Redwire testing environment, this dependency becomes particularly relevant when a backend component becomes unavailable. The analysis of the existing failure scenarios has shown that a backend failure or temporary service interruption can prevent users from accessing the telemetry visualization service even when an alternative backend is available. Furthermore, restoring a failed backend does not by itself guarantee that the overall system returns to a consistent routing state.

The problem is therefore not the absence of telemetry visualization functionality, nor the absence of redundant infrastructure as such. Rather, the problem concerns the lack of an integrated mechanism that can autonomously detect backend unavailability, determine when a failure requires a failover, redirect users to an operational backend, and coordinate the return to the primary backend after recovery.

This problem is particularly relevant in a multi-user testing environment. If different users are routed to different backend instances following a failure and subsequent recovery, the system may enter an inconsistent operational state. Maintaining availability therefore requires more than simply providing a secondary backend; failure detection, failover, recovery detection, and user routing must operate according to defined and predictable rules.

Building upon the context and limitations established in this chapter, the overarching problem identified in Chapter 1 can now be framed specifically for the operational environment of Redwire. Accordingly, the core problem addressed by this research is formally defined as follows:

Problem Statement

The existing telemetry visualization infrastructure does not provide an autonomous and deterministic mechanism for maintaining reliable and consistent access to telemetry visualization during backend failure and recovery

To address this problem, ORVAL investigates a fault-tolerant visualization framework that combines backend health monitoring, failure detection, deterministic failover, user redirection, recovery detection, and coordinated return to the primary backend. The specific requirements and design decisions resulting from this problem definition are developed in the subsequent chapters.

3. Theoretical Background & State of the Art

3.1. Telemetry Visualization

Telemetry visualization is the presentation of spacecraft telemetry in a form that allows engineers and operators to monitor system behaviour and analyse measurements over time. From a technical perspective, spacecraft telemetry can be considered a multivariate time-series dataset containing potentially large numbers of parameters with different sampling characteristics and temporal relationships (Német, 2023). Visualization systems must therefore provide mechanisms for efficiently retrieving relevant data while preserving the temporal and relational information required for interpretation.

From an architectural perspective, telemetry visualization can typically be separated from the services responsible for acquiring, processing, and storing telemetry. This separation allows visualization clients to focus on presenting and interacting with data while dedicated backend services handle data processing and storage (European Space Agency (ESA), 2024). It also creates requirements for efficient communication between the visualization layer and the underlying data services, particularly when both real-time and historical telemetry must be supported. Modern mission-control platforms such as NASA's Open Mission Control Technologies (Open MCT) similarly provide visualization capabilities for real-time mission telemetry and other mission data. (National Aeronautics and Space Administration (NASA), 2026).

The architecture of a telemetry visualization system therefore cannot be evaluated solely on the quality of its graphical interface. The availability and responsiveness of the services supplying the data directly affect whether engineers can continue to use the visualization system. A failure in these supporting services can consequently interrupt telemetry access even when the visualization layer itself remains operational. This dependency makes the reliability of the underlying infrastructure an important architectural consideration for telemetry visualization systems.

For this thesis, telemetry visualization is therefore considered as a layered system in which the visualization functionality depends on supporting data services. The following sections examine how availability, fault tolerance, redundancy, and failure-management mechanisms can be used to improve the resilience of such systems.

3.2. High Availability

High availability (HA) refers to the ability of a system or service to remain operational and accessible despite interruptions affecting individual components. NIST defines availability as the provision of timely and reliable access to information or services, while high availability is associated with failover mechanisms that maintain availability during device or component interruptions (National Institute of Standards and Technology (NIST), 2026a; National Institute of Standards and Technology (NIST), 2026b).

Availability is commonly expressed as the proportion of time during which a service remains operational over a defined period. This is often represented using percentages or so-called nines. For example, 99.9% availability permits approximately 8.8 hours of downtime per year, whereas 99.999% permits approximately 5.3 minutes (Oracle Corporation, 2026). These figures illustrate that availability is fundamentally related to the duration and frequency of service interruptions rather than simply the presence or absence of a failure.

For distributed and service-based systems, high availability is typically achieved by reducing single points of failure and providing mechanisms that allow service operation to continue when an individual component becomes unavailable. High-availability architectures may therefore incorporate redundant components, health monitoring, automated recovery, and failover mechanisms (Al-Sayed, 2022). The objective is to preserve the availability of the service from the perspective of its users rather than necessarily preventing every underlying component failure.

This distinction is particularly relevant to the telemetry visualization problem considered in this thesis. A backend component may fail without the overall visualization service necessarily becoming unavailable, provided that another operational backend can take over the required service. High availability therefore shifts the architectural objective from preventing component failures to limiting their impact on service continuity.

However, high availability does not by itself specify how a system should determine that a component has failed, when a failover should occur, or how users should be routed following recovery. These mechanisms represent separate architectural concerns that contribute to the implementation of a highly available system. The following sections therefore examine fault tolerance and redundancy as mechanisms through which high availability can be achieved.

3.3. Fault Tolerance

Fault tolerance refers to the ability of a system to continue providing its intended functionality despite the occurrence of faults within one or more of its components. A fault represents a condition that may cause a system component to deviate from its intended behaviour, while fault tolerance concerns the mechanisms that allow the overall system to continue operating despite such a condition (Avizienis, Laprie, Randell, & Landwehr, 2004).

Fault tolerance is closely related to, but distinct from, high availability. High availability describes the objective of minimizing service interruption, whereas fault tolerance describes the capability of tolerating faults without allowing them to cause unacceptable system-level consequences. A system can therefore be designed to be highly available by incorporating mechanisms that detect failures and restore service, while a fault-tolerant system specifically aims to continue delivering its required functionality in the presence of faults (Avizienis, Laprie, Randell, & Landwehr, 2004; National Institute of Standards and Technology (NIST), 2026).

Redundancy is one of the mechanisms commonly used to achieve fault tolerance. By providing multiple components capable of performing a required function, a system can potentially continue operating when one component fails. However, the presence of redundant components alone does not make a system fault-tolerant. The system must also be capable of recognizing relevant faults and taking appropriate action to prevent the failure of one component from interrupting the required service (Avizienis, Laprie, Randell, & Landwehr, 2004).

This distinction is particularly important for ORVAL. The presence of a primary and secondary backend provides redundancy, but does not by itself ensure that telemetry visualization remains available during a backend failure. ORVAL must additionally detect the relevant failure condition, determine when the failed backend should no longer be used, and redirect users to an operational backend. Fault tolerance in this context therefore results from the combination of redundant infrastructure and mechanisms for detecting and responding to backend failures.

Consequently, this thesis considers fault tolerance as a system-level property rather than as a synonym for redundancy or high availability. The following section examines redundancy as the architectural mechanism through which alternative backend resources can be provided.

3.4. Redundancy

Redundancy refers to the presence of additional components or resources that can provide the same or a sufficiently similar function as another component. It is a common means of improving system dependability because the failure of one component does not necessarily have to result in failure of the overall system (Avizienis, Laprie, Randell, & Landwehr, 2004).

Redundancy can be implemented in different configurations. In an active-active configuration, multiple instances are operational simultaneously and can serve requests at the same time. If one instance fails, the remaining instances can continue serving the workload. In an active-passive configuration, one instance normally serves the workload while another remains available as a standby and can take over when the active instance fails (Cisco Systems, 2007).

Active-passive redundancy therefore establishes a clear distinction between a primary and a secondary component. This configuration can simplify failure handling because the system has a defined normal state and a defined alternative state. However, the presence of a secondary component alone does not perform the failover. Additional mechanisms are required to detect failure, determine when the primary should no longer be used, and redirect the workload to the secondary component (Cisco Systems, 2007).

Redundancy can also be applied to data. Replication maintains copies of data across multiple storage locations or database instances, allowing an alternative copy to remain available when one instance becomes unavailable. Data replication and service redundancy address different layers of a system: replication protects the availability of data, while redundant services provide alternative paths through which applications can access that data (Microsoft Corporation, 2025).

For the problem considered in this thesis, the distinction between redundancy and fault-tolerance is particularly important. Redundant components provide the resources necessary for a system to continue operating after a failure, but they do not by themselves determine when a failure has occurred or how the system should transition between components. Redundancy is therefore an enabling mechanism for high availability and fault tolerance rather than a complete failure-handling mechanism.

This provides the basis for the following sections, which examine how system availability can be monitored and how failures can be detected and handled through mechanisms such as health checks and failover strategies.

3.5. Health Monitoring

Health monitoring refers to the continuous or periodic assessment of a system component to determine whether it is available and able to perform its intended function. In distributed systems, health monitoring is commonly implemented through health checks that periodically test a service and use the resulting status to determine whether the component should continue receiving requests (Kubernetes Authors, 2026).

A health check can assess different aspects of a component's condition. A basic check may verify whether a service responds to a network request, while more detailed checks can determine whether the application is operational and ready to serve requests. For example, Kubernetes distinguishes between liveness and readiness checks. A liveness check determines whether an application is still functioning, whereas a readiness check determines whether it is currently capable of accepting traffic (Kubernetes Authors, 2026).

This distinction between these states is important for fault-tolerant systems. A component may still be running while temporarily being unable to provide its required service. Consequently, checking only whether a process is running may not be sufficient to determine whether it should be considered an operational backend. Health monitoring should instead evaluate the condition that is relevant to the decision being made.

Health checks can be implemented through different mechanisms, including Hypertext Transfer Protocol (HTTP) requests, Transmission Control Protocol (TCP) connections, application-level responses, or dedicated health-check endpoints. The appropriate mechanism depends on the system and on what constitutes a meaningful indication of service availability (Kubernetes Authors, 2026).

For a fault-tolerant visualization framework, health monitoring provides the information required to determine whether a backend remains suitable for serving users. However, a health check only produces an observation about the current state of a component. The system still requires explicit rules for interpreting repeated observations, determining when a failure has occurred, and deciding when a failover should be initiated. These aspects are addressed in the following sections on heartbeat mechanisms and failure detection.

3.6. Heartbeat Mechanisms

Heartbeat mechanisms are a common approach for monitoring the availability of components in distributed systems. A heartbeat consists of a periodic signal sent between components to indicate that a monitored component is still responsive. When the expected heartbeat is no longer received within a defined time interval, the monitoring component can interpret the absence of the signal as an indication that the monitored component may no longer be available (Chandra & Toueg, 1996).

A basic heartbeat mechanism therefore consists of three elements: a monitored component, a monitoring component, and a predefined heartbeat interval or timeout. The monitored component periodically sends a heartbeat, while the monitoring component records the received signals. If no heartbeat is received within the expected interval, the monitoring component can mark the component as potentially failed. Heartbeat-based failure detection can consequently provide a relatively simple mechanism for translating component availability into an observable signal (Chandra & Toueg, 1996).

The choice of heartbeat interval and timeout represents a trade-off between detection speed and detection accuracy. A short interval allows failures to be detected quickly, but increases the sensitivity of the mechanism to temporary communication delays or transient service interruptions. A longer interval reduces the likelihood of interpreting a temporary delay as a failure, but also increases the time required to detect an actual failure (Prakash, 2022). The parameters used by a heartbeat-based failure detector therefore directly influence both its responsiveness and its accuracy.

This introduces the problem of false failure detection. In a distributed environment, the absence of a heartbeat does not necessarily mean that the monitored component has failed. The heartbeat may have been delayed or lost because of network conditions, or the monitored service may be temporarily unable to respond while remaining operational. Failure detectors must therefore operate under uncertainty and may incorrectly suspect a functioning component of having failed (Chandra & Toueg, 1996).

This distinction is commonly expressed through the concepts of completeness and accuracy. Completeness concerns whether failures are eventually detected, whereas accuracy concerns whether functioning components are incorrectly suspected of having failed. Chandra and Toueg (1996) demonstrate that failure detectors in distributed systems cannot necessarily provide perfect information about failures, making the balance between timely detection and incorrect suspicions an important design consideration.

For the telemetry visualization problem considered in this thesis, this trade-off is particularly relevant. An overly sensitive mechanism could initiate an unnecessary failover because of a temporary interruption, while an overly conservative mechanism could leave users connected to an unavailable backend for too long. Heartbeat mechanisms therefore provide the basis for observing backend availability, but additional decision logic is required to determine when repeated observations constitute a failure and should trigger a failover.

This distinction leads directly to the following sections, which examine how failure detection can be structured and how detected failures can be translated into deterministic failover decisions.

3.7. Failover Strategies

Failover refers to the process of transferring service from an unavailable component to an alternative component in order to maintain service availability. A failover strategy combines failure detection with rules for selecting and activating an alternative service path (HAProxy Technologies, 2026).

Failover can be manual or automatic. Manual failover requires an operator to identify the failure and initiate the transition to an alternative component. Automatic failover performs this transition according to predefined system rules without requiring direct operator intervention; automated failover can reduce service interruption and is therefore commonly used in high-availability architectures (HAProxy Technologies, 2026).

Failover can also be implemented using different redundancy configurations. In an active-active configuration, multiple instances can serve requests simultaneously, allowing traffic to be redirected to remaining instances when one becomes unavailable. In an active-passive configuration, a primary instance normally handles requests while a secondary instance remains available as a standby and takes over following failure (Amazon Web Services (AWS), 2026). The latter provides a clearly defined primary and alternative service path.

A failover mechanism additionally requires a method for routing requests or users to the selected backend. This can be implemented through mechanisms such as load balancers, DNS-based routing, or application-level routing. Load balancers, for example, can use health checks to determine which backend instances should receive traffic and remove unavailable instances from the active pool (HAProxy Technologies, 2026).

For systems with multiple users, the routing mechanism also influences the consistency of the resulting system state. In a multi-user environment, centralizing this routing decision can ensure that clients follow a common backend-selection rule rather than making independent failover decisions.

The distinction between routing individual requests and maintaining a common service path is therefore relevant when selecting a failover architecture. Infrastructure-level routing mechanisms generally determine where traffic should be directed based on the current availability and routing configuration of backend resources (HAProxy Technologies, 2026). An application-level mechanism can instead maintain an explicit active-path state that remains applicable across requests and clients until a defined state transition occurs. For the telemetry visualization environment considered in this thesis, the latter approach is relevant because failover is not intended merely to route individual requests away from an unavailable backend, but to establish a common active service path for the visualization environment. This distinction becomes particularly important during recovery, where the system must determine when the primary path should again become the active path.

Failover should therefore be understood as a combination of failure detection, decision-making, and service redirection rather than simply activating a backup component. The specific rules used to determine when a failure has occurred are examined in the following section.

3.8. State-based Failure Detection

Failure detection in distributed systems can be represented using explicit system states rather than treating each individual health check as an independent decision. A state-based approach allows the system to maintain a representation of the current condition of a monitored component and to change that state when observations satisfy predefined conditions. This approach separates the observation of a component from the decision about how that component should be treated by the system (Chandra & Toueg, 1996).

A simple state model can distinguish between states such as healthy, suspected, and failed. A healthy component is considered available based on the current health observations. When health checks begin to fail, the component can first enter a suspected state rather than being immediately classified as failed. If the failure condition persists according to the defined detection rules, the component can then transition to the failed state. This provides a way of distinguishing persistent failures from isolated or transient communication problems.

State transitions can also represent recovery. When a previously failed component begins responding again, it may first enter a recovering state rather than immediately being treated as fully operational. After sufficient successful health observations, the component can be transitioned back to the healthy state. A simplified lifecycle can therefore be represented as shown in Figure 2:

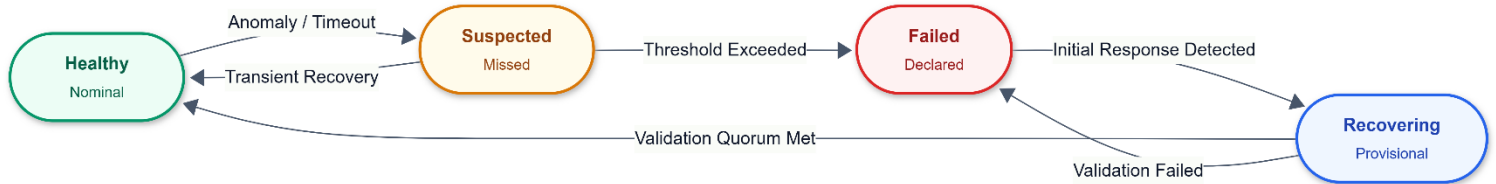


Figure 2: Finite state machine representation of the fault detection and component recovery lifecycle.

The exact states and transition conditions depend on the requirements of the system. The important characteristic is that transitions are governed by explicit conditions rather than by arbitrary individual observations. This makes the behaviour of the failure detector easier to reason about and provides a consistent basis for subsequent system actions.

State-based failure detection is particularly useful when transient failures are possible. A single missed heartbeat, for example, may indicate packet loss or temporary network delay rather than an actual backend failure. Introducing an intermediate suspected state allows the system to require additional evidence before declaring a component failed. Conversely, recovery can require multiple successful observations before the component is considered stable again.

For a fault-tolerant telemetry visualization system, this approach provides a structured basis for deciding when a backend should remain active, when it should be considered unavailable, and when it can safely be returned to service. The state model therefore connects health monitoring and heartbeat observations with the failover decision that follows from them.

State transitions alone, however, do not define which action should be taken in every situation. The conditions governing these transitions and the resulting actions must be deterministic if the system is expected to behave consistently under the same failure conditions. This relationship between state-based detection and deterministic decision-making is examined in the following section.

3.9. Deterministic Decision-making

Deterministic decision-making, as used in this thesis, refers to a system producing the same decision when presented with the same system state and input conditions. In dependable systems, deterministic behaviour is valuable because system actions become predictable and can be analysed against explicitly defined conditions, rather than relying on implicit or variable behaviour (Avizienis, Laprie, Randell, & Landwehr, 2004). Three distinct properties are relevant here and are used consistently throughout this thesis. Rule determinism means that the transition conditions are fixed and inspectable – for example, a failure is declared after three consecutive unsuccessful attempts, not after an adaptive or request-count-dependent threshold. Bounded timing means that, given the fixed rules, the detection and recovery intervals are bounded by configuration parameters (in ORVAL’s case, a 10-25 second detection window and a 0-30 second recovery window). Observed reproducibility means that repeated executions under the same conditions produce the same outcome within those bounds. Rule determinism and bounded timing are properties of the mechanism; observed reproducibility is an empirical property of the deployed system. The thesis uses ‘deterministic’ to mean all three jointly, and distinguishes them where the distinction matters for the validation results in Chapter 8.

In failure-handling systems, deterministic decision-making can be implemented through explicit state transition rules. For example, a system may define that a component moves from a healthy state to a suspected state after a defined number of unsuccessful health checks, and from suspected to failed after the failure condition persists. Similarly, recovery can require a defined number of successful observations before the component is considered operational again. The specific thresholds are system dependent; the important characteristic is that the conditions for changing state are explicitly defined.

Predefined rules also help prevent inconsistent responses to similar failure conditions. Without explicit decision criteria, a system may respond differently depending on timing, individual requests, or transient observations. In a multi-user environment, such differences can result in different users being routed according to different interpretations of the same backend condition. A deterministic decision mechanism can instead establish a common system-level response to a detected failure.

Determinism is therefore closely related to, but distinct from, failure detection. Health monitoring provides observations about the condition of a component, while state-based failure detection interprets those observations as changes in system state. Deterministic decision-making then defines the actions associated with those states, such as maintaining the current service path, initiating failover, or returning to the primary component.

For the telemetry visualization problem considered in this thesis, deterministic behaviour is particularly relevant because backend availability directly affects user access to telemetry. Failure detection and recovery must therefore lead to predictable routing decisions rather than independent or ambiguous responses. Explicit state transitions and predefined decision rules provide a basis for ensuring that the same operational conditions result in the same system behaviour.

The combination of health monitoring, state-based failure detection, and deterministic decision-making consequently provides a structured foundation for implementing predictable failover and recovery behaviour. These principles are used in the following section to examine existing technologies and architectures that apply related approaches to maintaining service availability.

3.10. Existing Approaches

Existing high-availability technologies implement several of the mechanisms discussed in the previous sections, including health monitoring, redundant service instances, failure detection, traffic routing, and automatic recovery. These approaches provide useful reference points for evaluating how the requirements identified in this thesis are addressed by existing technologies.

3.10.1. Load Balancing and Health-Based Routing

Load balancers provide one common approach to maintaining service availability across multiple backend instances. Rather than directing clients to a single fixed server, a load balancer can maintain a pool of backend resources and distribute incoming requests among available instances. Health checks can be used to remove unhealthy instances from this pool and restore them when they become operational again (HAProxy Technologies, 2026).

HAProxy, for example, supports periodic active health checks and configurable failure and recovery thresholds. A server can be removed from the load-balancing rotation after a defined number of consecutive failed checks and can subsequently be returned after a defined number of successful checks (HAProxy Technologies, 2026). This demonstrates how health monitoring, state transitions, and routing can be integrated within a single infrastructure component.

However, load balancing is primarily concerned with selecting an appropriate backend for individual requests or connections. Depending on the routing mechanism, different users or connections may be distributed across different backend instances. This differs from a system in which a centralized decision determines one common active backend for all users.

3.10.2. Active-Passive High-Availability Architectures

Active-passive architectures provide another established approach to service availability. One instance operates as the active component while another remains available as a standby. When the active component fails, the standby can be promoted or activated to provide the service.

HAProxy Enterprise, for example, supports active-standby configurations in which health checks and heartbeat mechanisms can trigger a transition between active and standby nodes (HAProxy Technologies, 2026). This approach is conceptually similar to the primary-secondary model described earlier, as the system maintains an explicitly defined normal service path and an alternative path.

Active-passive architectures therefore provide a useful basis for deterministic failover. However, the exact behaviour following recovery depends on the implementation. A system may automatically fail back to the original node, require manual intervention, or continue operating on the former standby node.

3.10.3. *Orchestration and Self-Healing*

Container orchestration platforms extend these concepts by managing the desired state of distributed applications. Kubernetes, for example, can automatically restart failed containers, replace failed application instances, and remove unavailable instances from the endpoint used for service traffic (Kubernetes Authors, 2026). This combines health monitoring, workload replacement, and traffic management into a broader self-healing architecture.

Kubernetes therefore demonstrates that availability can be maintained through continuous reconciliation between the observed system state and a predefined desired state. However, its primary objective is to manage distributed workloads and maintain application replicas rather than to provide application-specific deterministic routing behaviour between two predefined backend environments.

3.10.4. *Comparison of Existing Approaches*

The existing approaches demonstrate that the individual mechanisms required for fault-tolerant operation are well established. Load balancers provide health-based traffic routing, active-passive architectures provide standby service paths, and orchestration platforms provide automated recovery and maintenance of a desired system state (HAProxy Technologies, 2026; Kubernetes Authors, 2026).

Nevertheless, these approaches differ in the level at which they make decisions and in how recovery is handled. Load balancers typically make routing decisions for individual requests or connections, while orchestration systems generally manage the availability of application instances. Active-passive architectures provide a clearer primary-secondary relationship, but the exact failover and recovery behaviour depends on the implementation.

For the problem considered in this thesis, the relevant challenge is therefore not the absence of technologies capable of performing health checks or failover. Rather, the challenge is combining these mechanisms into behaviour that satisfies the specific operational requirements of the telemetry visualization environment. In particular, the system must be able to apply explicit failure-detection rules, redirect users to a redundant backend, maintain a common routing state, detect recovery, and coordinate the return to the primary backend.

These observations provide the basis for examining the limitations of existing approaches in the specific context of ORVAL. Modern service-mesh architectures represent the closest existing alternative to ORVAL's routing model, because they also combine health-based failover with programmable traffic management. Istio, for example, uses Envoy proxies to apply routing rules to traffic between services, while its outlier-detection mechanism can remove unhealthy service instances from load balancing and fail over to other available endpoints (Istio Authors, 2026). However, this routing operates at the level of individual requests or connections between services. A service mesh does not, by itself, enforce a single active backend across all clients of an application, nor does it coordinate convergence to the primary path after a recovered service becomes available again. In ORVAL's target environment, this distinction is operationally significant: clients are long-lived and stateful (interactive dashboards with continuously updating telemetry), and the requirement is not merely to route individual requests away from a failed backend, but to maintain one common active path for the whole visualization environment and to restore that path once recovery has been verified. This positions ORVAL's contribution more precisely than a general claim that existing approaches do not address application-level routing: the specific gap is coordinated, application-scoped convergence, not health-based failover as such.

Table 1: Mapping of existing approaches to operational conditions.

Approach	Operational conditions it can satisfy	Operational conditions it does not satisfy on its own
Active-passive HA	Component redundancy; availability to take over after failure	Single active path across long-lived clients; coordinated recovery across all users
Load balancer with health checks	Redundancy; ongoing health observation of backends	Single active path across all clients; rule-based failure decision; coordinated recovery
Kubernetes self-healing / orchestration	Availability of replicas; basic health signalling	Rule-based failure decision; application-scoped routing; functional recovery verification
Service mesh with outlier detection	Redundancy; health observation, per-request failover	Single active path across all clients; rule-based failure decision; coordinated return-to-home; integration without an interposed routing tier
Active-active replication	Continuous service from multiple instances; redundancy	Rule-based failover; coordinated recovery; single-active-path condition

3.11. Limitations in the Specific Context

The technologies discussed in the previous sections demonstrate that the individual mechanisms required for resilient service operation are well established. Load balancers can perform health checks and remove unavailable servers from the traffic rotation, while some systems can automatically redirect traffic to alternative servers or locations following a failure (HAProxy Technologies, 2026). Similarly, orchestration platforms such as Kubernetes use health probes to determine whether services are available to receive traffic and can automatically react to unhealthy components (Kubernetes Authors, 2026). These mechanisms demonstrate that failure detection, traffic redirection, and automated recovery are established concepts in resilient software architectures.

However, the surveyed approaches each leave specific aspects of the operational problem unaddressed. The four limitations below are stated in terms of the operational conditions that Chapter 2 established for the Redwire testing environment, rather than against the technologies in the abstract. First, the environment requires that all concurrent users of one visualization environment observe the same active backend. Per-request routing mechanisms – load balancers, service-mesh data planes – distribute requests across healthy backends according to their own policy and do not enforce a single application-scoped active path; different clients, or even different requests from the same client, may therefore be served by different backends at the same time. Second, the environment requires that recovery of the primary backend re-converges all users to the primary path in a coordinated way. Health-check-based failover returns a recovered backend to the healthy pool and lets new traffic reach it, but existing long-lived connections are not migrated; users therefore remain distributed between the primary and the previously active redundant backend after recovery. Third, the environment requires that a backend is treated as recovered only after it has been verified as functionally operational, not merely reachable. Health-check thresholds in the surveyed approaches typically distinguish ‘responsive’ from ‘not responsive’, but do not distinguish reachable from ‘verified functional’, so a service that passes a health check can be promoted back to active before its dependent state has recovered. Fourth, the environment requires the combination of application-scoped routing coordination, rule-based failure detection, autonomous failover, recovery verification, and coordinated return-to-home as a single behaviour. No surveyed approach provides this combination as a whole: each provides some subset, and the integration is left to the surrounding application. Each of the four limitations maps to a design decision in ORVAL; the operational conditions they reflect are formalised as system requirements in Chapter 4, and the design decisions that address them are described in Chapter 5. Together, they define the specific gap this thesis addresses.

HAProxy's configurable failure and recovery thresholds and Kubernetes' liveness/readiness distinction both demonstrate that health assessment and state transitions can be explicitly defined (HAProxy Technologies, 2026; Kubernetes Authors, 2026). Neither mechanism, however, specifies what the application should do with the resulting health state; the routing and recovery behaviour remains a property of the surrounding infrastructure and its configuration. This is the distinction that limitation three above captures.

The distinction becomes particularly relevant when considering the requirements identified for ORVAL. The target environment requires not only detection of backend failure, but also a deterministic sequence in which the primary backend is declared unavailable, users are redirected to a redundant backend, recovery of the primary backend is detected, and the routing state is subsequently restored to the primary backend. Furthermore, this routing behaviour must be coordinated across users rather than being treated solely as an independent decision for each incoming connection.

The existing approaches can therefore be summarized as follows:

Table 2: Comparative analysis of standard fault-tolerance approaches and their specific limitations within the ORVAL framework.

