Current and Emerging Techniques in Robotic Software Design for Industrial Deployment: ROS 2, DDS, and Software Architectures

Samuel Mbakara John
MSc Robotics and Autonomous Systems, Aston University, Birmingham, United Kingdom

ABSTRACT

Robotic software is now just as important as mechanical design, because it is the software that turns a robot into a dependable system in real work conditions. This report reviews the techniques that are most widely used today and the newer approaches that are shaping industrial robotics, with a strong focus on ROS 2 as a modern middleware choice. It explains how ROS 2 differs from earlier ROS versions through its DDS based communication, and why features such as Quality of Service settings matter when robots must be responsive, scalable, and reliable. The report also compares common architectural patterns used in robotic systems, including client server and publisher subscriber communication, alongside system level approaches such as sense plan act and layered control. These techniques are then discussed in the context of real industrial pressures such as tight latency requirements, large scale deployments, integration with older systems, downtime risk, security, and skills gaps. Finally, the report proposes practical directions that can reduce these barriers, including hybrid communication designs, edge computing for faster decision making, AI assisted monitoring and tuning, and the use of digital twins to test updates safely before deployment. Overall, the report highlights that no single method solves every problem, but careful architecture choices and deployment focused practices can make robotic software more robust and easier to scale in modern industrial environments.

Keywords

Robotic software design, ROS 2, DDS, Layered architecture, Sense plan act, Real time robotics, Industrial deployment, Edge computing, Digital twin, Scalable robotics systems

1. Introduction

Robots are no longer defined only by their mechanics. What separates a reliable industrial robot from a lab prototype is the software that connects sensors, decision making, and actuation in a way that is fast, safe, and repeatable. In real industrial environments, robots must cope with noisy sensor data, unpredictable changes, tight timing constraints, and constant pressure to minimise downtime. As robotics grows across manufacturing, logistics, inspection, and autonomous navigation, software design has become the core factor that determines whether a system can scale beyond a single demonstration and operate as part of a wider production workflow.

This report reviews the main techniques used in robotic software design today and the newer approaches that are increasingly important for industrial deployment. The discussion centres on ROS 2 as a modern middleware framework, explaining why its DDS based communication model, Quality of Service tuning, lifecycle management, and real time capable execution make it attractive for industry. Rather than treating ROS 2 as a toolbox, the report focuses on how design choices inside ROS 2 influence real outcomes such as latency, reliability, fault recovery, and maintainability. It also examines key communication and system architecture patterns used in robotics, including client server, publisher subscriber, and ROS 2 actions, alongside system level approaches such as sense plan act and layered architectures. To reflect current practice, the report also considers modern task coordination methods such as behaviour trees and state machines, which are widely used in industrial and mobile robot stacks to support safer sequencing and better recovery behaviour.

The report then connects these techniques to the challenges that appear when robots move from controlled testing to industrial use. These challenges include achieving consistent low latency performance on limited hardware, scaling communication across many distributed nodes, integrating with legacy ROS 1 or proprietary systems, improving fault tolerance and restart behaviour, meeting security and data integrity expectations, and addressing skills gaps that slow adoption. In addition, the report highlights two practical concerns that often decide success in industry: software verification and testing (including simulation, hardware in the loop testing, and logging for traceability), and safety expectations where predictable failure behaviour is essential.

Finally, the report proposes deployment focused improvements that can reduce these barriers in realistic ways. These include hybrid communication designs that mix topics with services

and actions for safety critical control, the use of edge computing to bring time sensitive processing closer to the robot, AI supported monitoring to detect anomalies and guide parameter tuning, and digital twin workflows that allow teams to test updates, validate behaviour, and reduce operational risk before changes reach the real robot. The analysis is informed by EI4ROS lecture material and supported by recent scholarly literature, with the overall goal of providing clear guidance on how current and emerging techniques can be combined to produce robotic software that is dependable, scalable, and ready for industrial use.

2. Robotic Software Design Techniques

2.1. Robot Middleware: ROS 2

Robot middleware sits between the operating system and the application logic, so developers can focus on behaviour and control rather than low level hardware details [1]. In practice, it provides the building blocks for communication, device integration, and system coordination, which is why middleware choices often decide how quickly a robotics project can move from prototype to deployment.

