TECHNICAL INSIGHT

Architectural Principles for Scalable Autonomous Systems

Autonomous systems are moving from experimentation to operational scale. In Ukraine, drone production reached roughly 2 million units in 2024, and the country has introduced hundreds of unmanned aerial systems and ground robotic platforms since Russia's full-scale invasion. The U.S. Department of War is following a similar trajectory through initiatives such as Replicator, Project Maven, Collaborative Combat Aircraft, and Drone Dominance — all of which reflect a broader shift toward AI-enabled operations, human-machine teaming, and autonomous systems deployed at greater scale and speed.

As these systems scale, success depends on more than autonomy logic alone. The underlying architecture determines whether sensors, software components, communications links and decision logic can work together reliably under real-world conditions. Poor architectural choices can lead to brittle integrations, excessive network traffic, limited fault recovery, and systems that are difficult to validate or evolve. This is especially important as autonomous systems expand from individual platforms to collaborative teams, loyal wingmen, and distributed mission architectures.

This brief examines several common architectural issues that can undermine autonomous system performance, resilience, and capabilities and will offer guidance on how to mitigate them.

You can set yourself up for success by addressing the following short list of common architectural priorities:

- Structured data model
- Removing concentrated points of failure
- Error handling
- Scalability
- Separation of concerns
- Safety assurance
- Security

Data Model

An effective data model is the foundation of high-performance autonomous systems. How types and topics are defined directly affects how easy the system is to integrate, how efficiently it uses bandwidth and compute, and how much rework is required as the platform evolves.

Here are a few of the best practices for effectively modeling data for efficient use within autonomous systems.

Model System States, Commands, and Input ~ Not the Event

Using data types that represent the state, commands, and information of your system will help you to end up with a well-defined data model, e.g., VehicleStatus, CommandedPosition, or SensorReading. Topics of these types have a straightforward mapping to their related Quality of Service (QoS) requirements.

For example, new applications joining the system should immediately receive the current system state rather than waiting for the next system state change to occur. Commands must be delivered reliably and deterministically, and sensor readings distributed at scale efficiently.

Use Keyed Topics for Identity

Keys allow a single Topic to manage multiple instances, e.g., a single FlightPath topic managing 50 different aircraft. An element such as aircraft_id in the data type can be used for this purpose.

Instances enable the middleware to efficiently sort, filter, and manage states and resources on a per instance basis. They also support DDS-specific features such as Ownership, Deadlines, and separate History queues.

NOTE: This paper assumes a basic understanding of Data Distribution Service (DDS®) and its terminology. If you are not already familiar with DDS, learn the basics here www.rti.com/products/dds-standard.

Embrace Extensible Types (XTypes)

Autonomous platforms tend to constantly evolve; new sensors, payloads, software versions, and behaviors must often be introduced. Revised and upgraded components need to be efficiently integrated and deployed while maintaining backwards compatibility with the legacy components. Version 1 should be able to communicate seamlessly with later versions which may be using a revised data model. This kind of type extensibility helps reduce upgrade friction by allowing older and newer components to continue interoperating as the system evolves.

Extensible type, i.e., XTypes, is the DDS feature that enables seamless translation between data model revisions. Leveraging this feature supports system evolution and eases the integration of new components.