Approach	What it addresses	What is not directly addressed in the ORVAL context
Load balancing	Traffic routing, health-based server selection, and failover to available servers	Application-specific routing semantics and coordinated return-to-home behaviour
Active/passive architectures	Maintaining a redundant standby component and switching to it after failure	The complete application-level sequence of detection, redirection, recovery, and coordinated restoration
Health checks	Determining whether a component is responsive or ready to receive traffic	The complete decision logic governing what happens after a health state change
Heartbeat mechanisms	Periodic liveness monitoring and failure detection	User routing and coordinated recovery behaviour
Orchestration	Automated component recovery, health-based availability, and service management	The specific deterministic routing state required by the target environment

Service mesh	Programmable traffic routing, health-based failover, and outlier detection across service endpoints	Application-scoped coordinated convergence to a single active backend, and coordinated return-to-home after verified recovery
Failover Mechanisms	Switching service operation from a failed component to an alternative	Coordinated restoration of all clients to the primary service

These observations do not indicate that the existing technologies are incapable of implementing parts of the required behaviour. On the contrary, they provide the fundamental mechanisms on which resilient systems can be constructed. The limitation is that these mechanisms are generally designed as generic infrastructure capabilities, whereas the problem addressed by ORVAL requires their behaviour to be combined according to specific operational rules.

The key observation is therefore that the individual mechanisms required by ORVAL already exist, but their specific combination is not directly addressed by the existing approaches considered in this thesis. In particular, the target environment requires the integration of deterministic failure detection, centralized user routing, automatic failover, coordinated recovery detection, and return-to-home behaviour into a single telemetry visualization framework.

This distinction defines the research gap investigated in the remainder of this thesis. ORVAL does not seek to replace established high-availability, health-monitoring, or failover technologies. Instead, it investigates how these principles can be combined into a deterministic fault-tolerant visualization framework that satisfies the specific operational requirements identified in the Redwire testing environment.

3.12. Research Gap & Design Implications

The theoretical background and state-of-the-art analysis show that the individual mechanisms required for fault-tolerant service operation are well established. High-availability architectures provide principles for maintaining service availability, redundancy provides alternative service paths, and health-monitoring mechanisms allow operation to continue following a component failure. Heartbeats and health checks can further support the detection of failures, while state-based approaches provide a structured way of representing changing system conditions.

However, the analysis also demonstrates that these mechanisms do not directly define the complete behaviour required by the telemetry visualization environment considered in this thesis. In particular, the combination of infrastructure-level failure detection with centralized user routing, deterministic failover, coordinated recovery, and return-to-home behaviour represents a more specific operational requirement than the individual mechanisms discussed in the literature.

Stated explicitly, the gap is defined by the conjunction of six operational conditions that the surveyed approaches do not satisfy together: (i) a single active service path shared by all users of one application environment, rather than per-request or per-connection selection; (ii) failure detection governed by fixed, inspectable decision rules rather than by request-count, latency, or adaptive thresholds; (iii) failover that occurs only after a confirmed failure condition, rather than as an immediate response to individual failures; (iv) recovery verification that distinguishes functional availability from basic reachability before the primary path is restored; (v) coordinated return of all users to the primary path after recovery, rather than passive re-availability of the recovery node; and (vi) the integration of (i)-(v) within the application itself, without introducing a separate routing tier between clients and the redundant backends. Each condition is individually satisfiable by at least one existing mechanism. It is the requirement that they hold jointly, for the same application environment and under the same fixed rules, that constitutes the research gap addressed by this thesis. These six conditions are subsequently formalised as the availability, fault-tolerance, and consistency requirements in Chapter 4.

This leads to a research and design gap at the intersection of telemetry visualization and fault-tolerant infrastructure. The approaches examined in this chapter therefore do not, individually or jointly, satisfy the six-condition gap stated above.

The literature therefore provides several design implications for ORVAL. The framework should incorporate:

Table 3: Key architectural requirements and functional objectives of the fault-tolerant telemetry framework.

Core Capability	System Role & Rationale
Health Monitoring	Continuously assess the operational availability and responsiveness of backend components.
Explicit Failure States	Formally represent and differentiate between distinct stages of component degradation, failure, and recovery.
Deterministic Failure Detection	Ensure failover execution is governed by predefined, predictable state transitions rather than ad hoc reactions.
Redundancy	Provide an active standby service path to preserve continuous data flow when the primary service becomes unavailable.
Automatic Failover	Maintain telemetry without requiring manual operator intervention.

<i>Centralized Routing</i>	Enforce consistent backend endpoint selection across all distributed clients in the visualization environment.
<i>Recovery Detection</i>	Reliably detect when a degraded or failed primary backend has fully returned to a validated operational state.
<i>Coordinated Return-to-home</i>	Synchronize client reintegration to the primary service based on a unified system state, preventing fragmented routing.
<i>Predictable System Behaviour</i>	Ensure that identical fault and recovery conditions consistently produce reproducible, deterministic system outcomes.

Among existing approaches, service-mesh routing with health-based failover represents the closest architectural alternative; however, it operates primarily at per-request granularity and does not provide application-scoped route convergence across long-lived stateful clients following service recovery. Addressing this combined gap forms the primary design focus of this research.

Rather than replacing generic enterprise infrastructure technologies, these theoretical insights establish the foundational principles and architectural boundaries for the system. In the following chapter, these architectural insights are integrated with the operational needs identified in Chapter 2 to formalize the concrete functional, availability, and consistency requirements for ORVAL.

4. Requirements & System Analysis

This chapter translates the operational problem identified in Chapter 2 and the architectural insights established in Chapter 3 into a set of concrete system requirements for ORVAL. The purpose of this process is to establish a clear specification against which the proposed architecture, implementation, and subsequent validation can be evaluated.

The requirements were established through an iterative process combining the initial requirements provided by Redwire Space, consultation with domain experts, analysis of the existing operational environment, technical investigation, and the theoretical principles discussed in the previous chapter. The initial requirements served as a starting point rather than a fixed implementation specification. During the development process, requirements were evaluated against the identified problem and refined where architectural or technical analysis indicated that an alternative approach would better satisfy the intended objective.

This distinction is important because the requirements describe what the system must achieve, whereas the architectural and implementation decisions describe how these objectives are achieved. For example, the requirement to maintain telemetry access during a backend failure does not prescribe the use of a particular redundancy mechanism or software architecture; these decisions were instead developed through technical analysis and subsequently reviewed with the relevant domain experts.

The resulting requirements are categorized into functional, non-functional, availability and fault-tolerance, and consistency and multi-user requirements. They are subsequently prioritized and linked to the corresponding design, implementation, and validation activities through a requirement traceability approach. This provides a direct connection between the operational needs identified at the beginning of the project and the evidence used later in this thesis to evaluate ORVAL.

4.1. Requirements Elicitation and Refinement

The requirements for ORVAL were established through an iterative requirements elicitation and refinement process involving both domain-specific input and technical analysis. Rather than relying on a single requirements-gathering technique, multiple sources were combined to ensure that the resulting requirements reflected the operational needs of the Redwire environment while remaining technically feasible.

The initial input consisted of a set of approximately 40 requirements provided by Redwire at the beginning of the internship. These requirements described both operational and user-oriented expectations for the proposed visualization system. They included requirements related to the deployment environment, telemetry visualization functionality, user interaction, and system behaviour during satellite testing and operations. The initial set was subsequently analysed throughout the project, and requirements that were no longer applicable to the refined system concept were removed or adapted.

Domain expertise formed an important part of this process. Requirements and proposed solutions were discussed iteratively with the internship mentors, Șerban Anghel and Stijn Ilsen, as well as within the broader satellite engineering environment. These discussions provided the operational context necessary to determine whether proposed functionality and architectural decisions were appropriate for the intended users and testing environment. Discussions were not limited to formal meetings; technical proposals were also continuously reviewed during the development process before being incorporated into the system.

The existing architecture and operational environment were used as a second source of requirements. Redwire provided the existing telemetry databases and information about the surrounding systems, while the implementation of ORVAL itself was developed from this starting point. Analysis of the available infrastructure influenced the system boundary and led to the decision to use the existing telemetry databases as the data source rather than reproducing functionality for telemetry packet decoding within ORVAL. This illustrates the distinction between an initial project assumption and the technical solution ultimately selected to satisfy the underlying operational objective.

Technical analysis was subsequently used to investigate how the identified objectives could be achieved. This included considering failure scenarios, user behaviour, backend availability, recovery conditions, and the interaction between the different components of the proposed system. Several requirements emerged or were refined through this analysis. For example, consideration of backend recovery revealed that independently redirecting users based only on the current availability of individual backends could result in different users being connected to different backend environments. This led to the requirement for coordinated routing and deterministic return-to-home behaviour.

Theoretical knowledge and the literature discussed in Chapter 3 provided an additional basis for evaluating the resulting requirements. Concepts such as high availability, fault tolerance, redundancy, health monitoring, heartbeat-based failure detection, and deterministic state transitions were used to contextualize and validate the architectural principles underlying the proposed requirements. Literature was therefore primarily used as a theoretical foundation and validation mechanism rather than as a direct source from which individual ORVAL requirements were copied.

Finally, proposed requirements and architectural solutions were subject to Redwire's formal project review process, including a Critical Design Review (CDR). This provided an additional validation step in which the proposed system could be evaluated against the intended operational objectives before implementation decisions were finalized.

The overall requirements process can therefore be summarized as an iterative cycle:

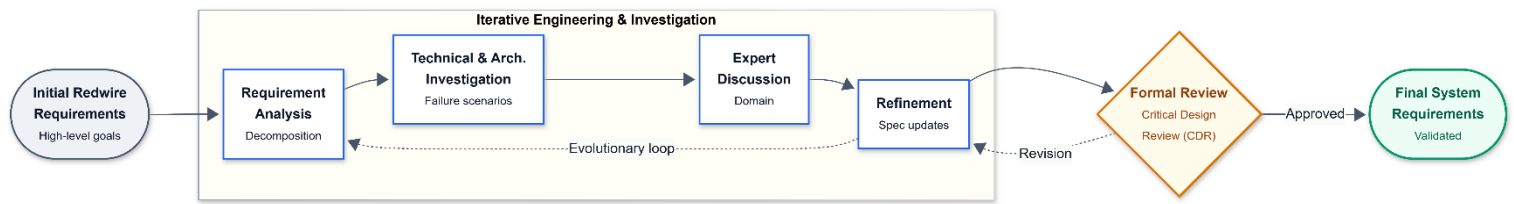


Figure 3: Iterative requirements engineering lifecycle and formal CDR validation gating.

This process ensured that the final requirements represented the agreed objectives of the project while allowing the technical solution to evolve as a deeper understanding of the problem and its failure scenarios was developed. The following sections define these final requirements and distinguish between the functionality ORVAL must provide and the quality and operational characteristics it must satisfy.

4.2. Functional Requirements

The functional requirements define the capabilities that ORVAL must provide to users and administrators. They were derived from the initial project requirements and subsequently refined through technical analysis, discussions with domain experts, and the iterative process described in Section 4.1. The requirements cover three principal functional domains: telemetry visualization, availability and failover management, and administrative control.

Table 4: Functional requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.

ID	Functional Requirement	Description
FR1	Telemetry data acquisition	ORVAL shall retrieve telemetry data from the configured telemetry database
FR2	Telemetry visualization	ORVAL shall allow users to visualize one or more telemetry parameters using graphical representations suitable for time-series data, supporting both historical and real-time data.
FR3	Visualization management	ORVAL shall allow users to configure and manipulate visualizations, including their parameters, time range, position, size, and ordering, as well as merge separate visualizations into a single graph via drag-and-drop interaction.

FR4	Dashboard management	ORVAL shall allow users to create, modify, save, and manage dashboards containing telemetry visualizations.
FR5	Dashboard deck management	ORVAL shall allow users to group multiple dashboards into a single dashboard deck.
FR6	Telemetry data export	ORVAL shall allow users to export selected telemetry data to Comma Separated Value (CSV) format.
FR7	Service health detection	ORVAL shall continuously monitor the availability of configured backend and database services.
FR8	Service failure detection	ORVAL shall detect service unavailability based on configurable health-check parameters and a configurable failure threshold.
FR9	Automatic failover	ORVAL shall automatically switch from an unavailable service to its configured redundant service.
FR10	Consistent service routing	ORVAL shall ensure that all active and newly connected users are routed through the same active backend and database path.
FR11	Service recovery detection	ORVAL shall continuously monitor unavailable primary services and detect when they have successfully recovered.
FR12	Automatic return-to-home	ORVAL shall automatically restore the primary service path for all active and newly connected users after successful recovery.
FR13	Administrative health and routing control	ORVAL shall allow administrators to monitor service health and manually override automatic routing decisions.
FR14	User management	ORVAL shall allow administrators to create, modify, and remove user accounts.

FR15	Administrative content management	ORVAL shall allow administrators to manage dashboards and dashboard decks globally.
FR16	Administrative system and log management	ORVAL shall allow administrators to perform system-level control operations and access application log information through the ORVAL interface.

Administrative user and session management (FR14, FR16) includes the capability to invalidate active operator sessions globally. In an operational testing control room, where engineering shifts rotate and shared terminal access must be cleared rapidly following test completion or security boundary shifts, session invalidation allows administrators to terminate all active credentials without requiring manual reconfiguration of individual client workstations.

As summarized in Table 4, the functional requirements cover three complementary aspects of ORVAL: telemetry visualization, availability management, and administration. The visualization requirements (FR1-FR6) define the functionality through which users retrieve, visualize, organize, and export telemetry data. Features such as graph positioning, resizing, linking, automatic layout, and locking are considered part of visualization management rather than separate functional requirements.

The availability requirements (FR7-FR12) define the fault-handling behaviour of the framework. An important requirement is that a service failure results in a system-wide routing decision rather than an independent decision for each user. When the active service becomes unavailable, all active and subsequently connecting users must therefore follow the same redundant service path. The same principle applies during recovery, where users are collectively returned to the primary service once it has been confirmed to be operational. The specific health-check interval and failure threshold are intentionally not defined at requirement level, as these constitute configurable design decisions that are addressed in Chapter 5.

The administrative requirements (FR13-FR16) provide administrators with the control required to monitor and manage the system during normal operation, maintenance, testing, and exceptional situations. This includes monitoring service health and routing, overriding automatic routing, managing users and shared content, and accessing application logs.

The requirements in Table 4 therefore establish the functional scope against which the subsequent architecture and implementation can be evaluated. The following section defines the non-functional requirements that specify the quality attributes and operational constraints that ORVAL must satisfy.

4.3. Non-Functional Requirements

While the functional requirements define what ORVAL must be able to do, the non-functional requirements define qualities and operational constraints under which these functions must operate. They were derived from the operational context, the existing telemetry visualization workflow, discussions with domain experts, and the technical analysis performed during the project.

The resulting non-functional requirements are summarized in Table 5.

Table 5: Non-functional requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.

ID	Category	Non-functional Requirement
NFR1	Performance	ORVAL shall retrieve and render requested historical telemetry datasets within a bounded latency acceptable for operational monitoring, targeting retrieval and display within 10 seconds for standard operational queries under nominal-segment network conditions.
NFR2	Performance	ORVAL shall support configurable periodic updates for live telemetry visualization, with a default refresh interval of five seconds.
NFR3	Maintainability	ORVAL shall be sufficiently documented to allow future developers and engineers to understand, operate, configure, and modify the system without requiring the original developer.
NFR4	Maintainability	ORVAL shall support reproducible deployment, restart, and redeployment through a documented containerized deployment approach.
NFR5	Scalability	ORVAL shall support multiple concurrent users within the intended operational context without requiring a separate application instance for each user.
NFR6	Observability	ORVAL shall record relevant user, administrative, and system events through application logging to support operational monitoring and troubleshooting.

NFR7	Observability	ORVAL shall provide administrators with sufficient information to assess the current operational state of the application.
NFR8	Operational independence	ORVAL shall operate without requiring an internet connection or external Internet-dependent services.

The requirements in Table 5 complement the functional requirements from Section 4.2. In particular, the performance requirements establish that the framework should not introduce additional latency compared with the existing visualization workflow, while still providing a configurable mechanism for live telemetry updates. The five-second refresh interval represents the default operational configuration rather than an immutable system limitation.

Maintainability is addressed both through documentation and through the deployment approach. The intention is that the system can be understood and maintained after the development period, including changes to its configuration and components. Containerization contributes to this by providing a reproducible deployment environment.

The default five-second live telemetry refresh interval (NFR2) was selected to match the operational pace of ground-segment satellite integration and testing at Redwire. While subsystem sensors generate measurements at frequencies up to several Hertz, updating visualization charts at a five-second cadence provides operators with near-real-time visibility into thermal, electrical, and mechanical trends while keeping database connection pool connection and browser-tier rendering overhead bounded during multi-dashboard monitoring.

The scalability requirement is intentionally expressed in terms of the intended operational context rather than a fixed maximum number of users. The project required support for multiple concurrent users, but did not establish a formal upper capacity limit. Consequently, the requirement is that normal multi-user operation remains possible without architectural changes for each additional user.

Observability focuses on providing administrators with meaningful information about system behaviour rather than recording every individual operation. Relevant user actions, administrative operations, service failures, failovers, and recovery events are therefore logged, while highly repetitive low-value operations may be omitted. This provides a balance between diagnostic value and excessive log generation.

Operational independence is addressed by requiring ORVAL to function without Internet connectivity or external Internet-dependent services. This reflects the intended operational environment in which the framework must remain usable independently of external network services.

Together, the requirements in Sections 4.2 and 4.3 define both the functional scope and the quality constraints of ORVAL. The following sections refine the requirements specifically related to availability, fault tolerance, and consistent multi-user behaviour during service failure.

4.4. Availability and Fault-Tolerance Requirements

The availability and fault-tolerance requirements define how ORVAL shall maintain access to telemetry data when individual services become unavailable. While the functional requirements in Section 4.2 established the required capabilities for health monitoring, failover, and service recovery, this section specifies the operational conditions and guarantees associated with these capabilities.

Table 6: Availability and fault-tolerant requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.

ID	Category	Availability and Fault-Tolerant Requirement
AFR1	Availability	ORVAL shall maintain telemetry access when an individual backend or database service becomes unavailable, provided that the configured redundant service remains operational.
AFR2	Failure detection	ORVAL shall distinguish temporary or isolated health-check failures from confirmed service unavailability using configurable failure-detection parameters.
AFR3	Failover	ORVAL shall initiate failover only after the configured failure condition has been reached, preventing unnecessary switching caused by transient failures.
AFR4	Failover	ORVAL shall transfer service operation to the configured redundant service without requiring manual intervention when operating in automatic routing mode.
AFR5	Recovery	ORVAL shall verify application-level responsiveness of a recovered primary backend service via structured health probing before initiating automatic return-to-home routing.
AFR6	Recovery	ORVAL shall continuously monitor an unavailable primary service to determine when recovery has occurred.

AFR7	Fault isolation	ORVAL shall prevent the failure of an individual backend or database service from inherently causing failure of the complete telemetry visualization system when a redundant service remains available.
-------------	-----------------	---

The requirements in Table 6 refine the availability and fault-tolerance behaviour introduced by the functional requirements. The central objective is that the failure of an individual service does not unnecessarily interrupt access to telemetry data. This is achieved by using a redundant service as an alternative when the active service becomes unavailable.

Failure detection shall account for the possibility of temporary or transient failures. A single unsuccessful health check shall therefore not necessarily result in an immediate failover. Instead, failure shall be determined using configurable health-check parameters and a failure threshold. The specific values used by ORVAL are implementation and configuration decisions and are addressed in Chapter 5.

Once a service failure has been confirmed, failover shall occur automatically when the system is operating in automatic routing mode. The redundant service must therefore be capable of taking over the required functionality without requiring an administrator to manually restore telemetry access. Administrative control over routing remains available through the functionality defined in FR13.

Recovery is treated as a separate condition from failure detection. A previously unavailable primary service shall not immediately be considered suitable for normal operation solely because it becomes reachable again. ORVAL shall verify that the service is functionally operational before restoring it as the active service path. This prevents the system from returning to a service that is reachable but not yet capable of fulfilling its required function.

The requirements also establish the intended fault boundary of ORVAL. ORVAL is responsible for detecting and handling failures of the services on which its visualization functionality depends, but it does not guarantee the internal reliability or generation of telemetry data within the underlying telemetry databases. If both the active and redundant services are unavailable, continuous telemetry access cannot be guaranteed by the framework.

Together, these requirements define the availability and fault-tolerance behaviour that the architecture and implementation must realize. The next section addresses the related requirement for consistent behaviour across multiple users during failover and recovery.

4.5. Consistency and Multi-User Requirements

The consistency and multi-user requirements define how ORVAL shall maintain coherent routing behaviour across multiple users during normal operation, backend failures, and service recovery. While the availability and fault-tolerance requirements in Section 4.4 establish that telemetry access shall be maintained during backend failures, this section focuses on ensuring that backend and database transitions remain consistent and transparent for all connected users.

Table 7: Consistency and multi-user requirements specification for the ORVAL telemetry visualization and fault-tolerant routing framework.

ID	Category	Consistency and Multi-User Requirement
CMR1	Multi-user consistency	ORVAL shall ensure deterministic route convergence across independent client sessions during normal operation, failover, and recovery through shared deterministic state transition rules.
CMR2	Failover consistency	ORVAL shall route users to the configured redundant service when the active service becomes unavailable and automatic routing is enabled.
CMR3	Recovery consistency	ORVAL shall ensure that newly initialized client sessions and existing active client sessions evaluate the same deterministic health criteria, converging on the primary service path following verified recovery.
CMR4	Coordinated recovery	ORVAL shall restore all connected users to the primary service once the primary service has been verified as functionally operational.
CMR5	Visualization continuity	ORVAL shall maintain visualization workspace continuity during service transitions, ensuring dashboard layouts and historical telemetry displays persist without disruption, and live telemetry polling automatically resumes upon redundant route activation.
CMR6	Routing state consistency	ORVAL shall maintain a consistent routing state across user sessions when transitioning between services.

The requirements in Table 7 extend the fault-tolerance behaviour defined in Section 4.4 by considering the effect of backend or database transitions on multiple users. A service transition shall not result in different users being unintentionally associated with different active services. Maintaining a consistent routing state is particularly important because the telemetry visualization framework may be used simultaneously by multiple users whose activities depend on the same underlying telemetry environment.

During a confirmed service failure, all users shall be transferred to the configured redundant service when automatic routing is enabled. The failover process shall therefore be applied consistently rather than independently for each user. This ensures that the backend or database selected by ORVAL represents a system-wide routing decision and prevents users from operating against different service instances following the same failure event.

Consistency shall also be maintained during recovery. When the primary service becomes available again, ORVAL shall not simply route newly connected users to the primary service while existing users remain connected to the redundant service. Such a situation would result in users being distributed across different service instances and could lead to inconsistent system behaviour, particularly for shared logging and operational monitoring. Instead, recovery shall be coordinated so that the primary backend becomes the active service path for all users once its functional availability has been established.

Service transitions shall furthermore remain transparent to the user where possible. The transition between services shall not unnecessarily interrupt the visualization of telemetry data. When the target backend or database is operational, ORVAL shall maintain the continuity of the displayed telemetry information such that users can continue monitoring telemetry without requiring manual reconnection or service selection; the implementation and observed transition behaviour are addressed further in the subsequent chapters.

These requirements establish consistency as a system-wide property rather than a property of an individual user session. ORVAL shall therefore coordinate routing decisions across users while maintaining continuous access to the telemetry visualization functionality. This ensures that failover and recovery preserve both the availability of the system and the coherence of the environment in which multiple users operate.

Together, these requirements define how ORVAL shall handle multi-user behaviour during service transitions. By coordinating routing decisions across users and maintaining continuous telemetry visualization, ORVAL can preserve both the availability and consistency of the system during failover and recovery.

4.6. Requirement Prioritization

The requirements identified in Sections 4.2 to 4.5 differ in their importance to the core purpose of ORVAL. To distinguish essential system capabilities from supporting and desirable functionality, the requirements were prioritized using the MoSCoW prioritization method. This method classifies requirements into four categories: Must have, Should have, Could have, and Won't have. Must-have requirements are essential for the system to fulfil its primary purpose and are therefore mandatory for a functional implementation. Should-have requirements are important and provide significant value, but the system could fulfil its core purpose without them. Could-have requirements are desirable additions that provide additional functionality or convenience but have a lower priority. Won't-have requirements are explicitly outside the scope of the current implementation.

The classification was established through an interactive decision-making and peer-review process involving technical analysis and formal operational reviews with Redwire domain mentors (Şerban Anghel and Stijn Ilsen). Initial requirement weights were proposed based on system criticality and technical dependency trees. These classifications were subsequently evaluated and finalized during project milestone reviews, including the formal Critical Design Review (CDR).

Requirements directly governing real-time telemetry access, backend fault containment, deterministic failover, and multi-user routing consistency were formally designated as Must-have, as their absence would directly compromise ground-segment visibility during critical operational phases. In contrast, secondary operational conveniences – such as dashboard deck grouping (FR5) and direct CSV exporting (FR6) – were agreed upon as Could-have extensions, ensuring engineering efforts remained focused on architectural dependability without blocking baseline qualification.

Table 8: MoSCoW prioritization of ORVAL functional, non-functional, availability, and consistency requirements.

ID	Priority
FR1	Must
FR2	Must
FR3	Should
FR4	Should
FR5	Could
FR6	Could

FR7	Must
FR8	Must
FR9	Must
FR10	Must
FR11	Must
FR12	Must
FR13	Should
FR14	Should
FR15	Should
FR16	Should
NFR1	Must
NFR2	Must
NFR3	Should
NFR4	Should
NFR5	Must
NFR6	Should
NFR7	Should
NFR8	Must
AFR1	Must
AFR2	Must
AFR3	Must

AFR4	Must
AFR5	Must
AFR6	Must
AFR7	Must
CMR1	Must
CMR2	Must
CMR3	Must
CMR4	Must
CMR5	Must
CMR6	Must

The Must-have requirements define the minimum capabilities required for ORVAL to fulfil its primary purpose. This includes acquiring and visualizing telemetry data, continuously monitoring service health, detecting confirmed service failures, and automatically switching to a redundant service when required. The requirements concerning service recovery, fault isolation, and consistent multi-user routing are equally critical because the fault-tolerant behaviour of ORVAL depends not only on detecting a failure, but also on maintaining a coherent and operational system during failover and subsequent recovery. The core performance, multi-user, and operational independence requirements are also classified as Must-have requirements because they directly reflect the conditions under which ORVAL is intended to operate.

The Should-have requirements provide important supporting functionality but are not essential to the core fault-tolerant mechanism. These requirements primarily concern visualization management, dashboard management, administrative control, user management, maintainability, deployment, and observability. Although these capabilities significantly improve the usability, operability, and long-term maintainability of ORVAL, the system could still provide its core telemetry visualization and failover functionality without all of them.

The Could-have requirements represent additional functionality that improves the flexibility or convenience of the framework but is not essential to its primary purpose. Within the current requirement set, dashboard deck management and telemetry data export fall into this category. Their lower priority allows development effort to remain focused on the core telemetry visualization and fault-tolerance.

No requirements were classified as Won't-have. This does not mean that category is irrelevant; rather, none of the requirements currently defined in Sections 4.2 to 4.5 were explicitly identified as being outside the scope of the project. Functionality that was not considered part of the intended ORVAL system was excluded during requirements analysis rather than being included in the requirement set and subsequently assigned a Won't-have priority.

The prioritization therefore establishes a clear distinction between the essential behaviour of ORVAL and its supporting functionality. The Must-have requirements form the minimum set against which the implemented system must be evaluated, while the Should-have and Could-have requirements represent additional capabilities that contribute to the overall quality and usability of the framework. This prioritization provides a basis for interpreting implementation decisions and evaluating the extent to which the final system fulfils the defined requirements.

4.7. Requirement Traceability

Requirement traceability establishes a connection between the requirements defined in this chapter and the subsequent design, implementation, and validation activities. Its purpose is to ensure that the requirements are not treated as isolated specifications, but remain connected to the technical decisions and system components developed throughout the project. This provides a basis for demonstrating that the architecture and implementation of ORVAL directly address the needs identified during the requirements analysis.