ROS 2 is widely used as a modern middleware framework because it was designed with scalability, reliability, and industry needs in mind. A major change from ROS 1 is that ROS 2 does not depend on a central master. Instead, it uses the Data Distribution Service (DDS), which supports distributed discovery and efficient message passing across networks [2]. This shift makes ROS 2 more suitable for multi robot systems, factory floors, and deployments where machines must keep running even when parts of the system restart or move across networks. ROS 2 also supports multiple operating systems such as Ubuntu, Windows, and macOS, which helps organisations integrate robotics software into existing IT environments rather than forcing everything into one platform.

One of the most practical strengths of ROS 2 is how it supports both performance and control over communication behaviour. Developers can write nodes in C++ using rclcpp or in Python using rclpy, which makes it easier to combine fast control loops with higher level logic in the same system. More importantly, ROS 2 allows Quality of Service settings that let teams choose how messages behave under real conditions. For example, a control command may need high reliability, while a high rate sensor stream may prioritise fresh data over perfect delivery. These choices can improve determinism and stability, which matters in automation where timing issues quickly become safety and productivity issues.

That said, ROS 2 comes with trade-offs. DDS introduces a layer of concepts that can feel heavy at first, especially for teams moving from ROS 1 or from traditional PLC style automation. QoS is powerful, but it is also easy to misconfigure. In real deployments, poor QoS choices can lead to dropped messages, unstable timing, or systems that work in the lab but fail on a noisy network. Another challenge is integration with older ROS 1 systems. As ROS 1 has reached end of life, many industrial users face the reality of mixed fleets and legacy tooling, where bridging and gradual migration become necessary rather than optional [2]. Finally, even though ROS 2 supports real time capable behaviour, getting consistent low latency performance still depends on hardware, operating system configuration, and careful software design, especially on smaller embedded platforms.

In short, ROS 2 offers a strong foundation for industrial robotics, but it rewards teams that treat configuration and system design as engineering work, not as afterthoughts.

2.2 Architectural Styles

Robotic systems rarely fail because of one bad line of code. They fail because of poor structure: the wrong communication pattern for the job, unclear ownership of responsibilities, or designs that do not handle delays and faults well. Two communication styles show up repeatedly in robotics because they map closely to what robots actually do: request driven actions and continuous data flow.

2.2.1 Client–Server Architecture

In a client–server setup, one component asks for a service and another component responds. In robotics, this pattern is common for commands that need a clear answer, such as “reset localisation,” “load a configuration,” “start a calibration routine,” or “turn on a safety mode.” It fits well with ROS 2 services and also with ROS 2 actions when a task takes time and needs feedback, such as navigation goals or manipulation tasks [3].

The biggest advantage is clarity. You know who requested something, who responded, and whether it succeeded. That makes it easier to debug and easier to build predictable behaviours for important operations. The downside is that client–server communication can block if the server is slow, down, or overloaded. It is not always accurate to describe this as deadlock, but in practice it can feel like the system has frozen when timeouts and fallback behaviour are not designed properly. For industrial use, the real challenge is fault tolerance. A good client–server design needs timeouts, retries, and safe fallback actions so the robot can degrade gracefully instead of stopping abruptly. Latency is another concern when requests happen too

frequently. If a system sends rapid service calls for something that should have been streamed, performance can suffer and behaviour can become unstable.

So client–server is best used for well defined tasks, configuration changes, and safety critical commands where you want a firm acknowledgement that something happened.

2.2.2 Publisher–Subscriber Architecture

Publisher–subscriber communication is built for continuous information flow. A publisher sends messages to a topic, and any subscriber can listen without the publisher needing to know who they are [3]. This is why it is the natural choice for sensor data, state estimates, robot status messages, and anything else that is produced continuously and consumed by more than one part of the system.

The main benefit is flexibility. You can add or remove subscribers without redesigning the whole system, which helps a lot as systems grow. It also scales well across multiple nodes and multiple robots, especially when combined with ROS 2 QoS controls that can prioritise either reliability or freshness depending on what the topic represents. However, pub–sub systems can become messy if topics are not managed carefully. Too many topics, unclear naming, and inconsistent message definitions can make large systems hard to understand and maintain. Data overload is another real issue. High frequency publishing can saturate bandwidth or overwhelm subscribers if rates are not chosen wisely.