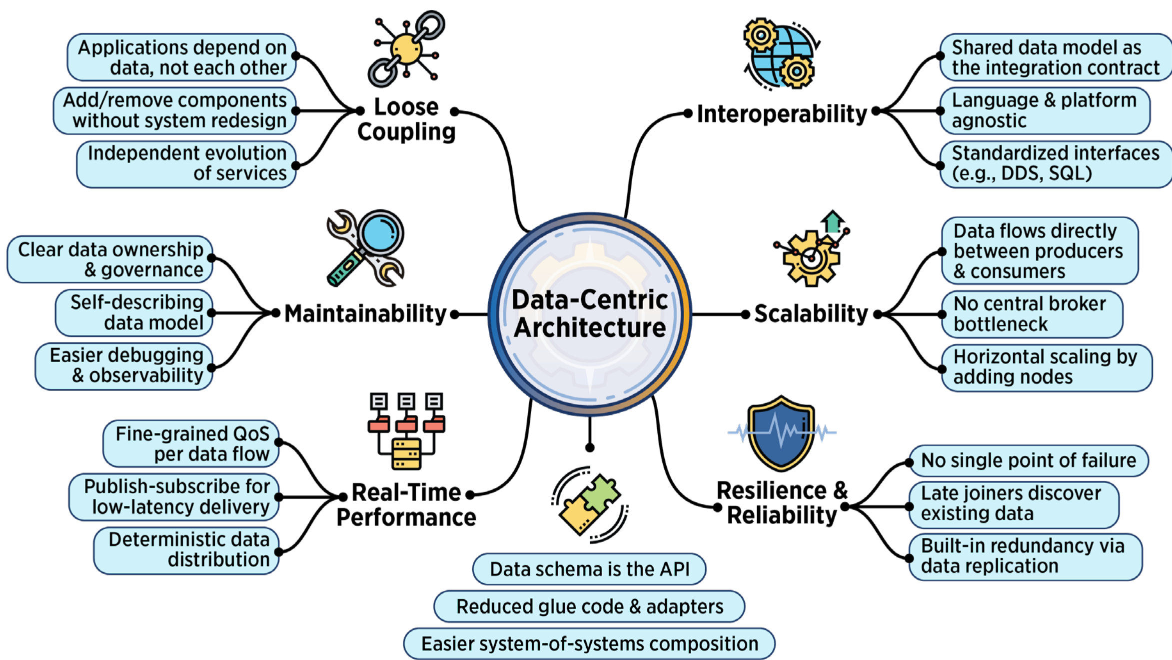

Figure 1: The advantages of a data-centric architecture

Optimize for Efficiency

The physical size of your data impacts latency and throughput. For high-frequency data, use flat structures with primitive types. For large data like high-resolution images, consider the use of Asynchronous Publishing alongside a flow controller to smooth out the flow of data over the network. Additionally, avoid deep nesting of data types. Deeply nested structures increase the complexity of the serialization and deserialization steps which results in additional CPU overhead and latency.

Use Modules to Create Namespaces within the Data Model

As systems grow, naming collisions within the data model becomes a risk. This risk encompasses both data types and topics. Defining namespaces within your data model reduces or eliminates this risk. Namespaces create a clear hierarchy within the data types and constants while mapping cleanly to C++ namespaces or Java packages. Use of constants to define topic names within these namespaces is also encouraged. This hierarchy results in more maintainable code.

Concentrated Points of Failure

A concentrated point of failure consists of one or a small group of parts that can restrict or take down the entire system. These potential points of failure can make your system fragile. To avoid this vulnerability, you should incorporate redundancy and distributed knowledge within your architecture.

A resilient autonomous architecture like DDS helps avoid centralized communication dependencies that can become system-wide failure points. By avoiding centralized brokers and allowing data to flow directly between publishers and subscribers, the architecture reduces the likelihood that one failed component will interrupt the rest of the system. This is the core value of a data-centric architecture: systems integrate to shared data contracts rather than to each other, reducing integration complexity while improving real-time performance, resilience, and composability.

In addition to peer-to-peer communication, implementing additional redundancy strategies is recommended. Here are some steps you can take.

- **Redundant Applications:** Introduce multiple DDS DataWriters and DataReaders for critical topics.
- **Failover Tuning:** Use Quality of Service (QoS) settings — such as Ownership, Deadline, and Liveliness — to tune how the system reacts to a failure.
- **Network Resilience:** Built on the DDS standard, RTI Connext® sends data over multiple network interfaces and can prioritize or failover between them automatically. Note that some of these features require specific support from your network hardware.
- **Service Redundancy:** Deploy multiple instances of Routing Service or Persistence Service on independent machines. These redundant services provide multiple communication paths for your application data. Connext will automatically filter redundant data from being delivered to the application eliminating the need to filter the redundant data within the application.

Error Handling

Autonomous systems must operate without human interaction. To achieve this, they must be able to efficiently recognize and recover from errors. Without these capabilities, autonomy can't be trusted. Therefore, error handling should be considered a core requirement.

For example, consider a path following application that requires regular position updates via GPS. If the update rate falls below a certain threshold, then the desired path can't be followed.