The traceability approach used in this project follows the progression from requirement to design, implementation, and validation. Requirements are first translated into architectural goals and design decisions in Chapter 5. These designs are subsequently realized during the implementation described in Chapter 6 and evaluated through the validation activities described in Chapter 8. This creates a direct relationship between what ORVAL is required to achieve and how the resulting system is designed and evaluated.

Because several requirements concern the same underlying system mechanisms, traceability is presented at the level of related requirement groups rather than repeating every requirement individually. The individual requirement identifiers remain explicitly referenced so that each requirement can still be traced to the relevant parts of the subsequent chapters.

Table 9: Traceability between ORVAL requirements and subsequent system development and validation activities.

Requirement group	Related Requirements	Primary Design Sections	Validation
Telemetry acquisition and visualization	FR1-FR6	5.2-5.5	8.2, 8.8
Service health and failure detection	FR7-FR8, AFR2-AFR3	5.6-5.7	8.3-8.5, 8.8

<i>Automatic failover</i>	FR9, AFR1, AFR4, AFR7	5.8	8.3, 8.6, 8.8
<i>Service recovery and return-to-home</i>	FR11-FR12, AFR5-AFR6	5.9	8.4, 8.6, 8.8
<i>Multi-user routing and consistency</i>	FR10, CMR1-CMR6	5.10-5.11	8.5, 8.6, 8.8
<i>Administrative functionality</i>	FR13-FR16	5.2-5.5, 5.12	8.2, 8.8
<i>Performance and live updates</i>	NFR1-NFR2	5.1, 5.2-5.5	8.2, 8.7-8.8
<i>Maintainability and deployment</i>	NFR3-NFR4	5.1, 5.12	8.2, 8.8
<i>Scalability and multi-user operation</i>	NFR5	5.2, 5.10-5.11	8.2, 8.5, 8.8
<i>Observability</i>	NFR6-NFR7	5.2, 5.6, 5.12	8.2, 8.5, 8.8
<i>Operational independence</i>	NFR8	5.1, 5.12	8.1, 8.8

The first group of requirements concerns the core telemetry visualization functionality of ORVAL. FR1-FR6 are primarily addressed through the overall architecture and the frontend, backend, and database architecture described in Sections 5.2 to 5.5. These requirements establish the basic functionality that allows users to retrieve, visualize, manipulate, organize, and export telemetry data. Their behaviour is evaluated during normal operation in Section 8.2 and consolidated in the requirement validation presented in Section 8.8.

The fault-tolerance requirements have a more direct relationship with the architecture described in Sections 5.6 to 5.9. FR7 and FR8, together with AFR2 and AFR3, are addressed through the health-monitoring and failure-detection mechanisms. These mechanisms provide the basis for detecting service failures without unnecessarily triggering failover due to transient health-check failures. Their behaviour is examined through the primary failure and three-attempt validation scenarios in Sections 8.3 to 8.5. The resulting requirement outcomes are summarized in Section 8.8.

FR9 and the associated availability requirements are realized through the autonomous failover mechanism described in Section 5.8 and implemented in Chapter 6. Their behaviour is validated through the primary failure, autonomous failover, and repeated failure scenarios in Sections 8.3, 8.6 and 8.8. This ensures that failover is evaluated not only as an individual mechanism, but also under repeated controlled failure conditions.

The recovery-related requirements are traced to the return-to-home mechanism in Section 5.9. In particular, FR11, FR12, AFR5, and AFR6 require ORVAL to detect when a previously unavailable primary service has recovered and to restore the primary service path only after its functional availability has been established. This behaviour is validated through the recovery and return-to-home scenarios in Section 8.4, with repeated behaviour considered in Section 8.6 and summarized in Section 8.8. This traceability is important because recovery is not treated as merely the inverse of failover, but as a separate system behaviour with its own architectural mechanism.

The multi-user requirements are primarily addressed by Sections 5.10 and 5.11; FR10 and CMR1 to CMR6 require routing decisions to remain consistent across users during normal operation, failover, and recovery. The corresponding architectural design therefore considers routing and state at system level rather than treating service selection as an independent decision for each user. This directly addresses the risk of users becoming unintentionally distributed across different service instances following a failure and subsequent recovery. The resulting behaviour is evaluated through the multiple-user consistency scenarios in Section 8.5 and the repeated failure scenarios in Section 8.6, and consolidated in Section 8.8.

The non-functional requirements are traced to multiple architectural concerns because quality attributes generally influence the system as a whole rather than a single component. Performance and live-update requirements influence the architecture and implementation of the telemetry visualization and update mechanisms and are evaluated through the normal-operation and performance tests in Sections 8.2, 8.7, and 8.8. Maintainability, deployment, scalability, and observability requirements are addressed through the corresponding architectural and implementation decisions and are considered in the relevant validation scenarios. Operational independence is considered as part of the validation environment and system setup described in Section 8.1.

The traceability established in Table 9 provides a continuous link between the requirements, the resulting architecture, and their eventual validation. During implementation, the architectural decisions are translated into concrete system components and mechanisms. During validation, the resulting behaviour can then be assessed against the original requirements rather than solely against the implementation itself. This makes it possible to determine not only whether a particular feature was implemented, but whether the implementation satisfies the requirement for which the feature was introduced.

The detailed requirement-level validation is consolidated in Section 8.8, where the relevant requirements are mapped to their corresponding tests, expected behaviour, observed results, and pass/fail outcomes. Consequently, the traceability established in this section provides the structural link, while Section 8.8 provides the evidence that the requirements were actually satisfied.

Together with the prioritization in Section 4.6, the traceability analysis completes the requirements analysis for ORVAL. The requirements define what the system must achieve, the prioritization establishes which capabilities are essential, and the traceability relationships establish where these requirements are addressed throughout the remainder of the project. Chapter 5 subsequently translates these requirements into the architectural and design decisions used to construct the ORVAL framework.

5. ORVAL Architecture & Design

This chapter describes the architecture and design of ORVAL and addresses the second research question: How can the identified requirements be translated into an architecture that supports deterministic and fault-tolerant telemetry visualization?

Based on the requirements and system analysis presented in Chapter 4, this chapter translates the identified functional and non-functional needs into concrete architectural decisions. It first establishes the overarching design goals, after which the overall system architecture and the individual frontend, backend, and database components are described. The chapter then focuses on the mechanisms that enable ORVAL's fault-tolerant behaviour, including health monitoring, failure detection, autonomous failover and return-to-home. Finally, the user-routing and state-management mechanisms are discussed, followed by the main design decisions and trade-offs made during the development of the framework.

5.1. Design Goals

The requirements defined in Chapter 4 describe the functional and operational behaviour that ORVAL must provide. To translate these requirements into an architectural solution, a set of overarching design goals was established. Rather than addressing each requirement independently, these goals group related requirements around the main architectural challenges of ORVAL. They provide the principles that guide the architectural decisions described throughout this chapter.

5.1.1. *Reliable Access to Telemetry Data*

The first design goal is reliable access to telemetry data. ORVAL shall remain capable of providing access to telemetry when an individual backend or database service becomes unavailable, provided that a redundant service remains operational. The architecture therefore needs to separate the visualization layer from individual service instances and provide alternative service paths that can be activated when required.

5.1.2. *Deterministic Service Behaviour*

The second goal is deterministic service behaviour. Service failures, failover and recovery shall result in predictable system transitions rather than ad-hoc or user-dependent behaviour. The architecture therefore defines explicit conditions under which a service is considered unavailable, when a failover is initiated, and when a recovered primary service can become active again. This deterministic behaviour is particularly important in a multi-user environment, where different users must not independently trigger conflicting routing decisions.

5.1.3. Autonomous Fault Tolerance and Recovery

The third goal is autonomous fault tolerance and recovery. Once automatic routing is enabled, ORVAL should be capable of detecting service failures and performing the required service transition without manual intervention. Fault handling does not end with failover: the architecture must also continuously monitor the unavailable primary service and determine when it has recovered sufficiently to resume operation. This results in a complete fault-handling cycle consisting of health monitoring, failure detection, failover, recovery detection, and return-to-home.

5.1.4. Multi-User Consistency

A fourth goal is multi-user consistency. Service transition must be coordinated at application-instance level so that components within the same ORVAL instance are not unintentionally distributed across different service paths. In particular, recovery of the primary service must not cause newly connected users to be routed back to the primary while existing users remain on the redundant service. Routing decisions therefore need to be coordinated through shared application state rather than being determined independently by each dashboard component or request.

5.1.5. Continuous Telemetry Visualization

The architecture must furthermore support continuous telemetry visualization during service transitions. Failover and recovery should be transparent to users wherever technically possible, avoiding unnecessary reconnections or manual service selection. The transition mechanisms must therefore preserve the connection between the visualization layer and the currently active service path while allowing that path to change when required.

5.1.6. Operational Independence, Maintainability and Deployability

Finally, the architecture is designed with operational independence, maintainability and deployability in mind. ORVAL is intended to operate in an environment where reliance on external Internet services cannot be assumed. At the same time, its components and deployment process should remain sufficiently structured and documented to allow the system to be operated and maintained by engineers other than its original developer.

Together, these design goals translate the requirements from Chapter 4 into a set of architectural principles for ORVAL. The remainder of this chapter describes how these principles are realized through the overall system architecture, the frontend, backend, and database layers, the health monitoring and fault-handling mechanisms, and the routing and state-management approach.

5.2. Overall System Architecture

ORVAL is designed as a layered telemetry visualization framework in which the user-facing frontend is separated from the services responsible for processing requests and providing telemetry data. This separation allows the visualization layer to remain independent from individual backend and database instances and provides the basis for the redundancy and fault-tolerance mechanisms described in the remainder of this chapter.

The reference architecture consists of a frontend layer and two alternative service paths. The primary service path is referred to as Alpha (α), while the redundant service path is referred to as Beta (β). Alpha represents the preferred service path during normal operation. Beta provides an alternative path that can be activated when the Alpha service path becomes unavailable. These names are used consistently throughout the remainder of this thesis to distinguish the primary and redundant paths.

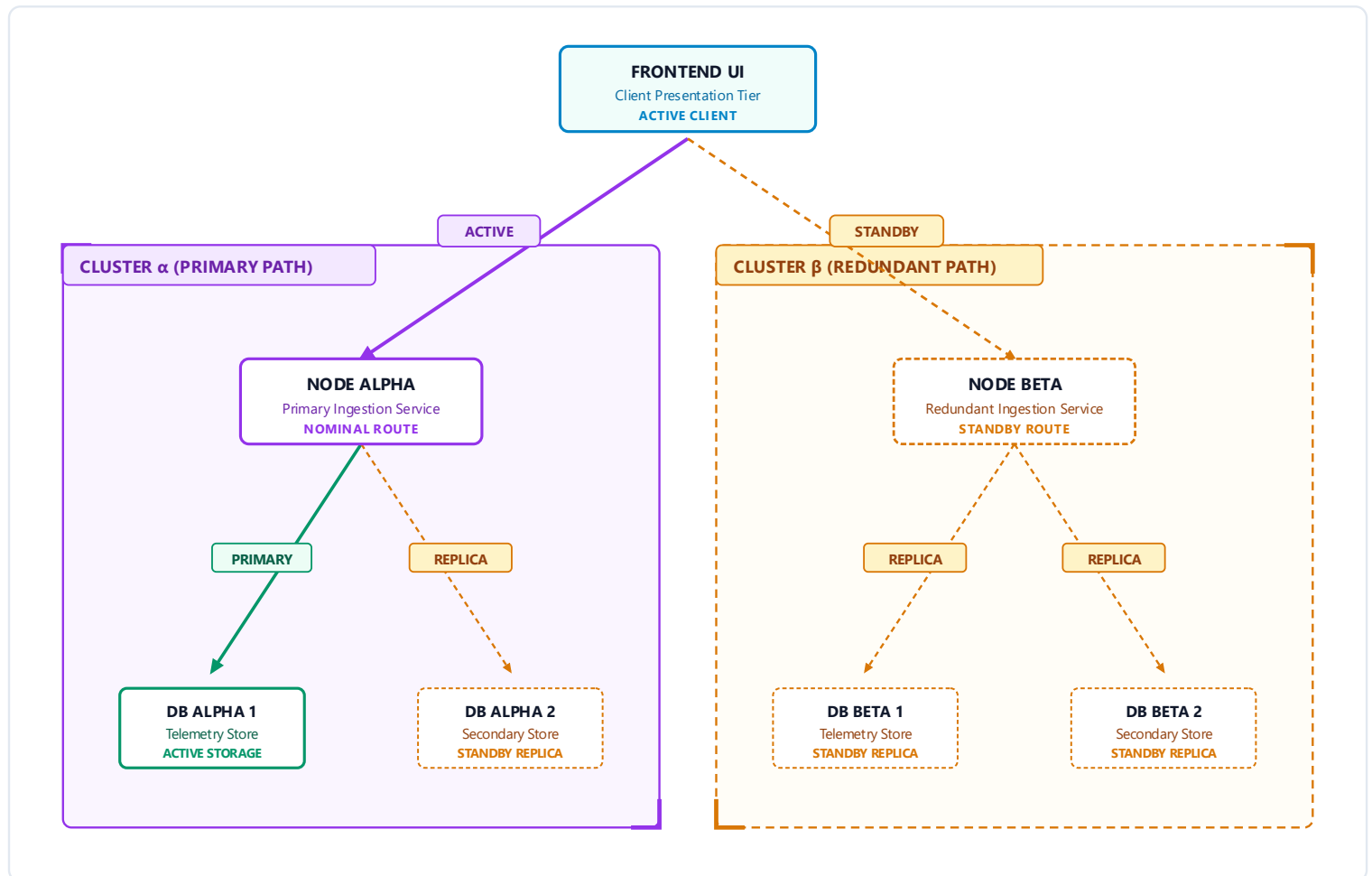


Figure 4: High-level reference architecture of ORVAL.

As shown in Figure 4, the ORVAL frontend provides the user-facing interface through which users interact with telemetry visualizations and dashboards. Requests from the frontend are directed towards the currently active service path. In the reference configuration, this consists of either the Alpha or Beta backend and its associated database instances. The routing and state-management mechanisms determine which service path is active, while the frontend remains independent of the specific backend instance currently serving the requests.

The Alpha service path contains Backend α (Node Alpha) and its associated database instances, while the Beta service path contains Backend β (Node Beta) and its associated database instances. During normal operation, Alpha is the active path and requests are handled through this path. Beta remains available as the redundant alternative. When the active service path becomes unavailable, the architecture allows the routing state to be changed so that requests can instead be handled through Beta. The mechanisms responsible for detecting such failures and performing this transition are described in Sections 5.6 through 5.8.

The architecture shown in Figure 4 represents the standard configuration used to explain and validate ORVAL, consisting of one frontend, two service paths, and two database instances associated with each path. The architecture is presented using this standard configuration rather than as a limitation on the conceptual separation between primary and redundant service paths. Supporting additional service or database instances would require corresponding extensions to the configuration, monitoring, and fault-tolerance mechanisms. The Alpha and Beta paths therefore represent the primary and redundant roles within the architecture rather than a fixed limitation on the number of services that ORVAL can support.

An important characteristic of the architecture is that service selection is coordinated at application-instance level rather than independently by individual dashboard components or requests. This is necessary because a service transition must result in a consistent active path for all users. For example, when Alpha becomes unavailable, subsequent requests within an ORVAL application instance are directed to Beta as part of the same routing decision. Similarly, when Alpha recovers, the application instance coordinates the return to Alpha rather than allowing individual dashboard components to independently select the recovered service. The detailed routing and consistency mechanisms are discussed in Section 5.10.

The overall architecture therefore provides the structural foundation for ORVAL's fault-tolerant behaviour. The frontend, backend, and database layers establish the normal data flow, while the distinction between Alpha and Beta provides alternative service paths that can be selected based on the current system state. The following sections examine these architectural layers individually, starting with the frontend architecture.

5.3. Frontend Architecture

The ORVAL frontend is implemented using `Vue.js` 3 and follows a component-based architecture based on the Composition API. The application is built and served using Vite, while Pinia is used for centralized state management and Vue Router provides client-side navigation and access control. The frontend is organized into route-level views, reusable interface components, state stores, services, and configuration modules. This separation allows user-interface functionality, application state, and supporting services to be maintained independently while working together as a single-page application.

The frontend is primarily responsible for providing the user interface through which telemetry data is selected, visualized, and managed. The main workspace is implemented in `Worksheet_Canvas.vue`, which manages the visualization canvas and the arrangement of individual chart components. Apache ECharts is used as the underlying visualization engine, with dedicated Vue components providing different visualization types. Users can configure parameters, time ranges, and visualization properties, while individual visualization cards can be moved, resized, locked, and combined. The frontend therefore not only renders telemetry data, but also manages the interaction and configuration required to construct telemetry dashboards.

The visualization layer supports both historical and real-time telemetry. For historical data, the frontend requests a specified time range from the backend and transforms the returned data into the format required by the visualization components. In live mode, the frontend periodically requests new telemetry data using a client-side polling mechanism. Newly received points are appended to the existing series, while points that fall outside the active time window are removed from the client-side data structure. This allows the displayed telemetry to be updated continuously without requiring a complete page reload.

Dashboard configuration and application state are managed through a combination of component-level state and centralized Pinia stores. `DashboardStore.js` maintains shared telemetry parameter information, while the worksheet maintains visualization configurations, layouts, dashboard tables, and time-related state. Dashboard and deck configurations can be persisted through local storage and backend storage, allowing users to restore previously configured visualization environments. The frontend also performs selected telemetry processing locally, including the calculation of virtual parameters and statistical operations such as moving averages and anomaly detection. These operations reduce the need to transfer already processed visualization data back and forth between the frontend and backend.

Communication between the frontend and backend is exclusively performed through REST-based HTTP requests using JSON. The frontend does not establish a direct connection to the telemetry database; all database access is mediated by the backend service. Requests are handled through a shared frontend communication layer that centralizes interaction with the backend and network-related behaviour. Backend endpoints for the primary and redundant service paths are supplied through the application configuration, allowing the frontend to communicate with both Alpha (α) and Beta (β).

An important architectural characteristic of ORVAL is that the frontend also participates directly in the system's fault-tolerance mechanisms. Unlike a conventional frontend that relies entirely on an external load balancer or gateway for service selection, ORVAL maintains awareness of both Alpha and Beta. The frontend maintains the currently active service path and can change this path when the active service becomes unavailable. The frontend also performs periodic service health checks and coordinates recovery monitoring for the primary service. Consequently, service selection and failover coordination form part of the frontend architecture rather than being completely hidden from it. The detailed mechanisms used for failure detection, failover, and recovery are discussed in Sections 5.6 through 5.9.

Despite this responsibility, the frontend remains separated from the database layer. Its interaction with telemetry storage is performed through backend APIs, while database connectivity, query execution, and database-level redundancy are handled by the backend. This establishes a clear architectural boundary: the frontend manages presentation, user interaction, client-side state, and selected client-side processing, while the backend mediates access to persistent telemetry data and provides the database-facing functionality.

Overall, the frontend architecture combines visualization, interaction, state management and client-side service coordination within a single application layer. The architecture allows ORVAL to provide a responsive telemetry visualization environment while also participating in the deterministic routing and fault-tolerance behaviour required by the system. The following sections examine the backend and database architectures that support these frontend capabilities.

5.4. Backend Architecture

The ORVAL backend is implemented using `Node.js` and `Express.js` and provides the API layer between the frontend and the telemetry databases. Its architecture follows a layered structure in which incoming HTTP requests are handled by route definitions and middleware, processed by controllers, and subsequently passed to the appropriate data-access or storage mechanisms. This separation establishes a clear boundary between the user-facing frontend and the underlying telemetry and application data.

The backend is responsible for resolving telemetry parameters, retrieving historical and newly available measurements, and applying the server-side processing required to transform the underlying telemetry data into responses suitable for visualization. Real-time telemetry is supported through dedicated backend operations that provide only newly available measurements, allowing the frontend to update its visualizations incrementally.

The backend also forms the database-access boundary for the telemetry infrastructure. Each service path is associated with redundant database targets, allowing database-level availability to be handled independently from application-node failover. The concrete database connectivity, query, and pooling mechanisms are described in Section 6.2.

In addition to telemetry retrieval, the backend manages application-level persistence that does not depend on the telemetry database. Dashboard blueprints, dashboard decks, user accounts, and system configuration are stored as JSON-based files on the backend. This separates application state from the telemetry data stored in MySQL and allows application-level configuration to be managed separately from the operational state of the telemetry database. The backend provides the application interfaces required to manage dashboard and deck configurations, users, and system settings.

Security and access control are enforced at the backend boundary through authentication and role-based authorization. This ensures that protected operations cannot rely solely on client-side restrictions.

The backend also provides logging, status, and diagnostic capabilities to support operational monitoring and failure investigation. Their concrete implementation is described in Sections 6.5 and 6.9.

ORVAL is deployed using separate backend instances for the Alpha (α) and Beta (β) service paths. Both instances run the same backend application and therefore share the same architectural structure, while their runtime configuration determines their respective roles and database targets. The instances operate independently and do not directly communicate with one another. Coordination between the two service paths is handled by the frontend routing mechanism described in Section 5.3. Consequently, redundancy exists at two distinct levels: the frontend coordinates switching between backend service instances, while each backend independently provides redundancy between its assigned database pools.

The backend architecture therefore forms the central application and data-access layer of ORVAL. It exposes the API through which the frontend accesses telemetry and application functionality, performs server-side telemetry processing, manages persistent application state, enforces security, and provides database-level redundancy. This separation of responsibilities allows the frontend to remain focused on visualization, interaction, and service routing, while the backend provides a controlled interface to the underlying telemetry and application data. The following section examines the database architecture that supports this data access layer.

5.5. Database Architecture

The telemetry databases used by ORVAL are part of the existing infrastructure provided by Redwire and were therefore not designed or created as part of this project. The database schema, telemetry storage structure, and data ingestion mechanisms were already established before the development of ORVAL. The development effort focused instead on understanding the existing data model and implementing the integration layer required for ORVAL to reliably connect to, query, and process the stored telemetry data. Consequently, this section focuses on the interaction between ORVAL and the existing telemetry infrastructure rather than documenting the underlying database schema.

The existing telemetry infrastructure is based on MySQL and provides access to telemetry parameters and their associated measurements. Telemetry data is organized according to the underlying value types, while a separate metadata structure provides the information required to identify and retrieve individual parameters. Each telemetry record contains information identifying the parameter, temporal information, and the corresponding telemetry value. This organization allows ORVAL to retrieve different types of telemetry through a common application interface without requiring changes to the existing database structure.

An important characteristic of the telemetry model is the distinction between the time at which a measurement is generated and the time at which it is stored in the database. The generation time represents the temporal context of the telemetry measurement and is used as the primary temporal reference for historical visualization. The storage time represents when the record becomes available in the database and is used by ORVAL when retrieving newly stored telemetry during live visualization. Maintaining this distinction allows the backend to handle telemetry that may be stored at a different moment from its original generation time.

Because telemetry parameters can be stored according to different value types, ORVAL does not rely on a single fixed data source for all telemetry requests. When a parameter is requested, the backend first retrieves the corresponding metadata and uses this information to determine how the parameter should be accessed. The appropriate data source is then selected through a controlled mapping mechanism before the telemetry query is constructed. This approach allows the same API to retrieve parameters with different underlying representations while keeping the existing database structure unchanged.

ORVAL establishes its connection to the existing databases through a dedicated database-access layer in the backend. This layer uses the MySQL client library to manage database connections and execute queries on behalf of the application. Each backend instance maintains connections to a primary and redundant database target. Each backend instance can therefore access both its primary and redundant database targets, providing the architectural basis for database-level redundancy. This provides redundancy at the database-access level independently of the redundancy between the Alpha (α) and Beta (β) service paths.

The backend also performs server-side telemetry processing before returning data to the frontend. Historical and live telemetry follow different access patterns: historical requests are processed according to the requested time range, while live requests retrieve newly available telemetry incrementally. This allows the database layer to support both historical analysis and continuous visualization while limiting unnecessary data transfer to the frontend.

The database architecture also relies on indexes to support the main telemetry access patterns. ORVAL verifies the availability of the indexes required for its database operations as part of its operational integration with the existing infrastructure. This does not constitute a redesign of the existing database architecture, but ensures that the database remains suitable for ORVAL's access patterns.

Database redundancy is handled within the ORVAL database-access layer. Each backend instance can use a primary and a redundant database target, allowing telemetry retrieval to continue when the active target becomes unavailable, provided that the redundant target is operational and contains the required data. ORVAL itself does not manage the underlying database replication or telemetry-ingestion infrastructure. The synchronization and availability of the underlying data sources therefore remain outside the architectural boundary of the application. ORVAL is responsible for detecting database availability problems and selecting the appropriate available data source.

The database layer consequently consists of two distinct aspects: the pre-existing telemetry infrastructure provided by Redwire and the ORVAL integration layer developed as part of this project. The existing infrastructure defines how telemetry is stored and made available, while ORVAL provides the connectivity, parameter resolution, querying, server-side processing, and database-level fault-tolerance mechanisms required by the visualization framework. This separation ensures that ORVAL can build upon the existing telemetry environment without taking responsibility for the creation or management of the underlying database system.

Overall, the database layer provides the persistent telemetry source on which ORVAL's visualization capabilities depend, while the backend integration layer abstracts the underlying storage structure from the frontend. The frontend can therefore request telemetry through a consistent API without needing to know how the data is physically organized within the database. The following sections build upon this integration architecture by examining how ORVAL monitors service availability and detects failures.

5.6. Health Monitoring Architecture

To support deterministic fault handling, ORVAL continuously monitors the availability of both its application services and database connections. Health monitoring is implemented at two architectural levels. The frontend monitors the Alpha (α) and Beta (β) service paths through HTTP health checks, while each backend independently monitors its configured database targets. This separation allows the health of application services and their underlying data sources to be observed independently.

Heartbeat

A heartbeat is a periodic, lightweight health check used to determine whether a monitored component remains reachable and responsive.

The frontend implements a global heartbeat mechanism that periodically probes both Alpha and Beta independently. The resulting availability information is maintained as shared frontend state and provides continuous visibility of the operational status of the two service paths and their associated database targets.

An important design characteristic is that the direct node probes performed by the heartbeat are observational. A failed probe updates the recorded health state but does not independently trigger a failover. This distinction separates health monitoring from the failure-detection and routing mechanisms described in Sections 5.7 and 5.8. The heartbeat therefore provides regularly updated information about the system without directly making service-transition decisions.

Database health is monitored independently within the backend database-access layer. Each backend monitors both its primary and redundant database targets and maintains their health states separately. This allows the backend to monitor the redundant database target even while the primary target is being used for normal queries.

The latest database health information is exposed to the frontend through the backend status endpoint, allowing service monitoring to access database availability without directly depending on a new database operation for each frontend probe. A separate diagnostic interface provides more detailed information for operational monitoring and troubleshooting.

The primary and redundant database pools are therefore monitored independently, with their health states maintained separately from the active database routing state. This allows database monitoring to remain independent from the routing decision itself.

Overall, ORVAL combines frontend service monitoring with backend database monitoring to maintain a continuously updated view of system availability. The next section builds upon this monitoring layer by describing how ORVAL distinguishes temporary communication problems from confirmed service failures.

5.7. Failure Detection

The health monitoring mechanisms described in Section 5.6 continuously provide information about the availability of ORVAL's service and database components. However, a single unsuccessful health check or request does not immediately indicate a confirmed failure. To distinguish temporary communication problems from actual service failures, ORVAL applies a deterministic three-attempt failure detection mechanism.

At the application level, failure detection is based on the currently active service path. When a request fails, ORVAL retries the same operation up to three times before declaring the service unavailable. The attempts are performed sequentially and separated by a defined delay. If any attempt succeeds, the request is completed normally, and no failover is initiated. A new request starts a new failure-detection sequence, meaning that isolated failures from previous requests do not accumulate into a global failure counter.

The failure classification is deliberately restrictive. Network-level failures, request timeouts, and server-side HTTP errors with status codes of 500 or higher are treated as infrastructure failures and therefore count as failed attempts. In contrast, HTTP 4xx responses such as 401 Unauthorized, 403 Forbidden, and 404 Not Found are not considered service failures. These responses indicate that the backend is reachable and has processed the request, even if the request operation itself cannot be completed. An expired authentication session, for example, is handled as an authentication issue rather than as a backend failure.