In industrial settings, the biggest challenges are keeping data consistent across distributed machines and controlling network load at scale. This is exactly where QoS becomes important again, because the system needs to be tuned so critical topics stay dependable while less important streams do not flood the network.

Publisher–subscriber works best for sensor and state streaming, while client–server or actions are better for commands and goal based tasks. Most real robots combine them, because no single pattern covers everything well.

2.3 Robot Architectures

Robotic systems need more than good algorithms and fast messaging. They need a decision structure that explains, in a disciplined way, how raw sensor signals become reliable actions. When that structure is weak, the robot may still work in a demo, but it becomes fragile in real use: small timing changes cause unstable behaviour, updates break unexpected parts of the system, and faults are difficult to diagnose. A well chosen architecture reduces these risks by

defining clear responsibilities, predictable information flow, and safe behaviour when parts of the system degrade. Two influential approaches that continue to shape modern practice are sense plan act and layered control.

2.3.1 Sense–Plan–Act Architecture

Sense plan act (SPA) follows a deliberate pipeline. The robot observes the environment through sensors, forms an internal representation of what is happening, chooses a plan that moves it toward a goal, and then executes that plan through control actions [4]. This structure remains valuable because it encourages careful reasoning about goals, constraints, and sequencing. In industrial robotics, it fits situations where the workcell is controlled and the task is predictable, such as scheduled inspection routes, repetitive manipulation, or motion plans in spaces with clearly defined boundaries.

SPA's main strength is transparency. Each stage can be tested and verified in isolation: sensing can be validated against known ground truth, planning can be checked for constraint satisfaction, and execution can be measured for tracking error and stability. That separation is attractive in industry because it supports traceability and structured debugging. When failures happen, engineers can identify whether the issue is perception, planning logic, or control tracking rather than treating the system as a black box.

The weakness of SPA appears when timing becomes unforgiving. Industrial environments are rarely static. Workpieces shift, people enter shared spaces, sensors produce noise, and communication delays vary. If the robot spends too long building a world model and generating a plan, the plan can become stale before it is applied. The result is a cycle of frequent replanning, stop start motion, or behaviour that looks hesitant under uncertainty. SPA can also become computationally heavy because the world model and planner must be refreshed regularly, especially when the robot must handle many moving elements.

For deployment, the key challenge is keeping the benefits of deliberate planning while meeting real time responsiveness. That is why modern systems rarely use SPA as a single uninterrupted pipeline. They keep planning where it adds value, but enforce fast safety and stabilisation behaviours that can act immediately when the environment changes. In practice, SPA becomes one layer within a broader design rather than the only structure that governs behaviour.

2.3.2 Layered Architecture

Layered architecture addresses these realities by separating decision making into levels, usually a fast reactive layer and a slower deliberative layer [5]. The reactive layer is responsible for time critical functions such as actuator control, collision checks, limit enforcement, and immediate recovery actions. The deliberative layer focuses on longer horizon decisions such as task sequencing, route selection, strategy choice, and parameter selection for different operating modes. This separation reflects an important engineering truth: a robot must remain safe and stable even when higher level planning is uncertain, delayed, or restarting.

A strong advantage of layering is robustness. Low level control can continue to hold position, maintain safe speeds, or trigger a stop even if the planning layer is busy or temporarily unavailable. Layering also supports clean interfaces, which makes the system easier to extend. Teams can change planning behaviour without rewriting the servo control loop, and they can refine low level control without touching high level task logic. This improves maintainability, which is often a bigger cost driver than initial development in industrial robotics.

The trade-offs are coordination and complexity at the boundaries. Layered systems fail when layers disagree or when responsibilities overlap. A typical problem is conflicting commands, where high level logic requests an action that violates constraints enforced below. Another issue is parameter tuning across layers. Gains, thresholds, timing windows, and safety margins interact, and small changes can cause large effects if tuning is done without a clear structure. Industrial deployments also demand predictable behaviour during mode changes, such as switching from automatic to collaborative mode, or from normal operation to recovery mode. If the handoff between layers is not carefully designed, transitions can become the moment where instability or unsafe behaviour appears.

Layering is widely used in mobile robots and collaborative systems because it supports the inclusion of dedicated safety behaviour close to the actuators, where it can be enforced without delay. This is valuable in human robot interaction, where the robot must respect speed limits, safety zones, and emergency stops regardless of what the higher level software is attempting to do.