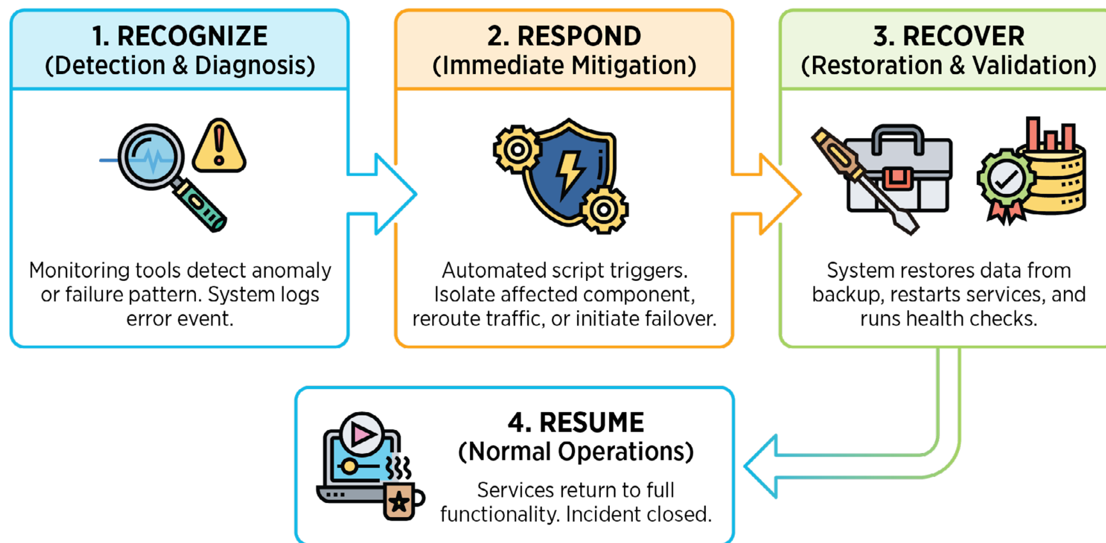

Figure 2: Four steps of the automated error handling life cycle

The DDS deadline QoS can be applied to this use-case. With deadline enabled, the path-follower is notified when GPS updates become intermittent or stop. The path-follower can respond by reducing speed, coming to a stop, etc. Once deadlines stop being missed, then normal operation can resume.

The error handling process is summarized by the following steps.

- **Recognize** when a problem or fault occurs
- **Respond** correctly
- **Recover** and **Resume** back to normal operation

At both the application and middleware levels, systems need mechanisms to detect faults, expose degraded conditions, and support appropriate recovery actions. Some faults, such as dropped or corrupted data are handled automatically. However, others are only reported and it is up to the application to handle them, e.g., deadline missed, sample rejected.

Communication failures generally fall into three categories:

- **Publisher/Subscriber Compatibility:** Most failures occur during development and integration. Tools like the RTI Administration Console and DDS events help identify and resolve these errors. While rare after deployment, these issues usually stem from bad configurations or the addition of new capabilities.
- **Application Health:** Measurements of runtime application health using metrics such as resource usage, liveliness checks, and deadlines is useful for predicting and addressing issues before they become a problem. DDS makes available information such as entity status, missed samples, resource utilization, and security events that can be retrieved from the middleware. Other events, such as application liveliness, are automatically tracked by DDS.
- **Data Delivery:** Errors in delivery typically indicate network instability or resource exhaustion. The subscriber's "sample lost" or "rejected" events are available to diagnose these issues.

Increasing application-level resources such as history depth to address error conditions can be tempting but use caution when taking this approach. There may be an underlying issue and increasing resources may only temporarily mask the problem. RTI tools such as Spy, Ping, Monitor, and PerfTest will help pinpoint the cause and provide a path to resolution.

Scalability

To survive long-term, a successful autonomous system must expand and adapt to the real world. Many factors influence how far these systems must scale. Factors such as mission growth, increased user demand, and new operational environments can dictate this. Because it may be difficult to predict future growth requirements, scalability should be designed in from the beginning.

Connext includes several features that support highly scalable systems:

- **Reliable Multicast:** Reliable multicast helps deliver the same data to many subscribers efficiently, reducing repeated transmission overhead on both the publisher and the network.
- **Data Filtering:** Use content-filtered topics and time-based filters to eliminate unnecessary data transmission. Content filtering filters data based on its content while time-based filtering filters data based on its publishing rate. Connext attempts to filter data at the publisher to prevent unnecessary data from being transmitted across the transport.
- **Performance Optimization:** Use the provided QoS profiles optimized for specific use-cases and data flow requirements as a starting point. Also consider data prioritization techniques, flow control for large data, and batching for small data to increase efficiency.
- **Data Isolation:** Leverage DDS domains, partitions, and topics to restrict data propagation. The following table summarizes some of the data isolation features that DDS offers.