The same principle is applied at the database level. When a database operation against the active database target fails, the failure contributes to the same deterministic three-attempts principle. Connection-related failures are therefore treated as database availability failures rather than as ordinary query errors. If one of the attempts succeeds, the database operation continues normally. Only after all three attempts fail is the primary database considered unavailable, allowing the database routing mechanism to transition to the redundant target.

The detection process is governed by explicit timeout and retry parameters. The design therefore provides a bounded detection window whose duration depends on the configured request timeout and the delay between failed attempts. Under the default configuration, the resulting detection window is deterministic for a given failure type, while different failure modes may result in different observed detection times.

The important design characteristic is that a service failure is not confirmed after a single unsuccessful attempt, but requires three consecutive unsuccessful attempts within the same request lifecycle.

An additional consideration is the concurrent nature of telemetry visualization. A dashboard can generate multiple requests simultaneously, meaning that several components may encounter the same backend failure at approximately the same time. ORVAL therefore ensures that concurrent requests respect a shared routing state. If another request has already completed the failure detection and changed the active service, remaining requests follow the newly established routing state rather than independently continuing to target the failed service. This prevents multiple concurrent requests from independently initiating the same service transition.

The distinction between failure detection and failover is maintained explicitly in the implementation. The three-attempt mechanism establishes that the currently active target can no longer reliably serve the request. The subsequent change of the active service and routing of traffic to the redundant target belong to the autonomous failover mechanism described in Section 5.8.

This approach provides a deterministic boundary between temporary request failures and confirmed service failures. By combining explicit error classification, sequential retries, configurable timing, and synchronization between concurrent requests, ORVAL reduces the likelihood that transient communication problems result in unnecessary service transitions.

5.8. Autonomous Failover

Once a failure of the active service path has been confirmed, ORVAL autonomously redirects the affected request to the redundant service path. This mechanism is implemented in the frontend request orchestration layer and does not require user intervention. The failover process is designed to ensure that the redundant path is successfully reachable before the new route is committed.

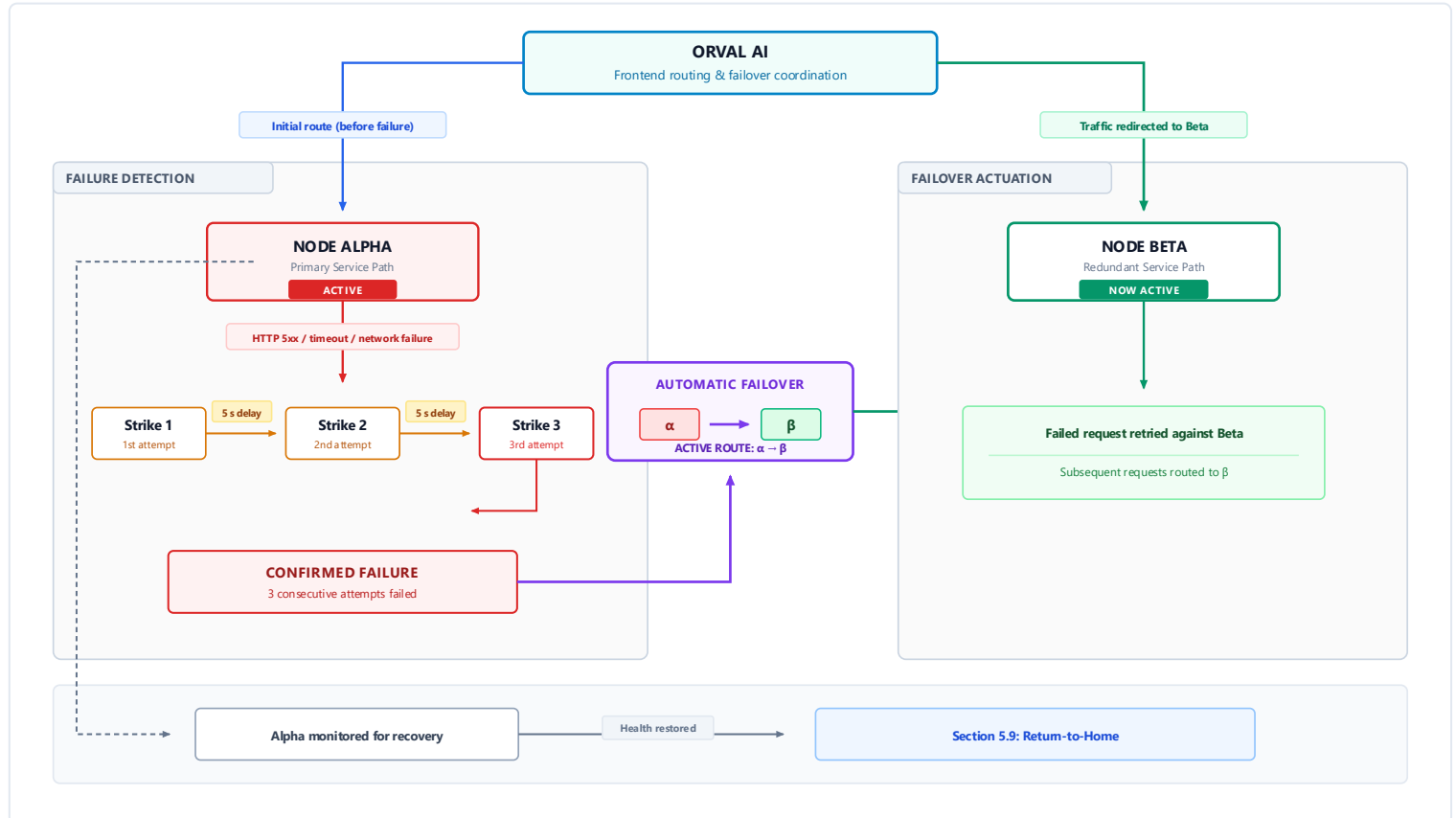


Figure 5: Autonomous failover flow from the primary service path to the redundant service path.

Figure 5 illustrates the complete transition from failure confirmation to operation on the redundant service path. Under normal operation, the frontend routes requests to Node Alpha, which represents the primary service path. Following the three consecutive failed attempts described in Section 5.7, the failure is confirmed, and the failover process is initiated.

An important design decision is that ORVAL does not immediately change the active route when failure is confirmed. Instead, the request that experienced the failure is first re-issued against Node Beta. This also acts as an additional availability check of the redundant path. Only when this request receives an acceptable response does ORVAL commit the routing change from Alpha to Beta. This prevents the application from switching its active route to a redundant node that is itself unavailable.

After a successful response from Node Beta, the active routing pointer is updated so that subsequent requests are directed to the redundant service path. The response obtained from Beta is also returned to the original caller. Consequently, the request that triggered the failover can still be completed without requiring the frontend component that initiated it to explicitly handle the server transition.

The failover mechanism also accounts for concurrent requests. Multiple dashboard components may detect the failover of Node Alpha at approximately the same time. ORVAL therefore coordinates these requests through the shared routing state so that concurrent requests respect a transition that has already been completed by another request. If the active route has already changed to Node Beta, remaining requests follow the newly established route rather than independently initiating another transition. This provides a lightweight synchronization mechanism without introducing a separate coordination service.

The failover sequence can therefore be summarized as follows:

Confirmed failure → retry the affected request on Beta → verify a successful response → commit the routing change (α to β) → route subsequent requests to Beta

Once the switch has been completed, ORVAL starts monitoring the previously failed Alpha path for recovery. The recovery process and the conditions under which the system automatically returns to Alpha are discussed separately in Section 5.9.

5.9. Return-to-Home Mechanism

The failover mechanism described in Section 5.8 redirects requests from the primary service path, Alpha, to the redundant service path, Beta, when a confirmed failure is detected. To restore the preferred operating state once Alpha becomes available again, ORVAL implements an autonomous return-to-home mechanism. This mechanism continuously monitors the primary service path while Beta is active and automatically restores Alpha as the active route when the recovery conditions are met.

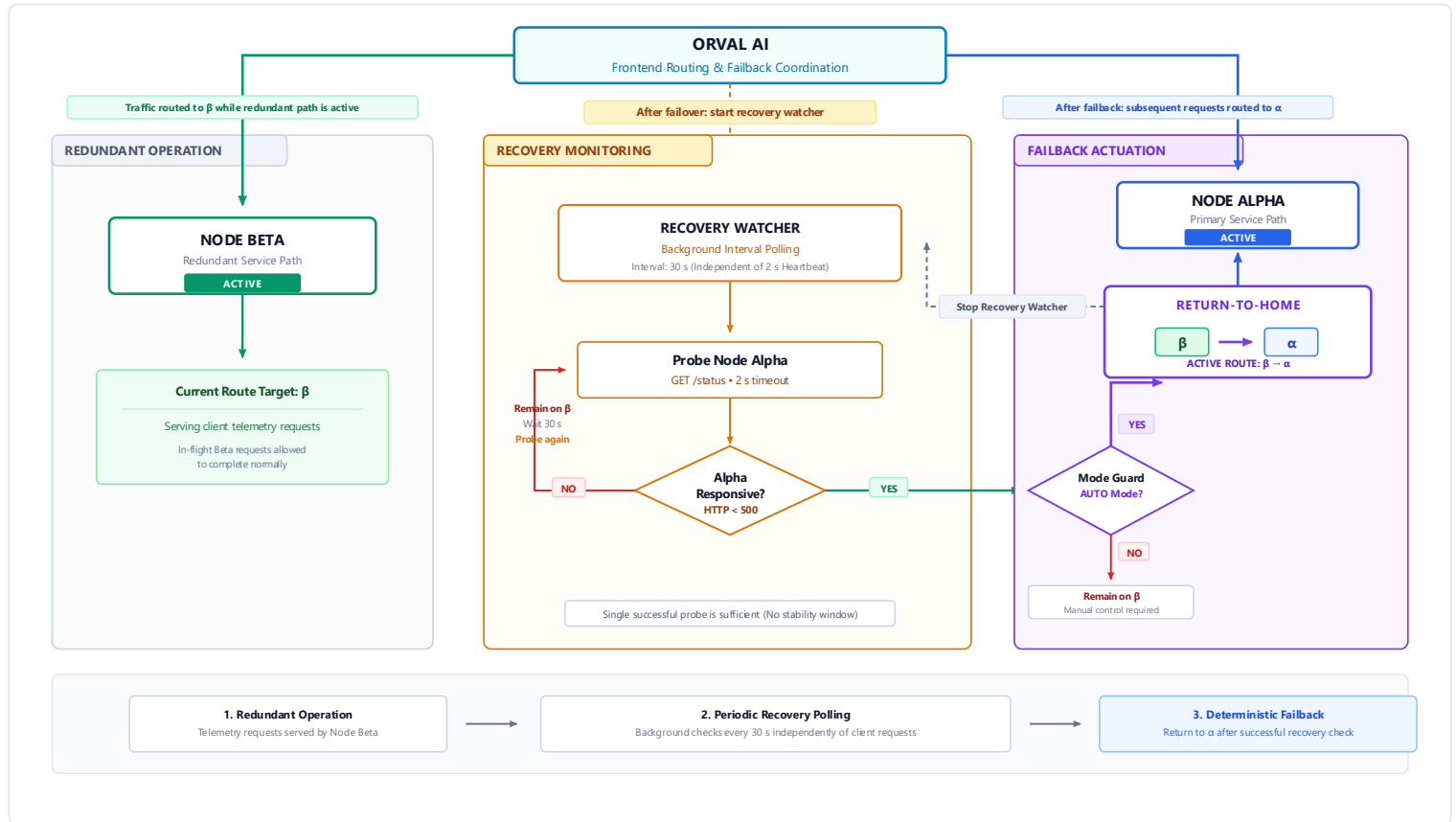


Figure 6: Autonomous recovery process from the redundant service path back to the primary service path

Figure 6 illustrates the implemented return-to-home process. After a successful failover to Beta, the frontend starts an independent recovery monitor that periodically checks the availability of Node Alpha. The recovery monitor operates independently of the regular service heartbeat and evaluates whether Alpha has become responsive again. A response with an HTTP status below 500 is considered sufficient to indicate that Alpha is responsive at the application layer.

Unlike the three-attempt failure detection mechanism described in Section 5.7, the recovery mechanism in this iteration evaluates a single successful probe (HTTP status below 500) to initiate failback. This design prioritizes rapid return to the preferred operational path. However, this introduces an architectural trade-off: an application service whose web server is responsive but whose downstream database connection remains degraded could trigger an eager failback. This risk is partially mitigated by backend-tier database failover and administrative manual override modes, but implementing a consecutive-success stability window represents an identified path for future hardening. If Alpha does not respond within the timeout or returns an HTTP status of 500 or higher, the active route remains unchanged, and the watcher waits for the next 30-second interval before probing Alpha again.

Before performing an automatic return-to-home, ORVAL also evaluates the configured node operating mode. Automatic failback is only performed when the system is operating in AUTO mode. If automatic operation is disabled, the system remains on the current node and leaves the decision to return to Alpha under manual control. This administrative guard prevents the recovery mechanism from overriding an intentional operational choice.

When the recovery conditions are satisfied, ORVAL updates its active routing state from Beta to Alpha. Subsequent requests are therefore directed to the primary service path. Requests that were already in progress through Beta are allowed to complete normally rather than being interrupted by the routing change. After the active route has been restored to Alpha, recovery monitoring is no longer required while Alpha remains the active path.

The return-to-home mechanism is deliberately implemented at the frontend service-routing level. It is therefore independent of the separate database recovery mechanism implemented within the backend database-access layer. Its purpose is to restore the preferred service path for telemetry access, rather than to verify or manage the underlying database recovery process.

Overall, the combination of autonomous failover and return-to-home provides a deterministic recovery cycle: ORVAL switches from Alpha to Beta after a confirmed failure and subsequently returns from Beta to Alpha after a successful recovery check. This allows the redundant path to function as a temporary alternative while preserving Alpha as the preferred operating path whenever it is available.

5.10. User Routing & Consistency

A fault-tolerant system must not only recover from service failures, but also maintain consistent routing behaviour while operating across redundant service paths. If routing decisions were made independently by individual dashboard components, worksheets, or requests, different parts of the same application could temporarily communicate with different service paths. For example, after a failure of the primary path, one component could switch to the redundant path while another component continues attempting to use the primary path. This split-component behaviour could result in inconsistent request outcomes and complicate the interpretation of system behaviour across dashboard components.

ORVAL addresses this issue by maintaining the active service path at the application-instance level. The frontend maintains a shared routing state that determines whether subsequent requests are directed to the primary path (Alpha) or the redundant path (Beta). Consequently, routing is not independently maintained by individual dashboard components. Once the active route changes during failover or return-to-home, subsequent requests originating from the ORVAL instance use the updated routing target. This provides a consistent routing view across the different components and background processes of the application.

The routing mechanism also accounts for concurrent requests during a transition. Requests that are already in progress are not forcibly cancelled when the active route changes. They are allowed to complete on their original service path, while requests initiated after the routing change use the newly active path. During failover, concurrent requests therefore respect the shared routing state and transition to the newly active service path once the routing change has been established. This allows routing transitions to occur without destructively interrupting requests that are already being processed.

The consistency guarantee is primarily scoped to an individual ORVAL application instance. Each operator workstation maintains its own frontend runtime and routing state; ORVAL does not implement a distributed consensus mechanism between independent workstations. Consequently, different workstations may temporarily operate on different service paths during a transition. However, because they use the same deterministic failover and recovery mechanisms, they independently converge towards the configured primary path after recovery.

This design prevents split-component behaviour within an ORVAL application instance while keeping the routing mechanism autonomous and deterministic. The mechanisms used to maintain and update the routing state are discussed in the following section.

5.11. State Management

ORVAL maintains a small set of shared frontend state variables to coordinate service routing and communicate the current system status to the user interface. The most important routing state represents the currently active service path and determines which backend path is used for subsequent API requests. By maintaining this value outside individual dashboard components, routing decisions remain independent of the lifecycle or state of individual worksheets, charts, or other interface elements.

The routing state is complemented by a reactive network-status object that exposes the current operational state to the frontend. This state contains information such as the active service path, connectivity status, and service health indicators. Interface components can therefore react to changes in the system state without directly controlling the routing mechanism. The separation between routing state and presentation state ensures that the state used for request execution remains independent from the state used to represent that information in the user interface.

The relationship between these two state layers is illustrated in Figure 7. During nominal operation, both layers indicate the primary service path. After a confirmed failure and successful failover, the execution-layer routing pointer is changed to the redundant path and the presentation-layer state is updated accordingly. While operating on the redundant path, the recovery watcher remains active. Once the primary path becomes responsive and automatic recovery is permitted, both state representations are changed back to the primary path, and the recovery watcher is stopped. This maintains a consistent routing state within the ORVAL application instance throughout the failover and recovery lifecycle.

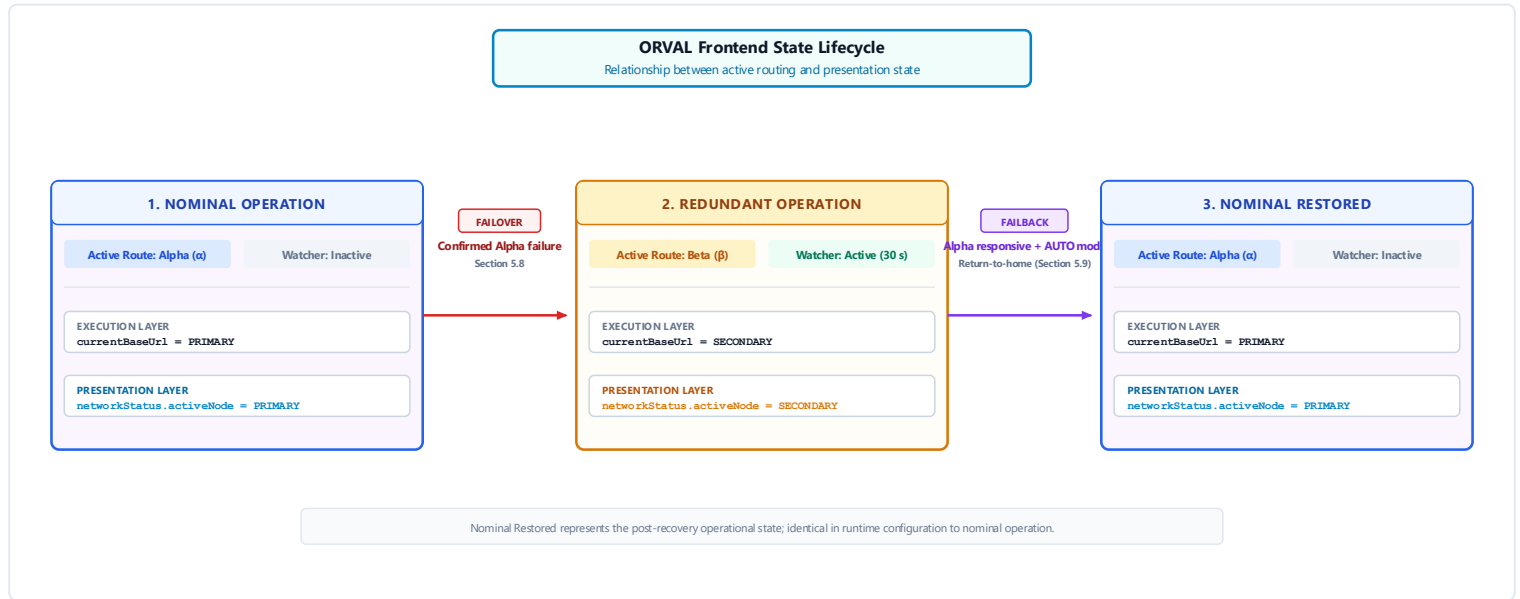


Figure 7: ORVAL frontend state lifecycle and corresponding route state.

The routing state is maintained in application-local memory and is shared across the components and background processes of the ORVAL application instance. This ensures that service selection is determined centrally rather than independently by individual components. The state remains local to each ORVAL application instance; independent operator workstations therefore maintain their own routing state. ORVAL does not use a distributed consensus mechanism between workstations. Instead, each instance applies the same deterministic failover and return-to-home mechanisms and independently converges towards the configured primary service path after recovery.

5.12. Design Decisions & Trade-offs

The architecture of ORVAL reflects several deliberate design decisions intended to balance fault tolerance, deterministic behaviour, implementation complexity and operational maintainability. Rather than introducing additional infrastructure or complex distributed coordination mechanisms, the design prioritizes predictable behaviour within the existing system architecture. The most relevant architectural decisions and their associated trade-offs are summarized in Table 10.

Table 10: Overview of the main architectural decisions and trade-offs in ORVAL

Architectural decision	Implemented approach	Main alternative	Rationale	Main trade-off
Failure detection	Deterministic, sequential three-attempt retry mechanism	Centralized circuit breaker with adaptive backoff	Provides predictable failure detection behaviour and prevents isolated transient failures from accumulating across unrelated requests	A failed request may experience increased latency while the retry sequence is completed
Failover coordination	Frontend-coordinated routing between Alpha and Beta	Centralized load balancer or reverse proxy	Keeps routing coordination within ORVAL and avoids introducing an additional infrastructure component between clients and the redundant service paths	Routing state remains local to each application instance, so independent workstations may temporarily use different service paths during a transition
Redundancy architecture	Separate application-tier and database-tier failover mechanisms	Coupled full-stack failover	Isolates database connectivity problems from application-node failover and allows each redundancy layer to recover independently	The interaction between the two redundancy layers increases the number of operational states that must be monitored

Recovery strategy	Autonomous return-to-home with administrative override	Sticky failover requiring manual recovery	Automatically restores the configured primary service path while retaining the possibility of operator intervention during maintenance	Recovery is based on application-level responsiveness and may therefore return to a primary node before all underlying services are fully operational
Application state distribution	Application-level propagation of selected state-changing operations to both service paths	Database-level replication or clustering	Allows ORVAL to distribute relevant application state across both service paths without modifying the underlying database architecture	A failed write on one service path can temporarily result in divergent application state between the two paths

The first design decision concerns failure detection. ORVAL uses a fixed three-attempt sequence with configured timeouts and delays rather than a global failure-state mechanism based on accumulated request history. This makes the behaviour of each request predictable and prevents failures from unrelated requests or earlier time periods from influencing a later routing decision. The trade-off is that the retry sequence deliberately introduces additional latency during an outage before failover is confirmed. This reflects the design choice to prioritize deterministic failure confirmation over the fastest possible reaction to an isolated failure.

A second decision is to coordinate service failover at the frontend rather than through an additional centralized routing component. This allows the ORVAL application to maintain the active service path and perform failover and recovery without requiring a separate routing layer. The consequence is that routing state remains local to each application instance, meaning that independent workstations may temporarily use different service paths during a transition.

The separation between application-tier and database-tier redundancy represents another important architectural choice. A database failure does not automatically imply that the corresponding application node must be considered unavailable, since the backend can independently switch between its configured database pools. This provides fault containment between the two layers. The trade-off is increased state complexity because the operational condition of an application node and the health of its database connections must be considered separately.

For recovery, ORVAL adopts an autonomous return-to-home mechanism instead of permanently remaining on the redundant service path after failover. Once the primary path becomes responsive and automatic recovery is enabled, routing is returned to the primary path without requiring manual intervention. This supports autonomous recovery while the administrative override provides control during planned maintenance. The main limitation is that application-level responsiveness does not necessarily prove complete service health, meaning that a primary node that responds prematurely could cause a failback before all underlying services are fully restored.

Finally, ORVAL uses application-level propagation of selected state-changing operations to both service paths. This approach distributes relevant application state to both service paths while keeping the existing database architecture outside the scope of the framework. However, the mechanism does not provide distributed transaction guarantees or database-level replication. If one service path is unavailable when a state-changing operation is performed, the corresponding application state may temporarily diverge between the two paths. This limitation is accepted in favour of a simpler integration approach that does not require changes to the underlying infrastructure.

Overall, these decisions show that ORVAL prioritizes deterministic behaviour, fault containment and operational simplicity over more complex distributed coordination mechanisms. The resulting architecture provides autonomous failure handling while keeping the individual mechanisms sufficiently independent to be understood, monitored and maintained. These trade-offs form the basis for evaluating whether the implemented framework satisfies the requirements identified in Chapter 4.

6. Implementation

This chapter describes the implementation of ORVAL and addresses the practical realization of the architecture and design decisions established in the previous chapter. It focuses on how the defined requirements and architectural components were translated into a functional telemetry visualization framework.

Based on the architecture and system design presented in Chapter 5, this chapter first introduces the technology stack used to implement ORVAL. The implementation of the database layer is then described, including database connectivity, connection pooling, query execution, and interaction with the primary and secondary database targets. The chapter subsequently discusses the implementation of the backend and frontend components, before focusing on the mechanisms that enable ORVAL's fault-tolerant behaviour, including health monitoring, failure detection, autonomous failover, and return-to-home. Finally, the deployment environment and security-related implementation considerations are discussed.

6.1. Technology Stack

ORVAL was implemented as a web-based application consisting of a `Vue.js` frontend and a `Node.js` backend, with MySQL used as the underlying telemetry database. The application is complemented by supporting technologies for data visualization, state management, authentication, logging, and containerized deployment. Table 11 provides an overview of the main technologies and their role within ORVAL.

Table 11: ORVAL technology stack and implementation roles

Layer	Technology	Role in ORVAL
Frontend	Vue.js 3 with Composition API	Implements the interactive user interface and dashboard components
Frontend	Apache ECharts	Renders telemetry time-series, gauges, and other data visualizations
Frontend	Pinia	Manages shared application state, including authentication and dashboard data
Frontend	Vue Router	Handles client-side navigation and access restrictions

Frontend	vue-grid-layout-v3	Provides the configurable grid used to arrange and resize dashboard widgets
Backend	Node.js with Express.js	Runs the backend services and exposes the Representational State Transfer (REST) API used by the frontend
Backend	mysql2/promise	Provides database connectivity, connection pooling, and SQL query execution
Backend	JavaScript Object Notation (JSON) Web Tokens	Supports authentication and role-based access control
Backend	Winston	Provides structured application logging and log rotation
Database	MySQL 8.0	Provides access to the existing telemetry data infrastructure
Deployment	Docker and Docker Compose	Containerize and coordinate the application components
Deployment	Nginx	Serves the compiled frontend application in the production environment

On the frontend, `Vue.js 3` and its Composition API form the basis of the user interface. The application is divided into reusable components that represent dashboard elements and interface functionality, while reactive state is used to update these components when telemetry or system information changes. Apache ECharts is responsible for the actual rendering of telemetry visualizations, including time-series plots and other graphical representations. The configurable worksheet interface is implemented using `vue-grid-layout-v3`, allowing dashboard widgets to be positioned and resized by the operator. Pinia is used for shared client-side state, while Vue Router manages navigation between the different application views and supports access restrictions for protected functionality. Vite is used as the frontend build tool to compile the Vue application into the static assets used for deployment.

The backend is implemented using `Node.js` and `Express.js`. It exposes the REST endpoints through which the frontend requests telemetry and system information and contains the application logic required to process these requests. Communication with the telemetry database is implemented using `mysql2/promise`. This layer provides the connection pools and SQL query execution used to communicate with the primary and secondary database targets. Authentication is implemented using JSON Web Tokens, while Winston provides application logging for monitoring and troubleshooting.

The telemetry data itself is stored in the existing MySQL 8.0 infrastructure. This database forms an external infrastructure dependency of ORVAL rather than a database designed or implemented as part of the project. ORVAL therefore starts at the application-side database access layer and uses the database through its configured connections and queries. The implementation of this interaction is discussed in more detail in Section 6.2.

For deployment, the application components are packaged using Docker. Separate containers are used for the frontend and backend, while Docker Compose is used to coordinate the required services and reproduce the multi-component environment. The frontend uses a multi-stage build in which the Vue application is compiled during the build stage and the resulting static files are served by Nginx in the production container. This separates the build dependencies from the environment required to serve the deployed application.

6.2. Database Implementation

ORVAL accesses the existing telemetry infrastructure through the `mysql2` library using its Promise-based API, allowing database operations to be handled asynchronously within the `Node.js` backend. Database connectivity is organized around two independent connection pools, one for the primary database target and one for the secondary target. Each pool is configured with a maximum of ten simultaneous connections and a connection timeout of five seconds. When all available connections are occupied, incoming requests are queued rather than creating additional connections. This provides controlled database access while allowing multiple telemetry requests to be processed concurrently.