Modern robotic systems often blend deliberate planning with layered control. Planning is used for goal driven reasoning, constraint handling, and efficient motion generation. Reactive layers handle fast stabilisation, safety enforcement, and short horizon adaptation. This

combination is what allows robots to remain purposeful while still behaving reliably when the environment becomes unpredictable.

3. Challenges in Industrial Environments

Industrial robotics is less forgiving than research or classroom prototypes because the software is judged by uptime, safety, and consistency rather than by a single successful demonstration. A robot deployed on a factory floor or in a warehouse must behave predictably across long shifts, recover cleanly from faults, and operate safely around people, equipment, and changing work conditions. These constraints expose weaknesses that may not appear during development, especially when systems are built from many distributed components communicating across networks.

Real time performance is one of the hardest requirements to meet in practice. Industrial tasks such as assembly, navigation, or coordinated manipulation depend on low latency responses and consistent timing, not just average speed. ROS 2 provides tools that support real time capable behaviour, but achieving reliable timing still depends on careful system design, operating system configuration, thread scheduling, and disciplined handling of callbacks and message queues [2]. On resource constrained hardware, small delays from logging, memory allocation, or poorly tuned Quality of Service settings can accumulate and show up as jitter, late control updates, or unstable control loops. The challenge is rarely that the software cannot run fast once, but that it cannot stay fast and predictable under load.

Scalability creates a second layer of pressure. Industrial deployments often involve many robots, many sensors, and many nodes running across separate machines. Publisher subscriber communication supports this distributed style well, but it can also become a bottleneck if message rates, topic design, and network usage are not controlled. High frequency streams such as vision, lidar, or state feedback can saturate bandwidth, causing delays for critical messages. As systems grow, teams must treat communication design as part of the architecture, deciding what must be reliable, what must be fresh, what can be dropped, and how to stop non critical data from starving safety critical traffic.

Interoperability remains a major barrier because industry rarely starts from a clean slate. Many organisations already have ROS 1 based systems, PLCs, proprietary controllers, or older communication stacks that still work and cannot be replaced overnight. Integrating these legacy systems with ROS 2 adds complexity at both the technical level and the process level, especially as ROS 1 has reached end of life, which increases the urgency of migration

while also increasing risk [2]. Bridging is possible, but it introduces extra points of failure, version mismatches, and operational overhead. A realistic architecture must plan for gradual migration, mixed fleets, and long term maintainability rather than assuming a full reset.

Reliability and fault tolerance matter because downtime is expensive and safety incidents are unacceptable. Industrial software must handle component restarts, temporary network disruption, sensor dropouts, and degraded modes without collapsing into unsafe behaviour. Request response designs can fail badly if services block without timeouts or if fallback actions are not defined. The problem is not only that the robot stops; the problem is that it stops unpredictably, or worse, continues moving without the right checks. Practical reliability requires timeouts, watchdogs, health monitoring, clear recovery behaviour, and safe state transitions so that failures become controlled events rather than system wide breakdowns.

Security and data integrity are also now unavoidable. Robots are increasingly connected to factory networks, cloud dashboards, and remote maintenance tools, which expands the attack surface. Protecting communication channels and preventing unauthorised control is essential when robots can affect physical safety and production continuity. ROS 2 can support secure communication through DDS security features, but configuration and key management can be complex, and the security model must be aligned with organisational IT policies and operational realities [6]. In many deployments, the challenge is not recognising the need for security, but implementing it without breaking performance, usability, or maintainability.

A final barrier is the skills gap. Industrial teams often include strong mechanical and process engineers, but modern robotics software requires comfort with distributed systems, middleware configuration, debugging tools, and performance tuning. ROS 2 is powerful, yet its flexibility means there are many ways to configure it incorrectly. Without the right skills, teams may struggle with QoS tuning, lifecycle management, deployment automation, and diagnosing latency or network faults, which slows adoption and increases the chance of fragile systems reaching production [6]. Closing this gap usually requires better tooling, clearer defaults, and development practices that make systems easier to operate and maintain, not just easier to build the first time.