Mechanism	Scope	Changeability	Primary Use Case
Domain ID	Transport Process	Static	Logical isolation between different subsystems or groups of applications within a distributed system. Each DDS domain acts as a virtual databus.
Domain Tags	Participant	Static	Layer of logical isolation within a DDS domain. Domain tags allow you to further subdivide a domain into multiple, isolated groups.
Partitions	Participant Pub Sub	Dynamic	Flexible method to control which Domain Participants, DataWriters, or DataReaders can communicate with each other, even when they use the same Topic.
Topic	Data Instance	Static	Uniquely identifies a stream of data by name and associates it with a specific data type.
Multichannel Datawriter	Pub	Static	Route data to different multicast groups based on the content of the data.
Access Control	Participant Pub Sub	Dynamic via Certificate Revocation Lists	Restricts which participants, publishers, and subscribers can access a topic

Separation of Concerns

Separating system components into specific responsibilities simplifies your architecture and makes it easier to understand. This clarity, in turn, streamlines development, testing, and maintenance.

Isolation is especially critical for safety; you must ensure safety-critical functions operate independently from non-critical activities. When each subsystem or layer operates independently, you improve fault tolerance and prevent failures or overloads in one layer from propagating to others.

Many designers employ the Layered Databus Architecture to implement this separation. This proven approach effectively manages discovery and resource overloads in very large autonomous systems.

Implementing a Layered Databus

There are different approaches that can be used when separating your system into layers. Some examples of these approaches are separation by location, function, or criticality. While no single "best" approach exists, you should select the one that fits your requirements and then apply it consistently. Start with a minimal number of layers and decompose them further, as needed.

The following diagram shows an example of a layered databus in which selected data is moved between layers through controlled routing.