The backend does not expose the individual database pools directly to the controllers. Instead, database operations are performed through a shared query layer that determines the appropriate database target before executing the query. This abstraction keeps database connectivity and target selection independent from the controllers, allowing the same telemetry operations to be executed regardless of the currently selected database target.

Telemetry retrieval is implemented through several types of SQL queries. Before telemetry values can be retrieved, ORVAL resolves the requested parameter metadata stored in the database. This metadata provides information such as the parameter identifier, name, units, and the corresponding telemetry data type. Based on this information, the backend determines which typed telemetry table contains the requested data. This allows a common API to retrieve parameters with different underlying data representations without requiring the frontend to know the database structure. For live telemetry, queries retrieve only records that have been stored since the timestamp supplied by the client and apply a result limit to prevent unnecessarily large responses. Historical queries instead use the parameter identifier and requested time range to retrieve the relevant telemetry interval.

For longer historical intervals, ORVAL performs data reduction directly in the database rather than returning every stored measurement to the browser. A configurable point limit is used to determine when the requested interval can be returned at its original resolution. When the interval exceeds this limit, the backend calculates a time-bucket size and uses SQL aggregation to group measurements into consecutive time intervals. For numerical telemetry, the query calculates the average, minimum, and maximum value within each bucket. The average provides a representative value for visualization, while the minimum and maximum values preserve information about variations within the interval. This reduces the amount of data transferred to the frontend while retaining relevant changes in the telemetry signal. For telemetry types for which numerical averaging is not applicable, the aggregation strategy is adapted to the underlying data type.

Database performance is further supported through indexes on the parameter identifier and the timestamps used by the main telemetry queries. During startup, ORVAL verifies whether the indexes required by its main telemetry queries are present and creates them when necessary. This acts as an operational safeguard when deploying the application against an environment in which the expected indexes are not yet available. The underlying telemetry database and its schema remain part of the existing infrastructure; ORVAL's implementation is limited to accessing this infrastructure and ensuring that the indexes required for its own queries are available.

In addition to query execution, the database layer performs lightweight connectivity checks against both database targets. These checks allow the backend to maintain an up-to-date view of database connectivity without relying solely on user-generated queries. Diagnostic database checks are also subject to a short caching interval to avoid unnecessary repeated requests when multiple clients request system status simultaneously. Together, connection pooling, targeted queries, in-database data reduction, indexing, and controlled health checks provide the database access layer required for efficient and reliable telemetry retrieval.

6.3. Backend Implementation

The ORVAL backend is implemented as a modular `Node.js` application using `Express.js`. Its structure separates request routing, authentication and authorization, application logic, database access, and logging into distinct components. This separation allows the controllers to focus on processing application requests without directly managing database connections or other infrastructure concerns.

Incoming requests first pass through the Express application defined in `index.js`, where global middleware is applied, and the different API route groups are registered. The routes organize the backend functionality into domains such as authentication, telemetry and dashboards, system administration, logging, and diagnostics. Protected routes are passed through authentication middleware before reaching the corresponding controller. This results in a consistent request flow in which the route determines the requested functionality, middleware verifies the caller's permissions, and the controller processes the request and constructs the response.

The controllers contain the main application logic for the different backend functions. For telemetry operations, for example, the dashboard controller validates the parameters supplied by the frontend, requests the required data through the database access layer, and formats the resulting data into JSON responses. Database connectivity itself is abstracted from the controllers through the shared data-access module described in Section 6.2. This prevents individual controllers from having to manage connection acquisition, release, or database-target selection and keeps database-specific logic centralized.

Authentication is implemented using JSON Web Tokens (JWTs). During login, the submitted credentials are verified against the locally stored user information, after which the backend generates a signed token containing the user's identity and role. Subsequent requests to protected endpoints include this token in the authorization header and are validated by authentication middleware before reaching the controller. Additional authorization middleware restricts administrative functionality and selected dashboard operations according to the user's assigned role. Global session revocation is also supported and is discussed further in Section 6.11.

Not all application state is stored in the telemetry database. ORVAL uses local JSON files for application-level information such as user data, system configuration, and saved dashboard layouts. Runtime information that only needs to exist while the backend is running, such as the currently active database target and cached database health information, is maintained in memory. This separation keeps application configuration and workspace information independent from the telemetry data source.

The backend also exposes dedicated status and diagnostic endpoints. A lightweight status endpoint provides process liveness and the latest database connectivity state without initiating a new database query for every heartbeat request. A separate diagnostic endpoint provides more detailed system information used by the administrative interface. These endpoints are implemented separately from ordinary telemetry requests so that system monitoring does not depend on application data retrieval.

Error handling is implemented at the controller level using try/catch blocks and explicit validation of incoming parameters. The backend distinguishes between invalid requests, authentication failures, authorization failures, missing resources, and internal processing or database errors through appropriate HTTP status codes. Errors are logged through the backend logging system, while responses sent to the frontend are kept sufficiently generic to avoid exposing internal implementation details. This provides a consistent interface for the frontend while keeping infrastructure-specific information within the backend.

6.4. Frontend Implementation

The ORVAL frontend is implemented as a Vue 3 single-page application using the Composition API. The interface is divided into reusable components that separate navigation, dashboard control, visualization, and administrative functionality. The main telemetry workspace is implemented in `Worksheet_Canvas.vue`, which manages the active dashboard and its associated widgets. Individual visualizations are implemented as specialized chart components, allowing different telemetry representations to be integrated without coupling their rendering logic to the main workspace.

Dashboard layouts are rendered dynamically using `vue-grid-layout-v3`. Each dashboard tab contains a grid in which widgets are defined by their position and dimensions. When a dashboard is rendered, the widget configuration is used to resolve the corresponding chart component dynamically, allowing different visualization types to be instantiated from the same dashboard structure. This allows operators to add, remove, reposition, and resize visualization components while keeping the underlying dashboard structure independent of the specific chart implementation. When dashboard configurations are saved, telemetry data itself is excluded from the stored layout. Only information required to reconstruct the dashboard, such as parameter identifiers, widget geometry, and display configuration, is persisted. Telemetry data is subsequently retrieved again when the dashboard is loaded.

Shared frontend state is managed through Pinia stores. The authentication store maintains the current user session and authentication token, while the dashboard store maintains the available telemetry parameter catalogue. The latter is loaded from the backend and retained in the client-side state to avoid repeatedly requesting the same parameter metadata. Authentication state is persisted in browser storage, allowing the application to restore a valid session after a page reload.

Telemetry visualization is handled by Apache ECharts through the specialized chart components. A chart instance is initialized when the component is mounted and updated reactively when new telemetry data becomes available. Instead of recreating the chart for every update, the existing ECharts instance is updated with the new series data. For numerical telemetry, the visualization can additionally represent the minimum and maximum values associated with the time interval as an envelope around the average signal. Charts can also be linked through a shared ECharts synchronization group, allowing interactions such as zooming, panning, and crosshair positioning to remain synchronized across multiple visualizations. Resize observers and component lifecycle hooks are used to adapt charts to changing widget dimensions and to dispose of chart instances when they are removed.

Live telemetry is implemented through periodic delta requests rather than repeatedly retrieving the complete time range. When live mode is enabled, the frontend collects the parameters displayed on the active worksheet and sends them in a single request to the backend at a configurable interval, with a default polling period of five seconds. The request includes a timestamp indicating the point from which new measurements should be retrieved. Newly received values are appended to the existing chart series, while older points outside the configured display window are removed to limit browser-side memory usage. The polling timer is explicitly cleared when the relevant component is unmounted.

Communication with the backend is centralized in `ORVAL_AI.js`. Its `smartFetch()` function provides a common interface for authenticated API requests and handles request timeouts, retries, and backend-target changes without requiring individual components to implement these mechanisms. For operations that modify application state, `broadcastFetch()` attempts dual-dispatch mutation across both backend endpoints to keep persistent configurations synchronized under nominal conditions. In degraded single-node operating modes, writes to the offline node are dropped without distributed transaction rollback, accepting temporary configuration divergence in favour of operational simplicity. The module also maintains the global heartbeat and exposes reactive network status information used by components such as the navigation bar and connection indicators. This centralization keeps network-related behaviour separate from the visualization components while providing the frontend with a consistent interface for backend communication.

6.5. Health Monitoring Implementation

ORVAL implements health monitoring at both the frontend service level and the backend database level. These mechanisms continuously provide information about the availability of the redundant components without directly initiating failover. This separation allows the application to distinguish between the observation of system health and the actions taken when an active request encounters a failure.

At the frontend level, health monitoring is centralized in `ORVAL_AI.js` through the `startGlobalHeartbeat()` function. The heartbeat executes every two seconds and probes both backend nodes independently and in parallel. Each probe requests the lightweight `/status` endpoint and is protected by a two-second timeout using an `AbortController`. The result of each probe is stored in the reactive `networkStatus` object, which maintains the reachability of both backend nodes and the database health information reported by each node. This allows the interface to distinguish between an unavailable backend service and a backend service that is operational but has a degraded database connection. The heartbeat therefore provides the frontend with a continuously updated view of the distributed system state.

Database health is monitored independently within the backend database layer. The `checkPoolHealth()` function executes every five seconds and also performs an initial check during backend startup. It tests the primary and secondary database pools separately using a lightweight `SELECT 1` query. A successful query indicates that the corresponding database connection can be established and used, while a failed query marks that target as unavailable. The resulting state is stored in the in-memory `poolHealth` object and can be accessed by other backend components without performing another database query. Checking both targets independently ensures that the failure of one database does not prevent the health state of the other from being updated.

The backend exposes this information through dedicated status endpoints. The lightweight `GET /status` endpoint is intended for frequent heartbeat requests and returns the latest database health snapshot from memory, avoiding additional database I/O for each frontend probe. A separate diagnostic endpoint provides more detailed system and database information, including database latency and process metrics. To prevent repeated diagnostic requests from unnecessarily increasing database traffic, live database metrics are subject to a five-second cooldown and are temporarily served from an in-memory cache.

An important implementation decision is the separation between health monitoring and failover actuation. The frontend heartbeat and backend database health loop only observe system state and update the corresponding status information. A failed heartbeat does not, by itself, change the active backend target. Failover is instead initiated by the request-handling mechanisms when an actual telemetry or database operation fails according to the defined failure criteria. This prevents transient failures of monitoring requests from unnecessarily triggering a system-wide transition while normal application requests may still be functioning correctly.

Through these mechanisms, ORVAL maintains separate and continuously updated health information for its backend services and database targets. The monitoring layer consequently provides the information required by the user interface and diagnostic functions, while the fault-tolerance mechanisms described in the following sections remain responsible for responding to confirmed failures.

6.6. Deterministic Failure Detection

Failure detection for frontend requests is centralized in the `smartFetch()` function of `ORVAL_AI.js`. This function acts as the common request gateway for communication with the backend and applies the same failure-detection procedure to each request. Individual requests are first executed through `tryFetch()`, which limits the duration of a standard request to five seconds using an `AbortController`. When the timeout is reached, the request is aborted, and the resulting error is handled in the same manner as a network failure. Explicitly identified long-running operations, such as large telemetry exports, can use an extended timeout of 60 seconds.

When a request to the active backend target fails, `smartFetch()` follows a fixed three-attempt sequence. This first attempt is executed immediately; if it fails, the function waits five seconds before performing the second attempt. A second failure results in another five-second delay before the third and final attempt. No additional delay is introduced after the third failure before the subsequent failover procedure is initiated.

Consequently, failure detection has a fixed maximum duration of 25 seconds when all three attempts consume the full five-second request timeout. When connection failures are detected immediately, only the two five-second inter-attempt delays contribute to the detection period, resulting in a duration of approximately 10 seconds.

The implementation applies the failure-classification rule defined in Section 5.7. Network errors, request timeouts, and HTTP5xx responses advance the three-strike sequence, while HTTP4xx responses are returned to the caller without triggering infrastructure failover. Authentication failures such as HTTP401 are handled separately by the authentication mechanism. This separation prevents valid application-level errors from being incorrectly interpreted as backend failures.

Retry state is maintained independently for each invocation of `smartFetch()`. This implementation does not use a global strike counter; instead, the sequential retry stages belong to the individual asynchronous function execution. Consequently, simultaneous requests from different dashboard components cannot combine their failures into a shared strike count. Each request must independently satisfy the failure condition before reaching the failover boundary.

Additional protection is provided for concurrent requests through a snapshot of the active backend target. When `smartFetch()` starts, it stores the current target in `initialUrl`. After each retry delay, the function compares this value with the current active target. If another request has already caused the system to switch targets, the comparison detects that `currentBaseUrl` has changed and the waiting request immediately uses the new target instead of continuing its remaining attempts against the failed backend. This mechanism does not require a central asynchronous lock while preventing trailing concurrent requests from independently repeating the same failure sequence. A separate logging flag also prevents simultaneous failures from producing duplicate failure notifications.

Together, the fixed attempt count, defined timeouts and delays, explicit failure classification, per-request retry state, and concurrent-request protection provide a deterministic mechanism for identifying backend failures. The subsequent change of the active backend target is handled separately by the failover implementation described in Section 6.7.

6.7. Autonomous Failover and Routing Implementation

Once the active backend has reached the defined failure threshold, ORVAL autonomously attempts to serve the pending request through the secondary backend. This failover mechanism is implemented in the `smartFetch()` function of `ORVAL_AI.js`, which acts as the central routing point for frontend requests. If the active target is Node Alpha and the three-attempt failure sequence has been exhausted, the same request is immediately re-issued to Node Beta. No additional delay is introduced at this stage, and the original request context, including its HTTP method, payload, query parameters, and authentication token, is preserved. The secondary attempt is subject to the same five-second request timeout as a standard backend request.

The routing state is only changed after the secondary backend responds without a connection failure, timeout, or server-side failure. Application-level HTTP errors are not treated as infrastructure failures, since they indicate the secondary backend is unreachable. This prevents the application from switching to an unverified backend target. Once the secondary response is accepted, the module-level `currentBaseUrl` variable is changed from the primary to the secondary backend. At the same time, the reactive `networkStatus` state is updated to indicate that the application is operating on the secondary node, the global connectivity status is restored, and a warning is displayed indicating that the primary backend is unavailable. Because `currentBaseUrl` is shared by the frontend service module, subsequent calls to `smartFetch()` automatically use Node Beta without requiring individual components to reconfigure their requests.

ORVAL also accounts for requests that are executing concurrently during a failover. Each invocation of `smartFetch()` stores the backend target that was active when the request started. Before continuing after a retry delay or initiating failover, the request compares this stored value with the current routing state. If another request has already changed `currentBaseUrl`, the waiting request abandons further attempts against the previous backend and immediately retries its operation using the new active target. This prevents multiple requests from independently repeating the complete failure sequence after a failover has already been established. In the event that multiple requests reach the failover boundary simultaneously, more than one secondary request may be initiated; assigning the secondary target to `currentBaseUrl` remains idempotent, and subsequent requests use the updated routing state.

If Node Beta also fails to serve the request, ORVAL does not change the active routing state to the unavailable target. Instead, the frontend is marked as disconnected, a critical system-unavailable status is exposed to the interface, and the error is propagated to the calling component. When the application is already operating on Node Beta, a subsequent failure results in the same blackout handling rather than an immediate switch back to Node Alpha. Recovery of the primary node is handled separately by the return-to-home mechanism described in Section 6.8.

Following a successful failover, ORVAL additionally initiates a non-blocking check of the database connection associated with Node Beta and starts the background recovery monitoring for the primary node. These operations do not delay the response to the original request. The implementation therefore combines autonomous target switching with explicit routing-state updates and protection against concurrent requests, allowing subsequent frontend requests to use the newly active backend without individual components requiring reconfiguration.

6.8. Autonomous Return-to-Home Implementation

After a successful failover to Node Beta, ORVAL continuously monitors the primary backend to determine when the primary backend is responsive again, and normal routing can be restored. This mechanism is implemented through the `startRecoveryCheck()` function in `ORVAL_AI.js`. The recovery watcher is started after a successful failover and uses a module-level interval reference to ensure that only one recovery watcher is active at a time.

Every 30 seconds, the watcher sends a direct `GET /status` request to Node Alpha. The request intentionally bypasses `smartFetch()` and `currentBaseUrl`, since the latter points to Node Beta while the system is operating in failover mode. A two-second `AbortController` timeout limits the duration of each recovery probe. If Alpha remains unreachable, the request times out or fails at the network level, or returns an HTTP status of 500 or higher, no routing change is performed. The watcher simply waits for the next polling interval while application traffic continues through Node Beta.

A recovery probe is evaluated as successful when Node Alpha responds with an HTTP status below 500. This check validates application process responsiveness on the primary node. In the current implementation, verification of the primary node's database connection pool is executed asynchronously after the routing switch is committed. While this ensures rapid failback execution, full operational validation prior to routing reassignment remains an architectural refinement for subsequent releases.

Before changing the active route, the recovery watcher checks the configured `ACTIVE_NODE_MODE`. Automatic recovery is only permitted when this setting is `AUTO`. If an administrator has explicitly forced the system to either the primary or secondary node, the watcher leaves the current routing state unchanged and continues monitoring. This allows manual administrative control to take precedence over autonomous recovery.

When Alpha responds successfully while the system is operating in `AUTO` mode, the watcher immediately changes `currentBaseUrl` back to the primary target. The reactive `networkStatus` state is updated accordingly: the active node is set to `PRIMARY`, connectivity is restored, and a recovery notification is displayed to the operator. The failure-log state is also reset so that future failures of Node Alpha can be reported normally. The recovery notification is automatically removed after four seconds.

After the routing transition has been completed, the recovery watcher is terminated using `clearInterval()`, and its timer reference is reset. This prevents unnecessary background requests once the primary node is active again. A guard at the start of `startRecoveryCheck()` also prevents multiple recovery intervals from being created when several concurrent operations trigger the recovery mechanism.

The resulting return-to-home process therefore provides autonomous recovery without requiring user intervention: ORVAL periodically probes the failed primary node, respects administrative overrides, restores the primary routing target once its service becomes responsive, updates the frontend state, and terminates the recovery watcher after the transition.

6.9. Logging and Observability

ORVAL includes a centralized logging and observability layer to make system behaviour, failures, and administrative actions visible during operation. Backend logging is implemented using Winston and `winston-daily-rotate-file` in `logger.js`. Rather than storing all events in a single log stream, five dedicated loggers are used for system, security, application, authentication, and dashboard events. This separation allows operational events to be filtered according to their purpose and simplifies troubleshooting of specific parts of the application.

Log entries are generated using a structured formatter that includes a timestamp, log level, service or source identifier, and additional metadata when available. Error entries can also contain stack traces to support troubleshooting. Log files are rotated daily and compressed into gzip archives. A configurable file-size limit and retention period prevent the logging system from continuously consuming disk space. The retention configuration can be adjusted through the application's system configuration.

Runtime events related to the redundant architecture are also recorded. The frontend uses the `logInternalEvent()` helper to report relevant network transitions, including failed connection attempts, confirmed failover to Node Beta, and successful recovery of Node Alpha. This helper deliberately uses native `fetch()` instead of `smartFetch()` so that logging does not enter the same retry and failover mechanism that it is intended to report. An optional destination allows incident messages to be directed to the currently reachable backend, ensuring that important failure events can still be recorded when the primary node is unavailable. Backend components such as the database layer and controllers write directly to the appropriate Winston logger.

Operational status is exposed through two complementary backend endpoints. The lightweight `GET /status` endpoint provides frequently requested health information from in-memory state and therefore does not require a new database query for each request. A separate diagnostic endpoint provides more detailed information about the backend process and database connectivity, including metrics such as uptime, memory usage, connection status, and database latency. Database diagnostics are protected by a short caching interval so that repeated requests do not unnecessarily increase database traffic.

This information is subsequently exposed to operators through the frontend. The shared `networkStatus` state provides information about the active backend, connectivity, database status, and current alerts. Administrative views use this information to display the state of the redundant infrastructure, including active, standby, and unavailable components, while infrastructure monitoring provides historical measurements of backend and database latency and process memory usage. A dedicated log viewer additionally allows operators to inspect recent events by log category.

Finally, ORVAL separates internal diagnostic information from externally returned error messages. Backend exceptions can be recorded with detailed error information and stack traces in the server logs, while API responses provide more generic error messages. This allows engineers to retain the information required for diagnosis without unnecessarily exposing internal implementation details to frontend clients. Together, the logging, diagnostic endpoints, and operator-facing monitoring provide the observability required to identify failures, understand system state, and support troubleshooting during operation.

6.10. Deployment Environment

ORVAL is packaged as a containerized application using Docker and Docker Compose. The production deployment consists of three containers: one frontend container and two backend containers representing the primary and secondary backend nodes. Docker Compose coordinates these services on a shared network, allowing the frontend and backend instances to communicate while keeping the redundant backend services as separate deployed units.

The frontend uses a multi-stage Docker build. During the first stage, the Vue application is compiled using Vite, and its production assets are generated. These assets are then copied into a lightweight Nginx container, which serves the compiled single-page application. The Nginx configuration also supports Vue Router by redirecting requests for client-side routes to the application's `index.html`. This allows operators to directly navigate to or refresh client-side application views without requiring a corresponding file for each route on the server.

The two backend instances are deployed from the same backend Docker image, ensuring that both nodes run the same application version. Their persistent application data is stored in separate host-mounted volumes, keeping configuration, dashboard data, user information, and log files isolated between the two instances. The telemetry database itself remains part of the existing infrastructure and is therefore not deployed as part of the production Compose environment.

Environment-specific configuration is separated from the compiled application where possible. In particular, the frontend uses a runtime configuration mechanism for the addresses of the redundant backend nodes. During the image build, symbolic placeholders are embedded into the generated JavaScript files. When the frontend container starts, `configureBackendURLs.sh` replaces these placeholders with the backend targets supplied through the deployment environment. This allows the same compiled frontend image to be deployed in different environments without rebuilding the application for each network configuration.

The production deployment therefore separates application packaging from environment-specific configuration while maintaining independent instances of the redundant backend. The resulting container structure provides a reproducible deployment environment in which the frontend can communicate directly with both backend nodes, while their persistent application state remains isolated.

6.11. Security and Operational Considerations

Security measures are integrated into ORVAL to restrict access to telemetry data and administrative functionality while supporting safe operation of the redundant environment. Authentication is implemented using JSON Web Tokens (JWTs). Users authenticate through the backend and receive a signed token containing their identity and assigned role. Subsequent requests to protected endpoints must provide a valid token, which is verified by authentication middleware before the request is passed to the relevant controller.

Authorization is implemented through role-based access control. ORVAL distinguishes between viewer, operator, and administrator roles, with increasing levels of access. Administrative endpoints are protected by dedicated middleware, while dashboard modification operations require the appropriate operator clearance. These checks are enforced on the backend, while frontend route guards additionally prevent unauthorized navigation within the user interface. The backend therefore remains responsible for the actual enforcement of access permissions regardless of the state of the frontend.

ORVAL also implements a global session revocation mechanism. Administrators can invalidate previously issued sessions through a dedicated administrative operation. Rather than maintaining a token blacklist, the backend records a global revocation timestamp and compares it with the issuance time contained in incoming tokens. Tokens issued before the revocation point are consequently rejected. When the frontend receives an authentication-related HTTP 401 response, the local session is cleared, and the user is redirected to the login page.

In the production deployment, sensitive values such as database credentials and the JWT signing secret are supplied through environment-based configuration rather than relying on deployment-specific values in the application source code. The production backend additionally runs under a non-root container user, reducing the privileges available to the application process.

Telemetry access is restricted to authenticated users, and database queries use parameterized inputs to prevent user-supplied values from being interpreted directly as SQL statements. Internal errors are recorded in server-side logs with detailed diagnostic information, while responses returned to clients use generic error messages to avoid unnecessarily exposing implementation or database details.

Finally, administrative operations that can affect system operation, such as node overrides and session revocation, require explicit confirmation in the administrative interface. Combined with role restrictions and security logging, these measures provide operational safeguards around the management functions of ORVAL without making security itself the primary focus of the framework.

7. Iterative Development & Design Evolution

The architecture and implementation presented in the previous chapters represent the final state of ORVAL. This final state was not established in a single design step, but emerged through successive iterations in which the system was extended and refined in response to technical requirements, domain knowledge, and feedback from the intended users. This iterative process was particularly important because ORVAL had to combine two distinct concerns: providing a flexible telemetry visualization environment and maintaining reliable access to telemetry data in the presence of infrastructure failures.

This chapter describes how the initial prototype evolved into the final ORVAL architecture. The focus is not on providing a chronological account of the development process, but on identifying the main changes in system capabilities and design decisions that shaped the final solution. The initial prototype establishes the technical baseline from which subsequent adaptations can be understood. The following sections then discuss the problems and limitations identified during development, the resulting design adaptations, and the lessons that emerged from the iterative process.

The evolution of ORVAL is therefore presented as a sequence of architectural and functional refinements rather than as a development diary. Validation of the resulting system is addressed separately in Chapter 8, where the final implementation is evaluated against the defined requirements and failure scenarios.

7.1. Initial Prototype

ORVAL was conceived as an in-house telemetry visualization framework tailored directly to Redwire's ground-segment testing workflows, addressing the latency and operational constraints of the incumbent infrastructure established in Chapter 2. Development commenced with the objective of establishing an extensible, dual-tier architecture capable of autonomous fault recovery.

From the outset, ORVAL was designed around a frontend-backend architecture. The frontend was responsible for the user-facing application and visualization environment, while the backend provided the application services and access to the existing telemetry database infrastructure. The databases themselves were not developed as part of ORVAL; they were existing infrastructure provided by Redwire. ORVAL therefore had to integrate with this infrastructure rather than redesign the underlying telemetry data platform.

An important characteristic of the initial development was that service redundancy was established at an early stage. Rather than first developing a completely single-node application and adding fault tolerance afterwards, the application was structured around two backend service paths, referred to as Alpha and Beta. Alpha represented the primary service path, while Beta provided a redundant path that could be used when Alpha became unavailable. This established the basis for autonomous service switching at an early stage of the system's development.

The first meaningful prototype consequently already demonstrated more than basic communication between the frontend and backend. The frontend could autonomously detect the unavailability of the primary backend and redirect subsequent requests to the redundant backend. When the primary backend became available again, the prototype could also restore the connection to Alpha. This provided an early demonstration of autonomous failover and recovery, even though the mechanisms supporting these behaviours were still considerably simpler than those of the final implementation.

The prototype also provided the first structure around which the later telemetry visualization environment was built. It included a home interface, a topology-oriented view for observing the state of the Alpha and Beta service paths, and a diagnostic interface providing operational information such as latency measurements. User authentication and basic management functionality were also established. These components provided a functional application environment in which the behaviour of the redundant architecture could be demonstrated independently of the more advanced dashboard functionality that was developed subsequently.

At this stage, however, the redundancy mechanisms were still limited in scope. Health monitoring and application-level state propagation were not yet fully coordinated with the routing mechanism, and operations that required communication with both service paths could therefore not yet be handled reliably in all situations. Administrative functionality was also still incomplete. Most importantly, the telemetry dashboard and visualization engine had not yet reached their final form. The prototype should therefore be regarded as an architectural and functional foundation rather than as a reduced version of the final system.

The initial prototype nevertheless established an important design direction for ORVAL. Although manual switching between the redundant service paths would have been possible, autonomous switching was considered a safer approach because it reduced the dependency on operator intervention during failures. This principle therefore became an important part of the subsequent development. Rather than introducing redundancy for the first time, the development focused on making the existing autonomous behaviour deterministic, reliable, consistent across concurrent requests, and integrated with the telemetry visualization requirements.

7.2. Domain Expert Feedback & Identified Problems

The initial prototype established the main architectural foundation of ORVAL, but further development revealed additional requirements and technical challenges that were not fully addressed by the first implementation. These emerged from two complementary sources: feedback from domain experts and intended users at Redwire, and problems identified through development and testing. The former primarily defined the operational expectations from the system, while the latter exposed technical constraints that had to be addressed to satisfy those expectations reliably.