4. Innovative Solutions

The challenges discussed in the previous section do not have a single “magic” fix, because most industrial robotics problems sit at the boundary between software design, networking, operations, and safety. What does work is a set of practical design strategies that reduce

failure points, protect timing, and make systems easier to operate at scale. The approaches below are realistic within ROS 2 based deployments and focus on decisions that have clear impact on latency, reliability, and maintainability.

4.1 Hybrid Architectures

A useful way to design industrial robot software is to stop treating communication patterns as competing choices and start using them as tools for different jobs. Many failures come from using one pattern everywhere. For example, using services repeatedly for high rate control traffic can introduce blocking and latency, while using only topics for safety critical commands can make it harder to confirm that an action was received and applied.

A hybrid architecture combines publisher subscriber topics for continuous data flow with services and actions for commands and goal based tasks. Sensor streams, state estimates, and robot health data fit naturally as topics, where multiple consumers can subscribe without coupling the system. Critical commands and mode changes are better handled with services or actions because they support acknowledgement, timeouts, and clear success or failure reporting. In ROS 2, this design can be strengthened by pairing commands with watchdog behaviour, so that if acknowledgements do not arrive within a defined window the robot falls back to a safe state. This approach supports fault tolerance because a failed node does not silently leave the robot in an unknown condition, and it supports scalability because high volume data stays in the pub sub space rather than overwhelming request response paths [2].

4.2 Edge Computing Integration

Edge computing helps industrial robotics because it places time sensitive computation near the robot, instead of pushing everything into the cloud or a distant server. The simplest way to think about it is separation by timing. Functions that must respond quickly, such as control loops, obstacle avoidance, sensor fusion, and safety checks, belong on the robot or on a local edge node with predictable latency. Functions that can tolerate delay, such as reporting dashboards, long term optimisation, fleet analytics, and some scheduling logic, can run on higher level servers.

In a ROS 2 system, an edge approach usually means running the core real time nodes on the robot and using edge hardware for heavier perception workloads, such as vision inference, mapping, or high density point cloud processing. The results are then published back to the robot as compact messages rather than raw data streams. This reduces bandwidth, improves timing stability, and makes large deployments easier to scale because the network is not

flooded with unnecessary high rate payloads. It also improves resilience because the robot can keep operating safely even if the cloud connection is slow or temporarily unavailable [6].

4.3 AI Driven Middleware Support

AI is often mentioned as a solution, but it becomes valuable only when it is applied to specific operational problems. In industrial robotics, the most practical use is not "AI making decisions for the robot," but AI helping engineers keep the system stable and healthy. Machine learning can support anomaly detection from logs and telemetry, flagging early warning signs such as rising latency, dropped messages, CPU saturation, memory growth, repeated node restarts, or communication patterns that usually precede failure. This is especially useful in distributed ROS 2 deployments, where faults may look like random behaviour unless they are tracked across time.

AI can also support smarter configuration. QoS tuning, for instance, is one of the most common reasons ROS 2 systems behave inconsistently across environments. A data driven approach can recommend QoS profiles based on observed network conditions and message rates, reducing trial and error. For request response patterns, predictive monitoring can detect when service response times are drifting upward and trigger mitigation such as rate limiting, rerouting to backup services, or switching the robot into a safe reduced capability mode rather than letting performance collapse. Used this way, AI becomes a reliability tool that supports operations, not a risky replacement for deterministic control logic [6].

4.4 Digital Twin Integration

Digital twins offer a strong safety and testing advantage because they give engineers a realistic environment for evaluation before code touches production hardware. A digital twin is more than a simple simulation. It is a maintained virtual replica of the robot and its working context that can be fed with real ROS 2 data streams and used to replay scenarios, test updates, and study failure modes [7]. This is particularly valuable in industry because updates that look harmless can introduce timing changes, new dependencies, or subtle behaviour shifts that only appear under real workloads.

When integrated properly, the digital twin becomes part of the development and deployment workflow. Teams can validate new control parameters, new QoS settings, or new recovery logic using recorded data, then test the same changes in the twin under stress conditions before deployment. This reduces commissioning time, improves traceability, and supports safer operations, especially for collaborative robots where the cost of unexpected behaviour is

high. Digital twins also support predictive maintenance by comparing expected behaviour with live behaviour and highlighting drift, wear, or sensor degradation earlier than manual inspection would catch [7].