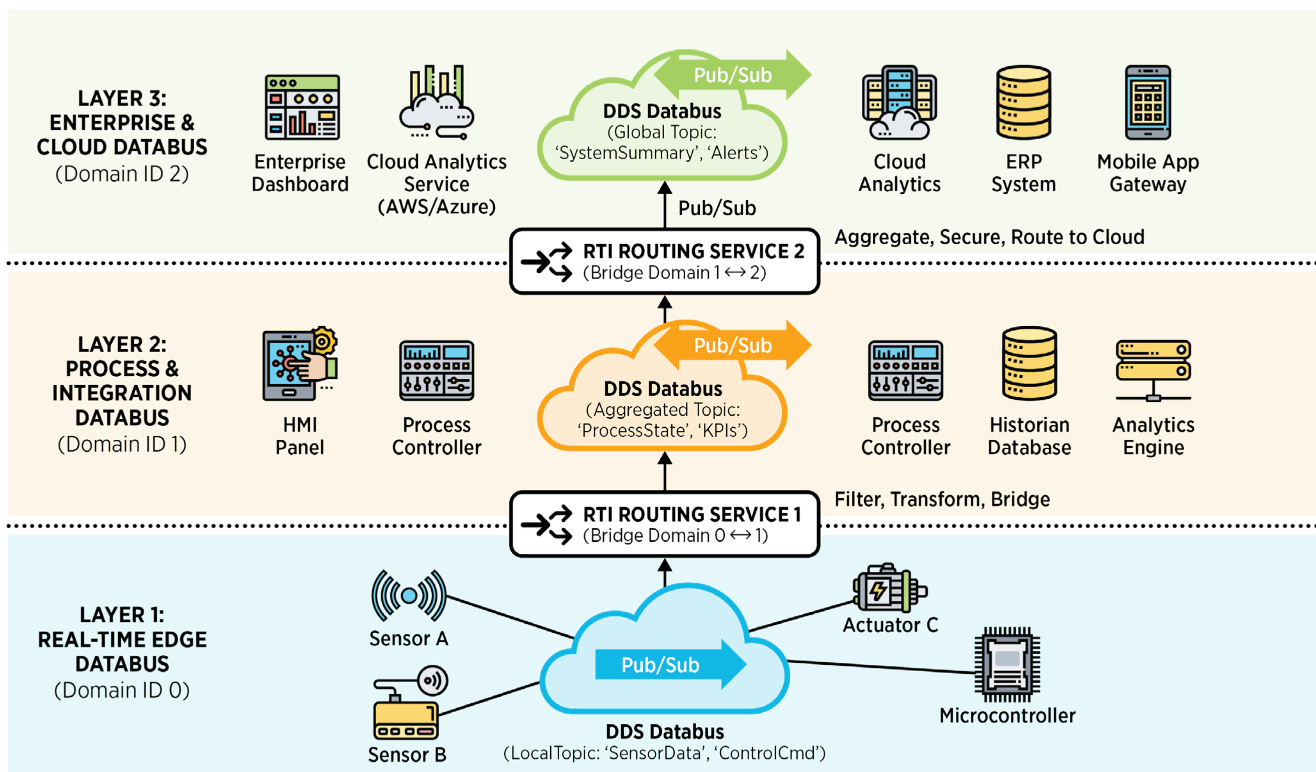

Figure 3: RTI Connext Layered Databus Architecture, including RTI Routing Service

Bridging the Layers

Because only some data must travel between layers, you should use DDS domains, partitions, or multicast groups to maintain boundaries. Carefully selecting which topics to share limits data transfer and keeps critical and non-critical data apart.

RTI Routing Service typically bridges these layers. It functions as a gateway, forwarding only selected topics between layers. It can also filter and transform data as it flows along its routes. Routing Service can also be used to adapt data between DDS and non-DDS transports which enables integration of external systems.

Safety Assurance

Safety requirements often stem from regulatory compliance, but they also reflect a commitment to protecting people and property. Examples of safety examples include DO-178C for avionics, ISO 26262 for automotive, and IEC 61508 for industrial systems. They generally demand a rigorous level of assurance. To meet the requirements of these standards, you must address several core requirements:

- **Perform** risk assessments and requirements-based development.
- **Implement** fault mitigation strategies.
- **Verify and validate** functions.
- **Maintain** traceability between requirements, code, and test procedures.
- **Enforce** independence between developers and verification teams.
- **Execute** structural coverage testing of code.
- **Ensure** quality through development processes and audits.

RTI Connext® Cert offers a certified solution that has been successfully deployed in systems requiring safety certification.

Security

Autonomous decisions rely on data integrity because modified or bogus data can lead to serious consequences. Attackers may also attempt to hijack the system to manipulate its operations maliciously. Therefore, it is important to understand your security requirements early in the development process.

DDS provides a pluggable security framework that secures communication from the data-centric perspective. You can tailor the level of protection for each data flow to balance security, ease of use, and performance:

- **Authentication:** Uses public-key cryptography (PKI) to ensure only authorized applications and devices join the DDS domain.
- **Access Control:** Provides fine-grained control to specify which participants can publish or subscribe to specific topics, enforcing “need-to-know” security.
- **Confidentiality:** Protects data in transit by encrypting DDS messages. You can configure encryption for specific topics as needed.
- **Integrity:** Uses Message Authentication Codes (MACs) and digital signatures to prevent data tampering and forgery.
- **Logging and Auditing:** Logs security-relevant events to increase visibility and support compliance.

These features remain transport-agnostic, meaning you can apply them across shared memory, UDP, or TCP. These policies are managed and configured using QoS and through XML files which define governance and permissions settings.

Conclusion

Building a reliable autonomous system is an exercise in managing complexity, performance, safety, and security. As we have explored, success is not just about the algorithms that drive decision-making, but the underlying global dataspace that supports them. By prioritizing a data-centric model, eliminating single points of failure, and baking in safety and security from the start, developers can avoid common issues that often derail complex projects.

Utilizing a robust framework such as RTI Connext provides the tools, infrastructure, and capabilities necessary to transform massive amounts of data into safe, real-time actions, ensuring your autonomous system is not only functional but resilient enough for the challenges of the real world.

If you’re new to data-centric design or have questions, click here to get in touch with your local RTI team.

ABOUT RTI

RTI is the real-time data streaming company for intelligent distributed systems. RTI Connext® software is the critical nervous system for over 2,000 designs across Aerospace and Defense, MedTech, Automotive, and Robotics.

RTI, Real-Time Innovations and the phrases “RTI Runs a Smarter World” and “Your systems. Working as one,” are registered trademarks or trademarks of Real-Time Innovations, Inc.
All other trademarks used in this document are the property of their respective owners. ©2026 RTI. All rights reserved. TI-006 V1 0626

rti Your systems. Working as one.
CORPORATE HEADQUARTERS: 232 E. Java Drive, Sunnyvale, CA 94089 • +1 (408) 990-7400 • info@rti.com
rti.com rti.com/blog rti.com/podcast company/rti rti_software rtisoftware