From the perspective of the domain experts, the dashboard did not initially require highly advanced interaction capabilities. The essential functionality was considered to be the ability to add and remove telemetry parameters, and to move and resize visualizations within a dashboard. The architectural framework was designed to accommodate the operational expectation of 25 to 30 concurrent users through connection pooling and targeted SQL delta queries. Multi-user validation verified consistent routing across multiple simultaneous client sessions, while formal high-throughput stress benchmarking remains an objective for future qualification. This requirement introduced an important scalability consideration: the number of requests generated by the frontend could grow rapidly when multiple users simultaneously monitored telemetry.

The expected workload therefore led to a central database-access problem. A straightforward implementation could issue an independent database request for every parameter in every dashboard. Under this workload, this would result in a large number of repeated requests, particularly when different users were monitoring the same telemetry parameters. This created a need to retrieve the required parameters for a user's active dashboards efficiently, rather than issuing an independent database request for every parameter. The resulting approach grouped the required parameters into a single telemetry retrieval operation and requested only the distinct parameters required by that user. In addition, connection pooling was used to avoid repeatedly establishing database connections, while appropriate indexing improved the efficiency of the main telemetry access patterns. These changes shifted the database layer from simple request-based retrieval towards an access model designed to support sustained concurrent use.

Live telemetry introduced an additional scalability consideration. Unlike historical queries, live dashboards repeatedly request newly available measurements at a configurable interval. With the default polling interval of five seconds, a large number of simultaneously active dashboards could continuously generate database traffic. ORVAL was therefore designed to retrieve only data that became available since the previous poll rather than repeatedly retrieving an entire window. This reduced unnecessary data transfer and database processing while allowing multiple visualizations to be updated continuously.

The redundant architecture introduced a second major design challenge: maintaining a consistent service transition when a backend became unavailable. The initial failover behaviour demonstrated that a frontend could switch from Alpha to Beta autonomously, but multiple dashboard components could issue requests concurrently. If each request independently detected the failure and attempted to perform a service transition, different requests could make conflicting or redundant routing decisions. The routing mechanism therefore had to evolve towards a shared application-level routing state, allowing concurrent requests to recognize when another request had already established a new active service path. This became particularly important for transitions in both directions, from Alpha to Beta during failure and from Beta back to Alpha after recovery.

Development and testing also revealed correctness issues that were not directly related to service availability. One example concerned the representation of time. The timestamp displayed by ORVAL initially did not always correspond to the time reference used by the telemetry infrastructure because the application relied on local system time rather than the mission-relevant database time reference. As a result, a telemetry measurement associated with 14:00 in the mission-time reference could, for example, be displayed as 16:00 for a user operating in a UTC +2 timezone. Since telemetry analysis depends on accurate temporal alignment, the handling of timestamps had to be reconsidered so that telemetry remained associated with the intended time reference.

Another issue concerned the interaction between long-running telemetry queries and failure detection; the same network mechanism responsible for detecting an unavailable backend or database also had to handle legitimate requests that required substantially more processing time, particularly when retrieving large historical time ranges. Treating every timeout identically could therefore incorrectly classify a slow but functioning backend as unavailable and initiate an unnecessary failover. The request-handling mechanism was consequently refined to distinguish normal short requests from operations that legitimately require a longer execution window.

The telemetry visualization requirements also resulted in several refinements to the way data was retrieved and represented. Large historical time ranges could produce datasets too large to transfer and render efficiently in the browser. Reducing the number of returned points was therefore necessary, but simple averaging could obscure short-lived peaks in telemetry values. The resulting approach retained minimum and maximum values alongside the average trend, allowing the visualization to reduce the volume of data while preserving information about transient variations. Similarly, the processing of non-numerical telemetry required separate handling because numerical aggregation methods are not appropriate for categorical values.

In addition to the technical issues identified during development, feedback from domain experts influenced the organization and usability of the application. An important example was the introduction of the concept of decks: rather than treating each dashboard as an isolated configuration, multiple dashboards could be grouped and saved together. This better reflected the way operators could organize related monitoring views. Other advanced dashboard capabilities, such as synchronized graphs, automatic layout behaviour, and additional interaction mechanisms, were introduced as design improvements rather than as explicit initial requirements. They resulted from considering how the basic dashboard functionality could be made more efficient for realistic telemetry analysis.

Operational feedback also emphasized the need for clear separation between ordinary users and administrative functionality. Operators should not automatically have access to detailed system information such as internal logs or infrastructure statistics. This contributed to the separation of regular application functionality from administrative and diagnostic interfaces. Logging itself was expected to remain clear and usable rather than exposing unnecessary implementation detail to ordinary users.

Finally, a lightweight visual mode was requested after the prototype was demonstrated. The primary interface remained the more extensively designed dark theme, while the light mode focused on readability and functional clarity rather than reproducing every visual detail of the primary interface. This illustrates the distinction between requirements originating directly from users and additional refinements introduced as part of the design process.

These findings shaped the subsequent evolution of ORVAL in several directions. The system had to support higher levels of concurrent telemetry access, maintain consistent routing during autonomous service transitions, distinguish infrastructure failures from legitimate long-running operations, and provide telemetry representations that remained useful at different time scales. At the same time, domain feedback established practical expectations for dashboard interaction, organization, access control, and usability. The following section describes the resulting design adaptations and how these requirements and identified problems were incorporated into the architecture and implementation.

7.3. Design Adaptations

The initial prototype established the main architectural direction of ORVAL, but subsequent development focused on reducing the risk of incorrect routing decisions, improving efficiency, and ensuring that the redundant architecture remained consistent under changing conditions. The resulting adaptations were not primarily driven by failures of the overall architecture, but by identifying situations in which a seemingly functional solution could produce undesirable behaviour under realistic operating conditions. Several of these adaptations were therefore introduced proactively, with the aim of reducing the likelihood and impact of future failures.

7.3.1. Deterministic Failure Detection

A first important adaptation concerned the detection of backend and database failures. Although manual switching between redundant service paths would have been possible, relying on an operator to recognize a failure and initiate the appropriate transition introduces additional reaction time and creates an avoidable dependency on human intervention. From the beginning of the fault-tolerance design, autonomous detection was therefore preferred as a safer approach.

As established during architectural design, manual failover introduces unacceptable reaction latency during test monitoring. Prototype evaluation across repeated test executions confirmed that the deterministic three-attempt retry policy successfully suppressed transient network jitter without triggering premature route invalidation, with observed failover timings consistently remaining within the expected window. Engineering reviews with Redwire mentors confirmed that bounding detection between 10 and 25 seconds satisfied operational monitoring safety margins, establishing the mechanism as a fixed architectural standard for all subsequent iterations.

7.3.2. Coordinated Failover Under Concurrent Requests

The introduction of autonomous failover created a second architectural consideration: a dashboard does not generate only one request at a time. Multiple visualization components and background operations may be active concurrently. Consequently, when Alpha becomes unavailable, several requests may encounter the failure at approximately the same time.

A naïve implementation could allow each request to independently conclude that Alpha had failed and initiate its own failover decision. This would create redundant transitions and could result in different requests temporarily operating according to different routing decisions. The problem was therefore not detecting the failure itself, but ensuring that the result of a single failover decision was respected by other requests already in progress.

The routing mechanism was adapted to maintain shared application-level routing state. Requests retain knowledge of the service path they initially targeted, but re-evaluate the current routing state before performing a new failover transition. When another request has already changed the active path from Alpha to Beta, subsequent requests recognize that transition and follow the newly established route instead of independently initiating another transition.

This adaptation allowed failover to remain autonomous without requiring a separate coordination service. It also established the routing state as a central decision point for the frontend, rather than allowing individual dashboard components to make independent service-selection decisions.

7.3.3. Database Performance Optimisation

Database access was another area in which the initial functionality revealed opportunities for improvement. During development, telemetry queries over large time ranges could take considerably longer when the database lacked the indexes required by the main access patterns. Since telemetry visualization is inherently data-intensive, inefficient retrieval would directly affect the responsiveness of the application.

The database access was therefore optimized through appropriate indexing and connection pooling. Indexes were introduced for the main telemetry lookup patterns, reducing the amount of data that the database had to inspect for common queries. Connection pooling was introduced as a further efficiency measure so that database connections could be reused rather than repeatedly established for individual operations.

These changes were not intended to reduce the amount or quality of telemetry returned to the user. Instead, they addressed the same functional requirements through a more efficient execution path. This reflects an important design principle applied throughout ORVAL: when an operation can be made substantially faster without compromising the completeness or correctness of the resulting data, the more efficient approach should be preferred.

7.3.4. Separating Failure Detection from Long-Running Queries

The deterministic failure-detection mechanism introduced a new edge case for operations that legitimately require more processing time. The standard request timeout was intentionally short so that an unresponsive service could be detected within a bounded period. However, applying the same limit to computationally heavier telemetry operations could cause a valid request to be interpreted as an infrastructure failure.

This situation was encountered during testing of longer-running queries. When such a query exceeded the normal response-time limit, it produced a failure indication and could therefore contribute to the three-attempt failure sequence. Repeating the same legitimate query three times would not resolve the underlying issue; it could instead cause a healthy backend or database to be incorrectly classified as unavailable.

The request layer was consequently adapted to distinguish between ordinary requests and operations expected to require more processing time. Long-running telemetry operations use an extended timeout, while normal requests retain the shorter timeout required for deterministic failure detection. This separates two fundamentally different conditions: an infrastructure component that is not responding and an infrastructure component that is responding slowly because it is performing a legitimate operation.

The adaptation therefore preserved the deterministic nature of failure detection without forcing all database operations to conform to a timeout intended for lightweight service requests.

7.3.5. Correcting Telemetry Time Representation

A separate correctness issue emerged in the representation of telemetry timestamps. Telemetry timestamps use a reference associated with mission time, whereas the initial implementation interpreted timestamps relative to the user's local system time. As a result, a telemetry measurement generated at 14:00 in the mission-time reference could, for example, be displayed as 16:00 for a user operating in a UTC +2 timezone.

Although the underlying measurement was correct, its displayed temporal position was therefore incorrect. This could affect both the interpretation of historical data and the temporal relationship between measurements displayed on a dashboard.

The time-handling logic was adapted so that telemetry timestamps remain associated with their intended reference rather than being implicitly shifted to the user's local timezone. This ensured that the time represented by a telemetry measurement remained consistent between the underlying data and its visualization.

The change was important because telemetry visualization is not only concerned with retrieving the correct value, but also with placing that value at the correct point in time. Correct temporal representation was therefore treated as a data-correctness requirement rather than as a purely visual concern.

7.3.6. Autonomous Return-to-Home

The final major adaptation concerned the behaviour after a successful failover. Switching all requests from Alpha to Beta solved the immediate availability problem, but it introduced a new consistency issue once Alpha recovered.

Initial prototype testing exposed an operational hazard: remaining permanently on Node Beta after failover resulted in route divergence when new operators initialized sessions against the primary configuration. Implementing autonomous return-to-home resolved this divergence by establishing a closed-loop recovery lifecycle. Continuous background probing of Node Alpha, combined with an administrative override guard, ensured that client sessions converged back to the primary service path without operator intervention, closing the fault-tolerance loop.

Together, these adaptations refined the original ORVAL architecture without fundamentally changing its overall structure. The main architectural principles of redundancy and autonomous operation were retained, while the mechanisms governing failure detection, concurrent routing, database access, request execution, temporal correctness, and recovery were progressively strengthened. The next section describes the final iteration of this development process and the resulting state of the system.

7.4. Final Iteration

The adaptations described in the previous section progressively transformed the initial ORVAL prototype into a complete telemetry visualization framework. Importantly, this evolution did not require a fundamental change to the original system architecture. The core structure remained a frontend connected to two redundant backend service paths, with each backend providing access to its primary and redundant database targets. Instead, the development process focused on strengthening and optimizing this architecture so that its individual components operated together reliably under realistic conditions.

The difference between the initial prototype and the final system was therefore primarily one of maturity, integration, and operational behaviour. The prototype demonstrated that the basic concept of redundant telemetry access could work and already provided the foundation for autonomous switching and visualization. The final iteration extended this foundation with deterministic failure detection, coordinated routing during concurrent requests, optimized database access, appropriate handling of long-running operations, correct telemetry time representation, and autonomous return-to-home behaviour. At the same time, the dashboard engine evolved from a basic visualization environment into a flexible system supporting multiple configurable dashboards, live and historical telemetry, dynamic visualization components, and dashboard organization through decks. Administrative and monitoring functionality was also completed and integrated with the same architecture.

A central characteristic of the final iteration was therefore that fault tolerance was no longer an isolated redundancy mechanism. Failure detection, routing, health monitoring, recovery, and database redundancy were integrated with the normal telemetry visualization workflow. A backend failure could be handled autonomously without requiring individual dashboard components or users to manually reconfigure their connection. Similarly, recovery of the primary service path could be detected, and the application could return to the intended primary route. This made redundancy an intrinsic property of the application rather than an operational procedure performed outside the system.

The final ORVAL stack retained a deliberate and relatively simple structure: one frontend, two backend service paths, and two database targets per backend. This structure provided the required redundancy while avoiding unnecessary architectural complexity. The design is extensible, but adding additional service paths would require corresponding changes to the routing and failover logic. The final architecture was therefore optimized for the operational requirements of ORVAL rather than designed as a generic distributed infrastructure platform.

Following completion of the main development work, ORVAL was deployed on the ground-segment computers used by the engineering team. The system was subsequently demonstrated during an official presentation in which its architecture, autonomous redundancy behaviour, and visualization capabilities were presented to interested stakeholders. Feedback from this demonstration resulted in a limited number of further refinements, primarily concerning presentation and visual aspects of the application.

At the end of the development period, ORVAL had therefore progressed from a redundancy-oriented prototype capable of producing telemetry visualizations into a fully operational visualization framework with integrated autonomous fault tolerance. Although further improvements to aspects such as visual design or additional functionality remain possible, the system had reached a stable operational state and was actively used by engineers for satellite testing. Furthermore, its intended role extends beyond testing, with the framework planned as a telemetry visualization tool for future satellite launch operations.

This final iteration thus represents not a replacement of the original architecture, but its maturation. The fundamental structure established at the beginning of development remained intact, while successive adaptations made its behaviour more deterministic, efficient, consistent, and resilient. The resulting system provided the basis for the validation of ORVAL against the defined requirements and failure scenarios, which is addressed in Chapter 8.

7.5. Lessons Learned

The iterative development of ORVAL resulted in several broader engineering lessons that influenced the way the system was designed and refined. One of the most important lessons was the value of substantial reasoning before implementation. At the beginning of a software project, the combination of requirements, technical constraints, and possible solutions can create a seemingly complex problem space. During the development of ORVAL, it became clear that starting to write code immediately is not necessarily the most effective approach. Exploring possible architectures, identifying alternative solutions, and considering their consequences before implementation provided a clearer foundation for subsequent development. This initial investment in system-level thinking contributed to a more deliberate and optimized final design.

A second lesson concerned the importance of designing for failure rather than assuming that failure will be an exceptional event. In a system that depends on multiple infrastructure components, any component can become unavailable, respond slowly, or behave differently than expected. The development of ORVAL therefore reinforced the principle that potential failure should be considered an inherent part of the design problem. Rather than assuming that a particular failure scenario will not occur, the system should be designed with the expectation that any component that can fail should be considered a potential failure point. This mindset influenced the attention given to failure detection, bounded retries, redundant service paths, recovery behaviour, and concurrent requests.

The development process also reinforced the importance of efficiency as an engineering objective. Database access, telemetry retrieval, and live visualization can generate substantial amounts of activity, making inefficient operations increasingly problematic as usage grows. The use of connection pooling, indexing, incremental telemetry retrieval, and data reduction was therefore not treated merely as a response to observed performance problems, but as part of designing the system responsibly from the outset. At the same time, optimization was considered subordinate to correctness: reducing processing or transfer should not result in the loss of relevant telemetry information or a degradation of the visualization.

Another important lesson was that requirements alone cannot describe every desirable property of a system. The requirements provided the necessary foundation for ORVAL, but several improvements emerged from considering how the system could be made more robust, efficient, and operationally complete. This included mechanisms such as deterministic autonomous failover, coordinated routing during concurrent requests, and return-to-home behaviour. These additions illustrate the value of looking beyond the literal requirements and considering the broader consequences of design decisions. Feedback from domain experts remained important throughout this process, but engineering judgement was equally necessary to identify situations that had not been explicitly specified.

Finally, developing an actual application for operational use provided a stronger understanding of the difference between building software and building software that can be used reliably by others. Users interact with the application without knowledge of its internal architecture, implementation, or limitations. Consequently, the system cannot rely on users understanding how its underlying mechanisms work. This increases the importance of clear interfaces, predictable behaviour, sensible defaults, and handling of edge cases that users may never anticipate themselves.

Overall, the iterative development of ORVAL demonstrated that effective software engineering involves more than implementing predefined functionality. It requires anticipating failure, considering efficiency and edge cases, challenging assumptions, and continuously evaluating how technical decisions affect both the system and its users. These lessons shaped not only the final implementation of ORVAL, but also the engineering approach used throughout its development.

8. Validation & Results

The preceding chapters described the requirements, architecture, implementation, and iterative development of ORVAL. These chapters establish how the framework was designed to provide reliable telemetry access and how its fault-tolerance mechanisms were realized. This chapter evaluates whether the resulting system behaves as intended under both normal and failure conditions.

Validation focused primarily on the properties that distinguish ORVAL from a conventional telemetry visualization application: autonomous failure detection, deterministic failover, service recovery, return-to-home behaviour, and consistent routing during service transitions. These behaviours were evaluated by introducing controlled failures in the redundant backend and database infrastructure while ORVAL was operating in its intended ground-segment deployment environment. Normal telemetry visualization and multi-user operation were also validated to ensure that fault-tolerance mechanisms did not compromise the primary purpose of the framework.

The validation was performed on the ground-segment computers for which ORVAL was intended, using the production containerized deployment rather than an isolated development setup. Where representative satellite telemetry was available, real telemetry data was used; simulated data was used where specific behaviour could not practically be reproduced with available telemetry. Failure scenarios were repeatedly executed throughout development and system integration to verify that the resulting behaviour was consistent rather than dependent on a single successful execution.

The results are presented in terms of observed system behaviour, timing characteristics and requirement satisfaction. Particular attention is given to the transition between redundant service paths, since this is the point at which the reliability mechanisms of ORVAL are most directly exercised. The chapter therefore does not evaluate fault tolerance as an abstract architectural property, but as an observable characteristic of the implemented system.

8.1. Validation Methodology

8.1.1. Validation Environment

Validation was conducted in the intended ground-segment deployment environment. The production build of ORVAL was deployed using Docker on two Linux-based ground-segment computers. The deployment consisted of a frontend service and two redundant backend service paths, referred to throughout this thesis as Alpha and Beta. Each backend was associated with its corresponding database targets, providing redundancy at both the backend and database level.

Users accessed ORVAL from workstations connected to the ground-segment network. Consequently, the validation exercised the same application topology and networked deployment model for which the framework was designed, rather than testing the fault-tolerance mechanisms only within a local development environment. The spacecraft itself was not part of the validation environment; the validation concerned the ground-segment telemetry visualization system responsible for accessing and presenting telemetry data.

The containers were normally started together to represent the nominal operational state. Controlled failures were then introduced by stopping relevant Docker containers, allowing the behaviour of the remaining system components to be observed. After a failure scenario, the affected service could be restarted to evaluate recovery and return-to-home behaviour.

8.1.2. Test Setup and Scenarios

Validation consisted of repeated controlled functional tests covering both nominal operation and the main failure modes identified during requirements analysis. The principal scenarios were:

Table 12: End-to-end system validation scenarios and verification objectives across nominal and degraded states.

Scenario	Description & Verified behaviour
Normal operation	Telemetry retrieval, historical visualization, live telemetry updates, and regular application use were verified while the primary service path was operational.
Primary backend failure	The Alpha backend was made unavailable while Beta remained operational. The resulting failure detection, retry behaviour, and automatic transition to Beta were observed.
Primary backend recovery	Alpha was restored after failover. Recovery detection and automatic return-to-home were subsequently verified.
Database failure	Database targets were made unavailable to verify the redundancy implemented at the database level.
Multiple-user operation	Requests from multiple active users were generated during normal operation and service transitions to verify that routing state remained consistent within an ORVAL application instance.

Repeated failures

Relevant failure and recovery scenarios were executed repeatedly throughout development and integration to assess whether the observed behaviour remained consistent.

Administrative operation

Health information and manual routing controls were exercised to verify that automatic behaviour could be monitored and overridden by authorized administrators.

These scenarios were not treated as isolated unit tests. Instead, they exercised the complete deployed system across the frontend, backend, database, routing, and recovery mechanisms. This was particularly important for fault-tolerance validation because a successful failover depends on the interaction between several components rather than on a single function operating correctly in isolation.

8.1.3. Controlled Failure Conditions

Failures were introduced deliberately and in a controlled manner. For backend validation, the relevant Docker service was stopped while the remainder of the redundant deployment remained operational. Database failure scenarios were performed similarly by making the relevant database target unavailable. Recovery was tested by restoring the affected service and observing whether the framework detected its availability and subsequently restored the preferred primary path.

This approach allowed the failure condition to be known in advance and provided a clear reference point for evaluating the resulting system behaviour. It also ensured that the validation represented realistic component-level failures without requiring destructive changes to the underlying telemetry infrastructure.

The failure-detection mechanism itself was evaluated according to the deterministic parameters described in Chapter 6. A failed request can be attempted up to three times, with a five-second interval between attempts. Under the default five-second request timeout, a failure in which every attempt reaches the timeout therefore has a bounded detection window of up to approximately 25 seconds. Immediate connection failures result in a shorter window because the timeout does not need to elapse for each attempt. These values are configuration-derived bounds rather than measured averages and were used as reference points when interpreting observed behaviour.

8.1.4. Metrics and Observations

In accordance with standard dependability and fault-tolerance evaluation principles (such as the detection completeness and accuracy criteria established by Chandra & Toueg (1996) and Avizienis et al. (2004) discussed in Chapter 3), validation focused on operational metrics directly mapped to the system requirements defined in Chapter 4. To evaluate both failure detection correctness and recovery convergence empirically, the principal observations were:

- ⦿ Whether telemetry remained accessible during a component failure when a redundant path was available;
- ⦿ Whether a single unsuccessful request was correctly distinguished from a confirmed service failure;
- ⦿ The time required to detect a confirmed backend failure;
- ⦿ The transition time from the primary to the redundant backend;
- ⦿ Whether subsequent requests were routed through the new active path;
- ⦿ Whether recovery of the primary backend was detected;
- ⦿ Whether the system autonomously returned to the primary path;
- ⦿ Whether routing remained consistent for concurrent users during transitions;
- ⦿ Whether telemetry visualization continued without an observable interruption when a redundant path was available;
- ⦿ Whether the system correctly indicated a complete loss of backend availability when no operational path remained.

Where timing behaviour was determined directly by configured thresholds, these configuration values were treated separately from empirical observations. This distinction prevents deterministic implementation parameters from being presented as experimentally measured performance statistics.

8.1.5. Test Repetition and Acceptance Criteria

Testing was conducted iteratively throughout the development, integration, and operational deployment of ORVAL at Redwire. The validation protocol utilized a verification suite where the backend failover scenario was executed repeatedly – approximately 20 executions – in the ground-segment environment. Across these repeated manual executions, the mechanism demonstrated consistent operational timing, with observed failover durations exhibiting approximately one second or less variation between runs. Because the primary verification objective was to establish functional stability, deterministic boundary adherence, and reproducible state convergence rather than conducting a formal statistical sampling campaign, individual per-run timestamps were not officially retained across all executions. Instead, a fully instrumented end-to-end execution run was logged with high-resolution timestamps to provide concrete empirical evidence of the execution sequence within the theoretical bounds.

A test was considered successful when the observed behaviour satisfied the corresponding requirement and did not introduce an unintended loss of telemetry access or inconsistent routing state. For the fault-tolerance scenarios, this meant that an operational redundant path had to preserve telemetry access, failover had to occur only after the configured failure condition was reached, and automatic routing had to transition without manual intervention when automatic mode was enabled. For recovery scenarios, the primary path had to be functionally available before it was restored as the active path.

For multi-user scenarios, success required requests issued during a transition to respect the shared routing state of the ORVAL application instance, preventing newly processed requests from unintentionally reverting to an outdated service path. For visualization continuity, the expected result was that existing visualization state would be preserved and telemetry visualization would resume automatically once the redundant service path became active, without requiring a page reload.

The acceptance criteria therefore followed directly from the requirements established in Chapter 4 rather than from arbitrary performance thresholds. This provides a direct connection between the system's intended behaviour and the validation results presented in the remainder of this chapter.

8.2. Normal Operation

Before evaluating fault scenarios, ORVAL was validated under nominal operating conditions to establish that its primary function, reliable telemetry visualization, was fully operational without any service failures. During the tests, the system operated through Node Alpha as the active backend path, while Node Beta remained available as the redundant path and was continuously monitored in the background.

Historical telemetry was successfully retrieved and visualized for different time ranges and across multiple parameters simultaneously. Users could interact with the resulting visualizations

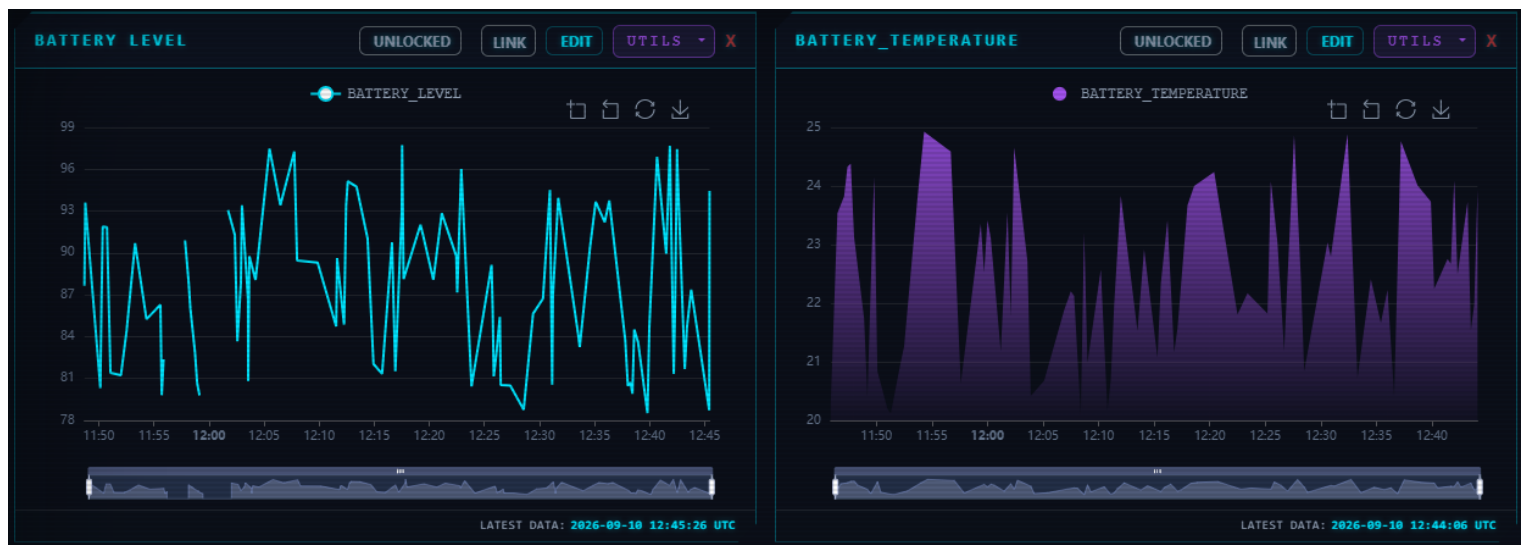


Figure 8: ORVAL dashboard displaying multiple telemetry visualizations during nominal operation.

as intended, including adjusting time ranges and manipulating dashboard layouts. The observed retrieval and rendering behaviour was responsive, with no noticeable delays during normal interaction. This established the baseline required for evaluating whether the later fault-tolerance mechanisms could preserve this functionality during service transitions.