4. Evaluation and Comparison

The table below compares the discussed techniques based on key criteria:

Criterion	ROS 2	Client-Server	Pub-Sub	SPA/Layered
Scalability	High	Low	High	Medium
Real Time Capability	High	High	Medium	Low
Reliability	High	Low	High	Medium
Complexity	High	Medium	High	Medium
Industrial Suitability	High	Medium	High	Medium

ROS 2 and publisher subscriber systems are a strong fit for industrial robotics because they scale well and support dependable communication when they are configured properly. The main drawback is not capability, but complexity. Teams often need solid experience with DDS concepts, Quality of Service tuning, and distributed debugging to get consistent performance across different networks and hardware. Client server patterns still have a place because they give clear request and response behaviour, which is useful for commands and configuration tasks, but they can become fragile if timeouts, retries, and fallback behaviour are not designed in from the start.

Sense plan act and layered designs continue to matter, but mostly as parts of modern hybrid systems. Sense plan act is valuable when tasks are structured and planning needs to be logical and traceable, while layered control improves responsiveness and safety by keeping time critical behaviour close to the actuators. The most practical direction for industry is not choosing one architecture and sticking to it, but combining the strengths of each approach so that planning remains goal driven while control and safety remain fast and stable. Looking ahead, many industrial deployments are likely to lean toward hybrid models that use ROS 2 for flexible integration, edge computing for low latency processing, and data driven monitoring to support better fault detection and tuning.

5. Conclusion

Robotic software is moving from research style prototypes toward production grade systems that must run safely and predictably in demanding environments. ROS 2 has accelerated this shift by offering a modern middleware foundation that supports distributed communication, scalable node design, and more control over message behaviour through DDS and QoS.

Publisher subscriber patterns remain essential for sensor and state data, while client server and actions support clear command handling and goal based tasks. At the same time, industrial settings still expose difficult problems, especially integration with older systems, timing stability under load, downtime risk, and secure deployment across connected networks.

The approaches discussed in this report point toward practical ways to reduce these gaps. Hybrid communication designs help match the right message pattern to the right job and improve fault handling. Edge computing reduces latency and network pressure by keeping critical processing close to the robot. AI supported monitoring is most useful when it focuses on reliability, such as spotting anomalies early and guiding configuration choices rather than replacing deterministic control. Digital twins strengthen testing and safety by allowing teams to validate updates before they reach production hardware. With these practices in place, robotic software can become easier to scale, easier to maintain, and more trustworthy in real industrial use, which is the standard modern deployments are now expected to meet.

References

[1] Elkady, A., & Sobh, T. (2012). Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012, Article ID 959013. https://doi.org/10.1155/2012/959013

[2] Maruyama, Y., Kato, S. & Azumi, T. (2016). *Exploring the Performance of ROS 2*. In Proceedings of EMSOFT '16, Pittsburgh, PA, USA. Available at: https://web.ics.purdue.edu/~rvoyles/Classes/ROS_MFET642/Maruyama.ExploringROS2.2016.pdf (Accessed: 16 October 2025).

[3] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R. & Ng, A. (2009). *ROS: an open-source Robot Operating System*. Available at: http://ai.stanford.edu/~mquigley/papers/icra2009-ros.pdf (Accessed: 16 October 2025)

[4] Nilsson, N.J. (1984) *Shakey the Robot*. SRI International Technical Report, No. 323. Available at: https://www.sri.com/wp-content/uploads/2021/12/629.pdf (Accessed: 16 October 2025).

[5] Brooks, R.A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation,* RA-2(1). Available at: https://www.robolabo.etsit.upm.es/asignaturas/irin/papers/Brooks86RobustLayeredControl.pdf (Accessed: 16 October 2025).

[6] Arul Kumar, P.V. (2022) Future robotics system design challenges including software engineering automation – a report. *Journal of Electrical Engineering and Automation*, 4(3), pp. 175–186. DOI: https://doi.org/10.36548/jeea.2022.3.005

[7] Gambo, M.L., Danasabe, A., Almadani, B., Aliyu, F., Aliyu, A. & Al-Nahari, E. (2025) 'A Systematic Literature Review of DDS Middleware in Robotic Systems', *Robotics*, 14(5), article 63. DOI: https://doi.org/10.3390/robotics14050063