Live telemetry was also validated using real telemetry data. After enabling live mode and selecting the required time window, the visualizations updated automatically at the configured five-second polling interval without requiring a page refresh or other user intervention. Rather than repeatedly reloading the visualization, the visible time window shifted forward as new measurements became available. The updates remained smooth and responsive throughout the tests, demonstrating that continuous telemetry visualization could operate autonomously under the nominal workload.

Normal user workflows were additionally exercised across multiple sessions; dashboards and decks were created, edited, saved, closed, and subsequently reloaded, including tests involving the same and different users. Visualization manipulation and telemetry export were also successfully performed. These tests confirmed that the visualization framework remained functional not only at the level of individual charts, but throughout the complete user workflow.

Concurrent-user testing further showed no noticeable degradation in responsiveness under the tested workload. During nominal operation, requests continued to use Alpha as intended, while Beta remained operational and available for failover. No unnecessary transition to the redundant path occurred while the primary service was healthy.

Overall, the nominal validation established a functional baseline for ORVAL: historical and live telemetry could be visualized continuously, normal user workflows remained responsive, and the primary service path operated as intended. The subsequent validation therefore focuses on the more demanding condition for which ORVAL was specifically designed: the controlled loss of an active service.

8.3. Backend Failure & Autonomous Failover

The main fault-tolerance scenario was validated by deliberately making the active Node Alpha backend unavailable while ORVAL was operating normally. This was achieved by stopping the Docker container hosting Node Alpha, allowing the complete sequence from backend failure to autonomous failover to be evaluated under controlled conditions.

Following the induced failure, ORVAL detected the loss of connectivity using its configured three-attempt failure detection mechanism. The validation logs recorded the first failed connection attempt at 13:08:20.719 and confirmed that Node Alpha was unreachable after three strikes at 13:08:32.954. This corresponds to an observed failure-detection time of approximately 12.2 seconds in this logged run. While this specific value represents a single instrumented trace rather than a calculated statistical average, repeated manual trials in the ground-segment environment consistently produced detection times in this same range, with observed variation of approximately one second or less. This implementation provides a theoretical detection window of 10-25 seconds depending on whether socket disconnections are immediate or consume the full five-second request timeout.

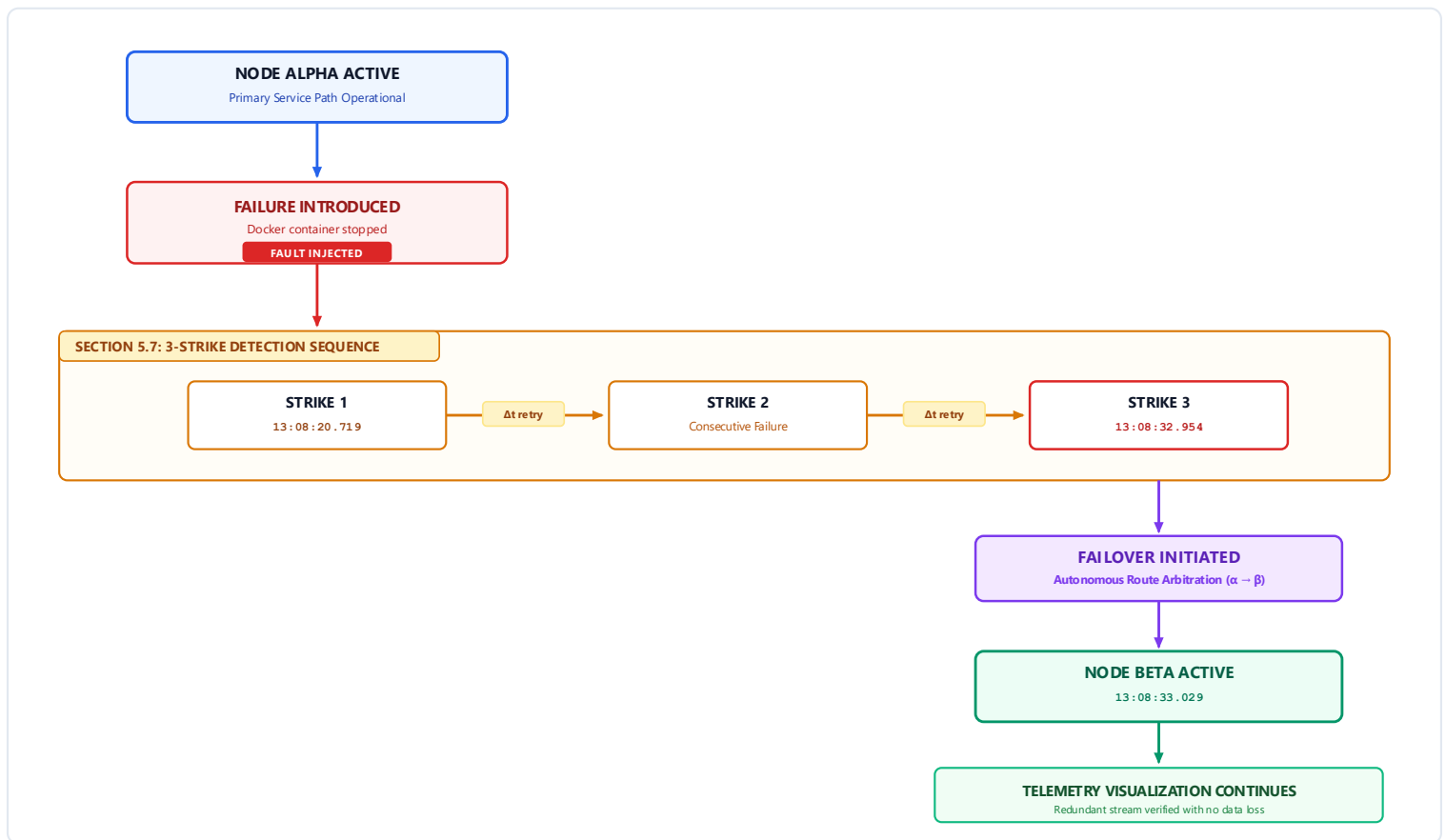


Figure 9: Observed Alpha-to-Beta failover sequence during controlled backend failure testing.

Once the failure condition was confirmed, ORVAL automatically initiated the connection to Node Beta without requiring any user interaction. The logs recorded the failover initiation immediately after the three-attempt threshold was reached, followed by confirmation that Node Beta was actively handling traffic at 13:08:33.029. From the first recorded failed attempt to confirmed operation on the redundant backend, the complete transition therefore took approximately 12.3 seconds.

From the user's perspective, the transition preserved workspace continuity without data corruption or session loss. Historical visualizations remained displayed without interruption, while live visualizations paused parameter ingestion for the duration of the 12.2-second detection window before resuming updates once the connection to Node Beta was established. No page refresh or re-authentication was required. Requests that had not yet completed were subsequently routed through Node Beta, while the frontend informed the user that Alpha was unavailable and that operations were continuing through the redundant backend. The system topology was also updated to reflect the active Node Beta path.

The failure scenario was executed repeatedly – approximately 20 executions – during development and ground-segment testing. Each execution successfully transitioned from Node Alpha to Node Beta without manual intervention, exhibiting no failover aborts or route lockups across the trials. Furthermore, the elapsed transition times remained stable across runs, consistently falling within the deterministic window with approximately one second or less observed timing variation. These repeated observations confirm that the autonomous failover behaviour is reproducible and dependable under the tested operational conditions.

8.4. Backend Recovery & Return-to-Home

After validating the failover to Node Beta, the recovery behaviour was evaluated by restoring Node Alpha while ORVAL was operating on the redundant backend. The stopped Alpha Docker container was restarted and, once available again, ORVAL continued monitoring the primary backend through its background recovery mechanism. The time between restarting the container and the recorded recovery event was not used as a recovery-time measurement, as the restart was intentionally performed after an uncontrolled waiting period.

Recovery was detected automatically through a lightweight health probe. ORVAL did not switch back merely because the Alpha service became reachable; the response also had to satisfy the application's health criteria. If Alpha could be reached but did not provide an acceptable response, the active route remained on Beta. Once a valid response was received, ORVAL automatically restored Alpha as the active backend. This behaviour was repeated during validation and consistently resulted in the expected return to the primary service.

From the user's perspective, the return-to-home transition was transparent. The frontend automatically updated to indicate that Alpha had become active again, while existing telemetry visualizations continued operating without a page refresh or re-authentication. Historical visualizations remained available, and live telemetry resumed smoothly through Alpha. No dashboard, deck, or visualization state had to be recreated.

Following the transition, Node Beta returned to its redundant standby role and remained available for a subsequent Alpha failure. This allowed the same failover and recovery cycle to be repeated without manually reconfiguring the system. Automatic return-to-home was active for all validation scenarios unless an administrator explicitly selected manual routing mode, in which case the configured routing decision was respected.

The recovery validation therefore demonstrated the complete autonomous cycle beyond initial failover: once the primary backend became functionally available again, ORVAL detected the recovery and restored the preferred service path without interrupting the user's session or requiring manual intervention.

8.5. Multiple-User Consistency

Multi-user validation was performed with multiple users connected to the same ORVAL environment, with both simultaneous and non-simultaneous requests being generated during normal operation and service transitions. While Node Alpha was available, all connected users were routed through Alpha, as the frontend maintains a single active backend path at a time. Normal operation remained responsive under the tested workload, with no progressive degradation of the ORVAL service stack observed during concurrent use.

The consistency of routing was then evaluated by introducing an Alpha failure while multiple users were actively connected. The tested clients converged on Node Beta, and each frontend reflected the updated active routing state. No persistent routing inconsistency was observed between the tested clients, and no page refresh or re-authentication was required. Existing dashboards and visualizations remained available, while the system topology was updated to reflect the active redundant backend.

Requests already in progress were also considered during the transition. Once a request triggered the failover and the active routing state changed to Beta, other unanswered requests detected the updated state and were redirected to the new active backend rather than independently continuing the failed Alpha transition. New requests issued after the switch were sent directly to Beta. This prevented concurrent requests from creating conflicting routing states or repeated failover transitions.

After the transition, all users were able to continue working normally through Node Beta, including loading and interacting with dashboards and telemetry visualizations. The same behavior was observed during recovery: once Node Alpha became available and passed the recovery check, all subsequent routing was automatically restored to Alpha without requiring user intervention.

The multi-user tests therefore demonstrated consistent routing behaviour across the complete service transition. Within each ORVAL application instance, concurrent requests followed the same active routing state, while the tested clients successfully converged on the same redundant service during backend failure and returned to the primary service after recovery.

8.6. Repeated Failure Scenarios

The main failure and recovery scenarios were repeated throughout development and ground-segment validation to assess whether the implemented behaviour was reproducible across multiple executions. Testing was performed manually by stopping and restarting the relevant Docker containers and by manipulating the routing configuration. Backend failover was executed approximately 20 executions, consistently completing within the expected timing envelope with approximately one second or less variation between runs. Since the validation was conducted as an iterative engineering verification rather than a formal statistical benchmarking campaign, individual timestamps were not logged for every historical execution, and an exact total sample size was not formally compiled.

Table 13: Summary of the repeated scenarios and their observed outcomes.

Failure/recovery scenario	Repeated	Observed outcome	Consistent across runs
Primary backend failure	Yes	Automatic failover from Alpha to Beta without user intervention	Yes
Primary backend recovery	Yes	Automatic return from Beta to Alpha in automatic mode	Yes
Repeated backend failure-recovery cycles	Yes	Subsequent Alpha failures could again trigger failover to Beta after recovery	Yes
Simultaneous backend failure	Yes	ORVAL notified the user that both backend nodes were offline and that a complete backend failure had occurred	Yes
Primary database failure	Yes	Database access was transferred to the redundant database path while the backend remained operational	Yes

Repeated database failure-recovery cycles	Yes	Database-tier failover successfully transfers query execution to the secondary database connection pool upon primary target unresponsiveness, maintaining query execution capability within the application layer without independently verifying underlying data store synchronization	Yes
Simultaneous database failure	Yes	Both database paths were made unavailable to verify the system response when no operational database path remained	Yes
Manual routing mode	Yes	Automatic return-to-home was suppressed while manual routing was active	Yes
Concurrent requests during backend failure	Yes	Pending requests followed the updated routing state without duplicate failover transitions	Yes

During primary database failure scenarios, query execution transferred to the redundant database pool without application-tier restart. Because the evaluation databases were statically seeded with identical telemetry records prior to fault injection, the visualization interface exhibited uninterrupted time-series displays. In operational ground-segment environments, telemetry completeness across database failovers depends directly on underlying database replication latency, which operates outside ORVAL's architectural boundary.

Repeated tests showed consistent behaviour across the evaluated scenarios. In particular, backend failover and recovery could be performed repeatedly, including consecutive failure-recovery cycles, without requiring a reset of the application state. The same principle was observed for database failures. Testing both backend instances and database paths simultaneously additionally verified the system response when no redundant service path remained available.

Overall, the repeated validation supports the reproducibility of the deterministic failure-handling mechanisms. No inconsistent failover or recovery behaviour was observed during the final validation scenarios.

8.7. Performance and Timing Analysis

The validation campaign was designed to verify deterministic functional compliance rather than to conduct an exhaustive statistical benchmark campaign. Consequently, two distinct categories of timing data are reported: configured deterministic bounds, derived mathematically from the retry thresholds and network timeouts; and empirical trace measurements, obtained from a representative controlled failure execution logged in the ground-segment environment. These trace measurements serve as an illustrative verification of the configured bounds rather than an experimental population mean.

Table 14: Summary of configured and observed timing metrics across failover and recovery operations.

Metric	Result	Evidence type
Backend failure detection	10-25 s configured range; ~12.2 s documented trace (broadly consistent across repeated trials, $\sim \leq 1$ s variation)	Configured bound + instrumented trace + repeated observation
Failover to redundant backend	~12.3 s in a documented run (consistent across repeated trials, $\sim \leq 1$ s variation)	Observed measurement + instrumented trace + repeated observation
Recovery detection	0-30 s after Alpha becomes responsive	Configured polling bound
Recovery probe timeout	2 s	Configured value
Frontend heartbeat	2 s interval, 2 s timeout	Configured value
Routing transition	Immediately after routing state change	Implementation + qualitative observation
Live telemetry polling	5 s default interval	Configured value
CPU/RAM/resource usage	Not systematically measured	Not available

To structure the validation evidence and prevent configured bounds from being conflated with operational observations, Table 15 delineates the different evidentiary levels supporting the failover mechanism. It contextualizes the derived theoretical window alongside repeated qualitative executions and the logged, high-resolution execution trace.

Table 15: Evidence classification and empirical observation profile for backend failover testing.

Verification dimension	Evidence classification	Characterization/observed outcome
<i>Theoretical detection bound</i>	Derived from configuration parameters	Bounded by the configured retry thresholds to between 10 and 25 seconds based on the retry thresholds and timeouts ($2 \cdot \Delta t_{\text{retry}} \leq T_{\text{detect}} \leq 3 \cdot t_{\text{timeout}} + 2 \cdot \Delta t_{\text{retry}}$)
<i>Operational test volume</i>	Repeated manual verification	Executed repeatedly, approximately 20 times, in the ground-segment environment across iterative development and testing cycles
<i>Functional reproducibility</i>	Repeated qualitative observation	Consistent, autonomous transitions from Node Alpha to Node Beta with zero observed route lockups or unhandled failover aborts
<i>Temporal consistency</i>	Operational observation	Observed transition durations remained tightly clustered across runs, exhibiting approximately one second or less variation
<i>Instrumented execution trace</i>	Observed single trace (high-resolution logs)	12.2 seconds failure detection; 12.3 seconds end-to-end failover completion to Node Beta in an instrumented representative execution
<i>Formal statistical sampling</i>	Not performed/intentionally omitted	Individual per-run timestamps, sample distributions, and statistical variance metrics were intentionally not compiled

Backend failure detection timing is bounded deterministically by the system configuration parameters: three consecutive failed attempts with a 5-second timeout per attempt (t_{timeout}) and a 5-second inter-attempt backoff delay (Δt_{retry}). The detection interval T_{detect} is governed by:

$$2 \cdot \Delta t_{\text{retry}} \leq T_{\text{detect}} \leq 3 \cdot t_{\text{timeout}} + 2 \cdot \Delta t_{\text{retry}}$$

This establishes a deterministic detection window of 10-25 seconds. The lower bound represents immediate socket disconnections (e.g., immediate TCP RST on container termination), while the upper bound represents unacknowledged network timeouts. Across repeated operational test cycles, all observed failovers adhered consistently within this envelope. To illustrate the exact execution sequence, an instrumented trace run logged during container-stop testing is presented: the initial socket refusal was recorded at 13:08:20.719, and attempt 3 was registered at 13:08:32.954, completing failure confirmation in 12.2 seconds before executing failover to Node Beta at 13:08:33.029 (total elapsed time of 12.3 seconds). This trace empirically demonstrates the deterministic sequence in an authentic operational environment.

The same run provides an observed end-to-end failover duration of approximately 12.3 seconds from the first failed request to confirmation that Node Beta was active. Once the failure threshold was reached, the redundant backend was contacted immediately. The active routing state was updated after a successful response from Beta, after which new requests were sent directly to the redundant backend. No additional routing delay was observed during validation. Across repeated test executions, failover timing remained tightly clustered around this duration, with observed variations of approximately one second or less between runs and no observed instances of unbounded delays or failed transitions.

Recovery used a different timing model. While operating on Node Beta, ORVAL probes Node Alpha every 30 seconds using a lightweight health check with a two-second timeout. Consequently, recovery detection can occur between zero and approximately 30 seconds after Alpha becomes functionally available. The exact recovery duration was not measured because the Alpha container was restarted after an uncontrolled waiting period during testing. Once a valid recovery response was detected, however, the routing state was changed immediately, and the system returned to Alpha without requiring user intervention.

The frontend heartbeat operates independently from the failover mechanism and checks backend availability every two seconds, with a two-second timeout. These values describe the monitoring configuration rather than measured network latency. Similarly, live telemetry uses a default five-second polling interval. This interval represents the frequency at which new telemetry is requested and should not be interpreted as an end-to-end telemetry update latency.

During multi-user validation, concurrent users generated little noticeable degradation in ORVAL's responsiveness. Requests continued to be processed while multiple users interacted with dashboards and telemetry visualizations, and no progressive slowdown was observed as the number of concurrent users increased within the tested workload. These observations remain qualitative because no formal load-testing campaign or systematic response-time measurements were performed.

Overall, the measured and configured timings indicate that ORVAL's fault-tolerance mechanisms operate within bounded and predictable intervals. The most significant latency is introduced by deliberate failure detection and retry behaviour rather than by the routing transition itself. The validation therefore supports the intended deterministic timing characteristics of the system without claiming performance results beyond the scope of the available measurements.

8.8. Requirement Validation

The final validation step assessed the extent to which the implemented ORVAL framework satisfies the requirements defined in Chapter 4. Rather than treating each requirement as an isolated feature, related requirements were evaluated together against the validation scenarios presented in Sections 8.2 to 8.7. This provides direct traceability between the original requirements, the implemented system behaviour, and the observed validation results.

Table 16 summarizes the validation of the functional, availability, fault-tolerance, and consistency requirements. Requirements that concern the same system behaviour are grouped to keep the validation concise while preserving traceability to the individual requirement identifiers.

Table 16: Validation of functional, availability, fault-tolerance, and consistency requirements.

Requirement(s)	Validation area	Validation basis	Outcome
FR1-FR2	Telemetry acquisition and visualization	Historical and live telemetry validated under nominal operation	Validated
FR3-FR4	Visualization and dashboard management	Dashboard creation, manipulation, saving, loading, and layout interaction	Validated
FR5-FR6	Dashboard decks and telemetry export	Deck management and CSV export tested during normal operation	Validated

FR7	Service health monitoring	Continuous backend and database health monitoring during normal and fault scenarios	Validated
FR8, AFR2-AFR3	Failure detection and threshold logic	Controlled Alpha failure and verification of the three-attempt mechanism	Validated
FR9, AFR1, AFR4, AFR7	Autonomous failover and fault isolation	Repeated Alpha failures with automatic transition to Beta	Validated
FR10, CMR1-CMR2	Active routing and request consistency	Concurrent-user and in-flight request scenarios during controlled Alpha failure	Validated within tested application scope
FR11, AFR5-AFR6	Recovery detection	Repeated recovery of Alpha through background health monitoring	Validated
FR12, CMR3-CMR4	Automatic return-to-home	Repeated Alpha recovery and automatic restoration in AUTO mode	Validated
CMR5-CMR6	Visualization and routing continuity	Multi-user failover, recovery, and concurrent-request scenarios	Validated within tested application scope
FR13	Administrative health and routing control	Administrative monitoring and manual routing overrides	Validated
FR14-FR15	User and content management	User, dashboard, and deck management functionality	Validated

FR16	System and log management	Log generation, inspection, and administrative system controls	Validated
-------------	---------------------------	--	-----------

The results show that the core functional and fault-tolerance requirements were directly exercised during validation. In particular, the combination of controlled backend failures, repeated failure-recovery cycles, and concurrent-user scenarios demonstrated that the redundancy mechanisms were not merely present in the implementation but operated as intended under actual service transitions. The consistency requirements are bounded by the scope of the architecture: routing coordination is guaranteed within an ORVAL application instance, while separate client machines operate independently and converge through the implemented routing and configuration mechanisms. No distributed consensus mechanism is claimed.

The non-functional requirements were evaluated using both operational testing and implementation-based evidence. Their results are presented separately in Table 17.

Table 17: Validation of non-functional requirements.

Requirement	Validation area	Validation basis	Outcome
NFR1	Telemetry access and visualization latency	Database indexing, downsampling, and incremental retrieval; operational comparison with previous workflow	Qualified
NFR2	Live telemetry performance	Repeated live operation using the configured five-second update interval	Validated
NFR3	Documentation and maintainability	Developer documentation, deployment documentation, and in-application user manual	Supported
NFR4	Reproducible deployment	Repeated containerized deployment, restart, and redeployment on the intended environment	Validated

NFR5	Multi-user operation	Qualitative multi-session concurrent trials during nominal and fault conditions; verified absence of instance-level route conflict	Validated within operational scope
NFR6-NFR7	Observability and operational visibility	Application logs, failure/recovery events and administrative monitoring interfaces	Validated
NFR8	Operational independence	Operation of prepared production images on ground-segment system without Internet connectivity	Validated

NFR1 is evaluated as Qualified based on operational field trials. During testing in the Redwire ground-segment environment, typical historical parameter queries were retrieved and rendered within the targeted 10-second envelope, presenting an operational improvement over the multi-minute response times observed with WebMUST. However, because query response times were not instrumented across a controlled, statistically isolated benchmark suite, this requirement is classified as operationally verified rather than formally benchmarked. The architectural optimizations supporting this latency profile – in-database downsampling, dynamic metadata resolution, and connection pooling – are verified through functional query execution.

NFR3 is supported by the available documentation evidence, while NFR8 was validated through operation of the prepared production images on the ground-segment computers without Internet connectivity. ORVAL includes dedicated documentation for future developers and engineers as well as an in-application user manual. NFR8 is satisfied by establishing an air-gapped runtime boundary: while the initial Continuous Integration/Continuous Delivery (CI/CD) container compilation requires package registry access, the resulting production container artifacts execute in a fully isolated ground-segment environment with zero external runtime dependencies.

Overall, the requirement validation demonstrates that the final ORVAL implementation addresses the defined functional, availability, fault-tolerance, consistency, and operational requirements within the scope of the validation campaign, with NFR1 explicitly qualified due to the absence of a controlled comparative benchmark. Most requirements were directly validated through controlled testing, while the few requirements without a formal quantitative benchmark were explicitly qualified rather than overstated. This distinction ensures that the reported results reflect the actual evidence available from the validation campaign.

8.9. Results Summary

The validation results demonstrate that ORVAL achieved the primary objective of providing reliable and consistent telemetry visualization while remaining operational in the presence of backend failures. The final implementation satisfied the defined requirements within the scope of the validation campaign, with the exception of NFR1, which was explicitly qualified because no controlled comparative latency benchmark was performed. More importantly, the results demonstrate that the fault-tolerance mechanisms operate as an integrated system rather than as isolated recovery functions.

The most significant result was the successful autonomous failover from Node Alpha to Node Beta. When the active backend was deliberately stopped, ORVAL detected the failure through its deterministic three-attempt mechanism and automatically transferred operation to the redundant backend without user intervention. In a fully instrumented validation run, failure was detected at 12.2 seconds and failover completed at 12.3 seconds. Across approximately 20 repeated test executions in the ground-segment environment, the failover sequence consistently completed within this same narrow window, with observed timing varying by approximately one second or less. This empirical consistency confirms that the deterministic retry thresholds reliably govern system behaviour within a 10-25 second detection bound. From the users' perspective, the transition was largely transparent: most engineers did not notice that a failover had occurred unless they saw the corresponding system notification.

Recovery proved equally important. Once Node Alpha was restored, ORVAL periodically monitored it through its independent recovery mechanism and automatically returned to the primary path in AUTO mode once a valid response was detected. Unlike failover, which necessarily introduces a detection interval during which live visualizations temporarily stop receiving new measurements, the return-to-home transition itself was effectively instantaneous once recovery was detected. Existing sessions, dashboards, and visualizations remained intact with no page refresh, re-authentication, or manual reconfiguration required. Repeated recovery scenarios produced the same expected behaviour. The validation also demonstrated consistent routing behaviour during concurrent use. Multiple users and simultaneous requests were able to operate through the same active service path, while in-flight requests adapted to routing changes without producing conflicting failover transitions. No routing inconsistencies or user-visible failures were observed in the final validation version. This consistency is guaranteed within an ORVAL application instance, while coordination between independent client machines relies on the deterministic routing and configuration mechanisms described in Chapter 5.

Beyond individual fault scenarios, repeated failure-recovery cycles demonstrated that ORVAL could return to a stable primary configuration and subsequently perform another failover when required. Backend and database failures were repeatedly introduced and recovered during development and ground-segment validation, with the final implementation handling the tested scenarios without manual intervention. This demonstrated that fault tolerance was not limited to a single successful transition, but formed a repeatable operational behaviour.

Finally, the results highlight the broader value of ORVAL beyond its original role as a replacement for the previous visualization workflow. The existing WebMUST-based workflow had an observed latency of approximately three minutes, whereas ORVAL was consistently experienced by engineers as substantially faster while providing a significantly more interactive and integrated visualization environment. A formal side-by-side latency benchmark was not performed, so this performance improvement is reported as an operational observation rather than a controlled comparative measurement. Nevertheless, the combination of efficient telemetry retrieval, incremental live updates, database optimization, autonomous fault handling, and a substantially richer visualization environment resulted in a system that went beyond the original expectation of providing another telemetry visualization tool.

ORVAL therefore emerged not simply as a replacement visualization application, but as a deterministic, fault-tolerant visualization framework designed around continuous operation. Its architecture separates the visualization and fault-tolerance mechanisms from the underlying telemetry source, meaning that the same principles can be applied to other data sources by adapting the data-access layer. The validation presented in this chapter consequently demonstrates both the successful implementation of the original requirements and the broader feasibility of ORVAL as a reusable framework for reliable operational data visualization.

9. Discussion

This thesis investigates how a deterministic, fault-tolerant telemetry visualization framework can be engineered to ensure reliable data access in the presence of backend failures. Driven by the operational vulnerabilities identified in Chapter 2 and grounded in the architectural principles established in Chapters 3 through 6, ORVAL was designed, implemented, and empirically evaluated. The validation results presented in Chapter 8 demonstrate that the framework successfully achieves deterministic failover, robust route convergence, and autonomous recovery across both nominal and degraded ground-segment conditions.

However, the significance of these empirical findings extends beyond verifying functional requirement satisfaction. This chapter interprets the observed results in direct relation to the underlying research objective, examining how deterministic fault handling, layered redundancy, and application-tier routing govern system dependability. Furthermore, it critically evaluates the architectural trade-offs, operational boundaries, and broader generalizability of the framework beyond its original ground-segment deployment at Redwire.

9.1. Interpretation of Results

The validation results confirm that ORVAL's operational resilience is driven by deterministic coordination rather than passive infrastructure duplication. By shifting failure detection and route arbitration into an autonomous application-tier state machine, the system eliminates human latency and operator error during ground-segment anomalies. The observed 12.3-second failover trace, supported by repeated consistent executions exhibiting approximately one second or less variation, confirms that predictable, rule-based decision logic can reliably maintain telemetry access within bounded operational margins without relying on complex consensus infrastructure.

The deterministic approach does, however, introduce a deliberate trade-off between reaction speed and stability. A single failed request is insufficient to trigger failover because transient network interruptions or temporary processing delays can occur during normal operation. Requiring three consecutive failures therefore introduces a bounded detection interval of approximately 10-25 seconds. This delay is accepted in exchange for reducing the likelihood of unnecessary transitions caused by short-lived communication problems. The validation results support this design choice: the observed failure detection occurred within the configured bounds, while repeated scenarios did not produce unnecessary failovers. The same principle is reflected in the distinction between normal and long-running requests, where longer operations use an extended timeout to avoid treating expected processing time as an immediate service failure. The result is not the fastest possible failure response, but a more predictable and context-aware one.

The results also show that fault tolerance operates at multiple levels. Database failures can be handled within the active backend through the available database paths, while a backend failure causes the frontend routing mechanism to escalate the response to the redundant backend. This layered approach allows a failure to be handled at the lowest appropriate level instead of immediately triggering a broader service transition. Consequently, the reliability of ORVAL is better understood as a coordinated hierarchy of failure-handling mechanisms rather than as a single failover function.

A critical consideration highlighted by these results is service flapping. If a failing primary node alternates between responsive and failing states at intervals exceeding the 30-second recovery poll, an eager failback could initiate repeated route reassignments. In the validated configuration, this was mitigated by the requirement for three consecutive failures to leave Alpha, ensuring that flapping results in stability on Beta primarily until Alpha achieves sustained reachability. Introducing a multi-probe hysteresis window remains the logical next step for mission hardening.

Routing consistency further contributes to operational reliability. ORVAL was not designed to distribute users between Alpha and Beta as a load-balancing system; such behaviour was outside the requirements and would introduce unnecessary operational complexity. Allowing different users to independently operate through different backend and database paths could make the system state harder to understand and complicate troubleshooting. The return-to-home mechanism therefore serves a broader purpose than simply restoring the preferred server: it returns the environment to a common primary configuration after recovery. Within the tested application scope, concurrent requests followed the same active routing state, and clients converged on the redundant path during failure and subsequently returned to the primary path. This provides a simpler and more predictable operational state without requiring the complexity of a distributed consensus mechanism.

Finally, the validation results demonstrate that ORVAL evolved beyond the initial objective of replacing the existing WebMUST-based visualization workflow. The resulting system combines telemetry visualization and retrieval with autonomous fault-handling, routing coordination, recovery mechanisms, and operational controls. The separation between these mechanisms and the underlying data-access layer further provides a basis for adapting the framework to other structured data sources, although such alternative sources were not validated in this study.

Taken together, these results show that the main contribution of ORVAL lies not in any single feature, but in the integration of deterministic decision-making, redundancy, recovery, and visualization into one operational framework. The validation demonstrates that these mechanisms can work together without requiring users to manually initiate service failures, while the architecture provides a foundation that extends beyond the specific telemetry workflow for which ORVAL was originally developed.

9.2. Strengths and Trade-offs

The validation results demonstrate that the chosen architecture provides the intended reliability benefits, but these benefits are accompanied by deliberate architectural compromises. ORVAL's fault-tolerance mechanisms were therefore not designed to maximize a single property such as reaction speed or scalability, but to achieve a balance appropriate to its intended operational environment.

9.2.1. Reliability vs. Complexity

The main strength of ORVAL's architecture is its ability to remain operational when individual backend or database components become unavailable. Redundant service paths, autonomous failure detection, database-level failover, recovery monitoring, and automatic return-to-home collectively remove several single points of failure from the visualization workflow. However, this resilience necessarily increases architectural complexity compared with a conventional visualization application.

The additional routing state, health checks, retry mechanisms, recovery logic, and redundant infrastructure introduce more components that must be implemented, configured, monitored, and maintained. Each additional mechanism also creates potential failure modes of its own. This means that redundancy should not be treated as an unconditional improvement: it is valuable when the operational consequences of component failure justify the additional complexity. In the context of ORVAL, the validation results indicate that this trade-off is justified because the added mechanisms directly address the requirement for continued telemetry access during backend and database failures.

9.2.2. Determinism vs. Reaction Speed

ORVAL deliberately prioritizes predictable failure handling over the fastest possible reaction. The three-attempt mechanism requires consecutive failed attempts before a backend is considered unavailable, resulting in a configured detection window of approximately 10-25 seconds. A simpler one-attempt mechanism could reduce this delay, but would also make the system more sensitive to transient network interruptions and temporary service delays.

The chosen approach therefore represents a conscious compromise. ORVAL does not attempt to predict whether a failure is genuine; instead, it applies a fixed and explainable decision rule. This makes the behaviour easier to understand, reproduce, and troubleshoot, at the cost of accepting a bounded delay before failover.

9.2.3. Centralized Coordination vs. Scalability

The use of a shared active routing state within an ORVAL application instance simplifies coordination between concurrent requests. Rather than allowing individual requests or users to independently select a backend, routing decisions are based on the same active service state. This reduces the risk of conflicting failover decisions within an application instance and provides a clear routing state for the requests handled by that instance.

This simplicity comes with an architectural limitation. ORVAL was designed around a limited number of redundant service paths and does not implement distributed consensus or globally coordinated routing across an arbitrary number of independent application instances. As the number of nodes or independently operating clients increases, maintaining a globally consistent routing state would become more complex. The current architecture is therefore well suited to the deployment scope for which ORVAL was designed, but it would require additional coordination mechanisms to support substantially larger or more distributed environments.

9.2.4. Autonomous Recovery vs. Premature Recovery

Automatic return-to-home removes the need for manual intervention after a primary backend has recovered, but autonomous recovery introduces its own risk: a service that becomes temporarily reachable is not necessarily stable enough to resume the active role. ORVAL addresses this by requiring the recovered backend to provide a valid response to its health check before restoring it as the active path.

This reduces the risk of switching back solely on the basis of basic network connectivity. Nevertheless, the mechanism cannot guarantee that the primary service will remain stable after a successful recovery check. A backend could pass the health check and subsequently fail again, resulting in another failover cycle. Autonomous recovery therefore improves operational continuity while retaining an inherent trade-off between restoring the preferred path promptly and waiting for stronger evidence of sustained stability. The implemented approach favours functional verification without introducing a substantially more complex recovery protocol.

Overall, the architecture reflects deliberate compromises rather than attempts to optimize every property simultaneously. ORVAL accepts additional complexity to gain resilience, accepts bounded detection delays to gain deterministic behaviour, limits centralized coordination to maintain manageable routing consistency, and uses functional recovery checks to balance autonomous restoration against premature switching. These trade-offs are appropriate to the defined scope, while also identifying clear boundaries for future architectural expansion.

9.3. Limitations

The validation results provide evidence that ORVAL behaves as intended under the tested failure conditions, but the conclusions remain bounded by the scope of the validation. Several limitations therefore need to be considered when interpreting the results.

9.3.1. Validation Scope

The validation was performed in a controlled ground-segment environment in which failure scenarios were deliberately introduced rather than observed under naturally occurring operational failures. This approach made it possible to reproduce specific failure conditions and evaluate the resulting behaviour consistently. However, it does not establish how frequently such failures occur in practice or how ORVAL would behave under unpredictable combinations of faults. The results demonstrate that the failover mechanism is functionally reproducible and temporally consistent across repeated trials under controlled conditions, though they do not establish formal statistical reliability over long-term operations.

The infrastructure used for validation consisted of two redundant backend paths and their associated database paths. This was sufficient to evaluate the redundancy and routing mechanisms defined for ORVAL, but does not demonstrate scalability to larger distributed infrastructures containing additional backend nodes or database instances. Similarly, the tested failure scenarios focused on service-level failures that could be reproduced within the deployed environment. Backend and database unavailability and repeated failure-recovery cycles were tested, but not every possible network or infrastructure failure mode, such as arbitrary packet loss, network partitioning, hardware failure, or complex combinations of simultaneous faults.

Finally, ORVAL was designed for operation within the ground segment and was validated accordingly. The results therefore demonstrate its behaviour within the intended ground-segment context, but do not cover interactions with other mission systems or the broader end-to-end conditions of an operational mission.

9.3.2. Infrastructure Dependencies

ORVAL's database failover relies on redundant database paths containing equivalent telemetry data. During validation, the database instances were launched together and populated with the same data, allowing ORVAL to switch between equivalent data sources. The mechanism responsible for maintaining this equivalence is part of the underlying database infrastructure rather than ORVAL itself. Consequently, the validation demonstrates ORVAL's ability to switch between available database paths, but does not independently validate the correctness of the underlying database replication or synchronization, nor data integrity under database failures.

This dependency is important when interpreting the fault-tolerance results. ORVAL can detect the unavailability of a database path and use the redundant path, but the quality and consistency of the data available after such a transition depend on the underlying database infrastructure. The demonstrated resilience should therefore be understood as application-level failover built on top of an existing redundant data layer.

9.3.3. Performance and Long-Term Validation

Testing and development took place iteratively over several weeks, including testing before the final implementation was fully completed. However, no dedicated long-duration endurance campaign was performed in which ORVAL operated continuously under representative mission workloads for an extended period. The validation therefore provides evidence of repeated correct behaviour but does not establish long-term stability, degradation, or reliability over months of continuous operation.

The validation also did not include a controlled performance benchmark or formal load-testing campaign. ORVAL was observed to provide substantially faster telemetry access than the previous WebMUST-based workflow in operational use, but the two systems operate under different architectures, workflows, and network conditions. Their observed performance therefore cannot be interpreted as a direct quantitative benchmark. Likewise, multiple simultaneous users were used to validate concurrent operation within the tested workload, but no formal campaign was performed to determine a maximum number of concurrent users or a quantitative scalability limit. NFR1 and NFR5 should consequently be understood as validated within the tested operational environment rather than through a formal performance or scalability study.

These limitations define the boundaries of the conclusions that can be drawn from the validation. The results demonstrate that ORVAL can autonomously detect and handle the tested backend and database failures, coordinate failover and recovery, and maintain the intended routing behaviour within the validated ground-segment configuration. The evidence therefore supports the effectiveness of the implemented mechanisms within this scope, while broader claims regarding statistical reliability, infrastructure-wide fault tolerance, and long-term endurance, or quantitative scalability require further validation.

10. Conclusion

This thesis investigated how a deterministic, fault-tolerant telemetry visualization framework can be designed and implemented to provide reliable and consistent access to telemetry data in the presence of backend failures. The work originated from a practical need within a space testing and satellite operations environment at Redwire, where telemetry visualization must remain usable while the underlying services are subject to failures and where operational continuity cannot depend entirely on manual intervention.

The research established the problem context and relevant state of the art, translated the identified needs into explicit system requirements, and subsequently used these requirements as the basis for the architecture and implementation of ORVAL. The framework was developed iteratively, with design decisions and implementation choices refined throughout the development process. The resulting system was then evaluated against the identified requirements through controlled failure scenarios in the intended ground-segment environment.

ORVAL ultimately combines interactive telemetry visualization with an integrated fault-tolerance mechanism. Rather than treating failure handling as a separate operational procedure, ORVAL incorporates failure detection, failover, recovery, and routing into the application itself. This allows the system to respond to backend failures autonomously according to predefined and reproducible decision rules, while preserving visualization state and automatically resuming telemetry updates after service transitions.

The following sections answer the research question directly, summarize the principal findings, and clarify the technical and practical contribution of the work.

10.1. Answer to the Research Question

The research question of this thesis was:

Research Question

How can a deterministic, fault-tolerant telemetry visualization framework be designed and implemented to provide reliable and consistent access to telemetry data in the presence of backend failures?

The results demonstrate that such a framework can be achieved by combining redundant service paths, deterministic health and failure detection, autonomous threshold-based failover, continuous recovery monitoring, and coordinated return-to-home routing within a single application architecture. The essential principle is that redundancy alone is insufficient: reliable access requires the system to autonomously determine when an active service has become unavailable, transition to an operational alternative according to a predefined decision rule, and subsequently verify recovery before restoring the preferred service path.

The first sub-research question addressed the requirements that a fault-tolerant telemetry visualization system must satisfy in a space testing and mission operations environment. The analysis resulted in a requirements set covering not only telemetry acquisition and visualization, but also service health monitoring, failure detection, automatic failover, routing consistency, recovery detection, return-to-home behaviour, observability, operational independence, availability, and multi-user consistency. These requirements established that fault tolerance could not be considered independently from the visualization workflow: maintaining access to telemetry data also requires the routing state and visualization behaviour to remain predictable during service transition.

The second sub-research question concerned the architectural approach required to satisfy these requirements. ORVAL addresses this through a layered fault-tolerance design. Database failures can be handled within the active backend through redundant database paths, while a backend failure causes the frontend to transition to the redundant backend path. Failure decisions are deterministic: a service is not considered unavailable after a single unsuccessful request, but only after the configured failure condition has been reached. The three-attempt mechanism therefore provides a predictable and inspectable basis for failover while reducing unnecessary transitions caused by transient failures. Once the primary backend becomes available again, recovery monitoring verifies its operational state and enables an automatic return-to-home transition. This makes the redundant path an active part of the system's reliability mechanism rather than merely a passive backup.

The third sub-research question addresses whether the resulting implementation satisfies the identified requirements under controlled failure scenarios. Validation across repeated failure-recovery executions, complemented by a detailed timestamped trace, demonstrated reproducible autonomous transitions between redundant service paths, stable timing behaviour within the configured bounds, and consistent routing coordination for concurrent users. The observed failover behaviour remained within the configured detection interval, while recovery was initiated automatically after the primary service became operational again. The validation confirms that the functional, availability, and fault-tolerance requirements were fully satisfied within the tested ground-segment configuration. Non-functional performance aspects, specifically query latency comparison against latency infrastructure (NFR1), were verified through operational field observations and remain qualified in the absence of a controlled side-by-side benchmark.

Taken together, these findings provide a direct answer to the main research question. A reliable and consistent telemetry visualization framework can be designed by treating visualization, redundancy, failure detection, failover, recovery, and routing as coordinated parts of one system rather than as independent features. Deterministic decision rules make failure handling predictable; redundant service paths prevent an individual backend failure from inherently terminating telemetry access; recovery monitoring allows the system to restore its preferred operational state without manual intervention; and coordinated routing provides a common and understandable service state for users during transitions.

This approach also demonstrates why the resulting system can be considered a framework rather than solely a replacement visualization application. The fault-tolerance and visualization mechanisms are separated from the underlying data-access layer, allowing the architecture to be adapted to other structured data sources by changing the corresponding data-access integration. ORVAL therefore extends beyond the original objective of replacing the existing telemetry visualization workflow: it provides a reusable architectural approach for combining operational data visualization with deterministic fault tolerance.

Within the scope of the validation, the central objective of the research was therefore achieved. ORVAL demonstrates that reliable telemetry access in the presence of backend failures does not require users to manually manage service transitions. Instead, reliability can be embedded into the visualization framework itself through deterministic failure handling, redundant service paths, autonomous recovery, and coordinated routing.

10.2. Main Findings

The validation and analysis of ORVAL resulted in five main findings. Together, these findings capture the most important outcomes of the developed framework and directly reflect the reliability, determinism, recovery, consistency, and operational improvements investigated throughout this thesis.

10.2.1. *Reliable Telemetry Access Through Autonomous Failover*

ORVAL successfully maintained telemetry access during the tested backend failure scenarios by autonomously transitioning from the active backend to its redundant counterpart. This demonstrates that the implemented redundancy was operational rather than merely structural: when the active service became unavailable, and an operational alternative remained available, the system could continue the telemetry visualization workflow without requiring manual failover.

10.2.2. *Predictable Failure Detection and Failover*

The deterministic three-attempt mechanism provided a predictable and reproducible basis for deciding when a backend should be considered unavailable. Across repeated test cycles, failover timing remained stable within the configured 10-25 second window, exhibiting minimal observed variation ($\sim \leq 1$ s) around the documented 12.3-second execution trace. This confirmed that the mechanism reliably balances rapid failure response with transient error suppression.

10.2.3. *Autonomous Recovery and Return-to-home*

Fault tolerance extended beyond the initial failover. Once the primary backend became operational again, continuous recovery monitoring allowed ORVAL to verify its availability and automatically restore the primary service path. Repeated failure-recovery cycles confirmed that this return-to-home behaviour could occur without manual routing intervention, completing the intended transition from primary operation to redundancy and back again.

10.2.4. Coordinated Routing During Service Transition

Within the validated application environment, concurrent requests followed the active routing state during both failover and recovery. Tested clients therefore transitioned to the redundant path and subsequently converged on the primary path without requiring individual users to select or restore a backend manually. This demonstrates that routing coordination formed an integral part of the fault-tolerance behaviour rather than being left to individual client sessions.

10.2.5. An Expanded Operational Visualization Framework

ORVAL substantially expanded the operational capabilities of the previous visualization workflow by combining telemetry visualization and retrieval with integrated failure detection, failover, recovery, routing, and operational controls. Operational use also indicated markedly faster access to telemetry than the previous workflow, although, as noted in Chapter 8, this was an operational observation and not a controlled benchmark. The resulting system therefore extends the integration of visualization and deterministic fault-tolerance mechanisms within a single operational framework.

10.3. Contribution

The contribution of this thesis can be considered from both a technical and a practical perspective. Technically, the work contributes an integrated approach to fault-tolerant telemetry visualization. Practically, it demonstrates this approach through a functioning framework developed and validated in an actual space ground-segment environment.

10.3.1. Technical Contribution

The primary technical contribution is a deterministic, fault-tolerant telemetry visualization architecture that integrates reliability mechanisms directly into the visualization framework. Rather than treating telemetry visualization and service fault handling as separate concerns, ORVAL combines them within a single architectural approach in which redundant service paths, deterministic failure detection, autonomous failover, recovery monitoring, and coordinated routing work together to maintain access to telemetry data.

A second contribution is the integration of these mechanisms into an operational visualization workflow. The resulting architecture does not rely on a separate manual procedure for handling backend failures. Fault detection, service transitions, recovery, and routing coordination are incorporated into the framework itself, allowing the system to respond according to predefined and reproducible behaviour.

Finally, the architecture provides a separation between the visualization and fault-tolerance mechanisms and the underlying data-access layer. This separation means that the core framework is not inherently tied to a single telemetry data source. By adapting the corresponding data-access integration, the architectural approach provides a basis for applying the same visualization and fault-tolerance mechanisms to other structured data sources. This generalizability is an architectural property of the resulting framework rather than a claim that such alternative data sources were themselves validated within this thesis.

10.3.2. Practical Contribution

From a practical perspective, the thesis resulted in a working telemetry visualization framework developed for an actual space ground-segment context. ORVAL was not evaluated solely as a conceptual architecture or prototype: its implemented functionality and fault-tolerance mechanisms were exercised within the intended operational environment under controlled failure scenarios.

The framework also reduces operational dependence on a single backend service path. Through redundant service paths and autonomous failover, a failure of the active backend does not inherently require an engineer to manually restore telemetry access when an operational alternative remains available. This embeds part of the operational resilience directly into the system.

For both the test-engineering workflow and future satellite operations, ORVAL combines telemetry visualization and retrieval with the integrated reliability mechanisms developed throughout the project. The resulting framework therefore provides a broader operational capability than the previous visualization workflow. Crucially, by reducing the need for manual intervention during service failures, ORVAL demonstrates the potential value of integrated fault-tolerance mechanisms in high-stakes operational environments.

Taken together, these contributions demonstrate that fault tolerance can be incorporated directly into an operational telemetry visualization framework without turning service failure handling into a manual operator task. The thesis therefore contributes both an architectural approach to deterministic fault-tolerant visualization and a practical implementation of that approach within a space testing context.

10.4. Final Perspective

ORVAL began with a practical objective: to provide a more capable alternative to the existing WebMUST-based telemetry visualization workflow. Through the development process, however, the scope of the resulting system extended beyond replacing a visualization tool. The need for reliable telemetry access in the presence of backend failures led to the integration of deterministic failure detection, autonomous failover, recovery monitoring, and coordinated routing directly into the visualization framework.

As a result, ORVAL represents more than an individual telemetry application. It demonstrates how operational data visualization can incorporate fault tolerance as an inherent part of its architecture rather than treating service failures as an external problem for users or engineers to manage. Visualization, data access, redundancy, failure handling, and recovery are brought together into a single framework designed around predictable operational behaviour.

This separation between these mechanisms and the underlying data-access layer further gives the resulting framework a broader significance. Although ORVAL was developed and validated for a specific space telemetry environment, its architecture provides a basis for adapting the same approach to other structured data sources where reliable operational visualization and service continuity are important.

In this sense, the evolution from a replacement for an existing visualization workflow into a deterministic, fault-tolerant framework represents the central outcome of the project. ORVAL demonstrates that reliability does not have to be added around a visualization system as a separate operational procedure; it can be designed into the system itself. The result is an approach to operational data visualization in which access to information and resilience of the service are treated as parts of the same engineering problem.

References

- Al-Sayed, M. (2022). High-availability clusters: A taxonomy, survey, and future directions. *Journal of Systems and Software*, 187, 111208. Retrieved from <https://www.sciencedirect.com/science/article/abs/pii/S0164121221002806>
- Amazon Web Services (AWS). (2026). *Active-active and active-passive failover*. Retrieved from Amazon Web Services: <https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/dns-failover-types.html>
- Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1, 11 - 33. doi:10.1109/TDSC.2004.2
- Baumgartner, A., Martínez-Heras, J., Donati, A., & Quintana, M. (2005). MUST – A Platform for Introducing Innovative Technologies in Operations. *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS 2005)*. Munich. Retrieved from https://www.researchgate.net/publication/228661784_MUST-A_Platform_for_Introducing_Innovative_Technologies_in_Operations
- Chandra, T., & Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 225-267. doi:10.1145/226643.226647
- Cisco Systems. (2007). *Network Management: Implementing and Operating High Availability Solutions*. Retrieved from Cisco Systems: https://www.cisco.com/en/US/technologies/tk869/tk769/white_paper_c11-449655.html
- ECSS. (2016). *Space engineering: Ground systems and operations — Telemetry and telecommand packet utilization (ECSS-E-ST-70-41C)*. European Cooperation for Space Standardization.
- European Cooperation for Space Standardization (ECSS). (2016). *ECSS-E-ST-70-41C: Telemetry and telecommand packet utilization*. ECSS Secretariat / ESA-ESTEC.
- European Space Agency (ESA). (2007). *Recovery from Unexpected Events*. Retrieved from <https://www.esa.int/esapub/br/br265/br265.pdf>
- European Space Agency (ESA). (2016). Space-to-Ground Data Viewer. *Living Planet Symposium*. European Space Agency.
- European Space Agency (ESA). (2024). *Mission Utility and Support Tools (MUST)*. Retrieved from ESA: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Mission_Utility_and_Support_Tools_MUST
- European Space Agency (ESA). (2024). *Routine operations*. Retrieved from European Space Agency: https://www.esa.int/About_Us/ESOC/Routine_operations

- European Space Agency (ESA). (2026). *Method and apparatus for analysing time series data*. Retrieved from ESA: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Method_and_apparatus_for_analysing_time_series_data
- Fortescue, P. W., Swinerd, G. G., & Stark, J. P. (2011). *Spacecraft Systems Engineering* (4th ed.). Chichester: John Wiley & Sons.
- HAProxy Technologies. (2026). *Configure two HAProxy Enterprise servers for active/standby clustering*. Retrieved from HAProxy Technologies: <https://www.haproxy.com/documentation/haproxy-enterprise/administration/high-availability/active-standby/>
- HAProxy Technologies. (2026). *Failover*. Retrieved from HAProxy Technologies: <https://www.haproxy.com/glossary/what-is-failover>
- HAProxy Technologies. (2026). *Health checks*. Retrieved from HAProxy Technologies: <https://www.haproxy.com/documentation/haproxy-configuration-tutorials/reliability/health-checks/>
- Istio Authors. (2026). *Traffic Management*. Retrieved from Istio Documentation: <https://istio.io/latest/docs/concepts/traffic-management/>
- Kubernetes Authors. (2026). *Kubernetes Self-Healing*. Retrieved from Kubernetes Documentation: <https://kubernetes.io/docs/concepts/architecture/self-healing/>
- Kubernetes Authors. (2026). *Liveness, Readiness, and Startup Probes*. Retrieved from Kubernetes Documentation: <https://kubernetes.io/docs/concepts/workloads/pods/probes/>
- Microsoft Corporation. (2025). *Redundancy, replication, and backup*. Retrieved from Microsoft Learn: <https://learn.microsoft.com/en-us/azure/reliability/concept-redundancy-replication-backup>
- NASA. (2024). *Basics of Space Flight: Telemetry, Tracking and Command*. Retrieved from <https://science.nasa.gov/learn/basics-of-space-flight/>
- NASA. (2024). *Ground Data Systems and Mission Operations*. Retrieved from NASA Science: <https://www.nasa.gov/smallsat-institute/sst-soa/ground-data-systems-and-mission-operations/>
- National Aeronautics and Space Administration (NASA). (2026). *Open Mission Control Technologies (Open MCT)*. Retrieved from NASA Science: <https://ammos.nasa.gov/openmct/>
- National Institute of Standards and Technology (NIST). (2026). *Fault Tolerance — CSRC Glossary*. Retrieved from Computer Security Resource Center (CSRC), NIST: https://csrc.nist.gov/glossary/term/fault_tolerance

- National Institute of Standards and Technology (NIST). (2026a). *Availability — CSRC Glossary*. Retrieved from Computer Security Resource Center (CSRC), NIST: <https://csrc.nist.gov/glossary/term/availability>
- National Institute of Standards and Technology (NIST). (2026b). *High Availability — CSRC Glossary*. Retrieved from Computer Security Resource Center (CSRC), NIST: https://csrc.nist.gov/glossary/term/high_availability
- Német, A. (2023). OXI: An online tool for visualization and annotation of satellite time series data. *SoftwareX*, 23, 101476. Retrieved from <https://www.sciencedirect.com/science/article/pii/S2352711023001723>
- Oracle Corporation. (2026). *High Availability*. Retrieved from Oracle: <https://docs.oracle.com/en-us/iaas/Content/cloud-adoption-framework/high-availability.htm>
- Prakash, A. (2022). Data-driven mixed-integer linear programming-based optimisation for efficient failure detection in large-scale distributed systems. *European Journal of Operational Research*. Retrieved from <https://www.sciencedirect.com/science/article/abs/pii/S0377221722000947>
- Solenix. (2026). *MUST: Mission Utility and Support Tools*. Retrieved from Solenix (or Solenix — MUST): <https://www.solenix.com/products/must>
- Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). CreateSpace Independent Publishing Platform.

EXECUTIVE SUMMARY // SYSTEM SPECIFICATION

Reliable access to telemetry data is essential during satellite testing and in-orbit operations, where engineers depend on visualization systems to monitor spacecraft behaviour and evaluate test results. When supporting backend services become unavailable, recovery often depends on manual intervention. This thesis introduces ORVAL (Orchestrator for Remote Visualization and Analysis of Logged telemetry), a deterministic, fault-tolerant framework developed for Redwire as an operational replacement for the legacy WebMUST workflow. ORVAL embeds autonomous three-attempt failure detection, dual-tier service redirection, and coordinated return-to-home recovery directly into the visualization tier, ensuring consistent route convergence across concurrent users without requiring manual intervention.

10-25 s	12.3 s	100%
CONFIGURED BOUND	OBSERVED TRACE	AUTONOMOUS RETURN
$2 \cdot \Delta t_{\text{retry}} \leq T_{\text{detect}} \leq 3 \cdot t_{\text{timeout}} + 2 \cdot \Delta t_{\text{retry}}$	Fully instrumented empirical failover from socket drop to active Node Beta Ingestion	Zero-touch failback to primary Node Alpha upon health verification in AUTO mode

CLIENT PRESENTATION TIER	Vue.js (Composition API), Vite, Apache ECharts, Pinia Store
SERVICE REDUNDANCY LAYER	Decoupled dual-node Express/Node.js ingestion paths (Node α / Node β)
PERSISTENCE INTEGRATION	MySQL 8.0 connection-pooled data store with in-database time-bucket downsampling
DEPLOYMENT TARGET	Multi-stage containerized Docker deployment engineered for air-gapped ground networks