From Architecture to Implementation: Robotic Data Pipelines for Digital Agriculture GIS (Application Paper)

Piotr Skrzypczyński[1,*], Filip Baranowski[1], Krzysztof Ćwian[1], Maciej Krupka[1], Jakub Pilarski[1], Antoni Sopata[1] and Robert Wrembel[1]

[1]Poznan University of Technology, Poznań, Poland

Abstract

The integration of mobile robotic platforms with Geographic Information Systems (GIS) is a key enabler of data-driven digital agriculture. While recent research has proposed architectures for integrating heterogeneous robotic and sensor data with GIS tools in cloud–edge environments, their practical realisation using existing data engineering technologies remains underexplored. This paper reports on the development of actual data processing pipelines compliant with the GIS4IoRT architecture, developed within our CHIST-ERA project. Focusing on precision agriculture use cases, we study representative spatio-temporal queries using real data collected from field robots and implement them using three distinct data integration approaches that differ in a programming model, system architecture, and execution paradigm. We analyse design choices, implementation effort, and system limitations, highlighting the trade-offs between the approaches. The results offer practical guidance on instantiating the GIS4IoRT architecture and on designing robotic–GIS data integration pipelines.

Keywords

data integration, GIS, robotics, spatio-temporal stream processing, edge–cloud data processing, sustainable agriculture

1. Introduction and motivation

Digital agriculture increasingly relies on mobile robots and sensor networks to monitor crops, soil conditions, and farming operations at high spatial and temporal resolution [1]. Ground and aerial robots equipped with GNSS, cameras, LiDAR, and environmental sensors continuously produce large volumes of heterogeneous data streams that must be processed, integrated, and analysed in near-real-time [2]. Geographic Information Systems (GIS) play a central role by providing spatial reference data, visualisation, and spatial analytics required for decision support in precision agriculture [3].

Integrating robotic data with GIS tools remains challenging. Robotic platforms typically rely on specialised middleware such as ROS 2 [4], custom data formats, and streaming communication models, whereas GIS systems have traditionally been designed for static or slowly evolving spatial datasets [5]. Bridging this gap requires architectures capable of handling multi-modal spatio-temporal data streams, tolerating intermittent connectivity, and supporting both continuous and historical queries [6].

Several stream processing architectures have been proposed for IoT and spatio-temporal data analytics, including Lambda-style designs [7] and cloud-based streaming platforms built on engines such as Apache Storm [8] and Kafka [9]. However, their integration with GIS environments is typically not discussed. These technologies offer very limited support for GIS-native spatial representations, geofencing semantics, or unified handling of continuous and historical spatial queries. This highlights the need for architectures that explicitly bridge stream processing systems and GIS-oriented data management.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2026 Joint Conference (March 24-27, 2026), Tampere, Finland

*Corresponding author.

piotr.skrzypczynski@put.poznan.pl (P. Skrzypczyński); filip.baranowski@student.put.poznan.pl (F. Baranowski); krzysztof.cwian@put.poznan.pl (K. Ćwian); maciej.krupka@put.poznan.pl (M. Krupka); jakub.pilarski.2@student.put.poznan.pl (J. Pilarski); antoni.sopata@student.put.poznan.pl (A. Sopata); robert.wrembel@put.poznan.pl (R. Wrembel)

0000-0002-9843-2404 (P. Skrzypczyński); 0000-0001-6037-5718 (R. Wrembel)

To the best of our knowledge, there is no work on building and comparing integration architectures for data produced by the Internet of Robotic Things (IoRT) and their orchestration with GIS tools. For this reason, the CHIST-ERA GIS4IoRT project addresses these challenges by proposing a plug-and-play, cloud-based middleware architecture for integrating data produced by the IoRT with GIS environments (see [10]). The architecture combines edge–fog–cloud computing with a mediated data integration and querying layer, Quality of Service (QoS) and Quality of Data (QoD) aware query execution, and compliance with GIS standards.

While the GIS4IoRT architecture specifies the required functionality, its practical realisation using existing data processing frameworks remains largely unexplored. In particular, it is unclear how different implementation choices affect system complexity, flexibility, and performance [11]. Addressing this gap is essential for translating architectural designs into practical systems.

Our previous corresponding paper [12] introduced the architecture, identified key research challenges, and outlined its applicability to several domains, including precision agriculture. This paper adopts an implementation-oriented perspective. We demonstrate how the GIS4IoRT architecture can be instantiated through three alternative processing pipelines based on different programming models and execution paradigms, namely:

1. a declarative SQL-based streaming approach built on a messaging backbone;
2. a low-level distributed stream processing framework with explicit state management and spatial indexing;
3. an IoRT-oriented edge–cloud stream processing system designed for execution across heterogeneous devices.

To ensure comparability, all three processing pipelines implement the same core functionality, centred on three representative queries derived from real precision agriculture scenarios:

1. **geofencing query**: determines whether a robot is outside a specified plot, evaluated as a continuous query with periodic updates (1a), and as a historical query over persistently stored data (1b);
2. **collision detection query**: detects potential spatial conflicts between robots based on their relative positions over time;
3. **IoT sensor proximity query**: raises an alarm when a robot is within a specified distance from a sensor and the sensor reading exceeds a defined threshold.

All queries combine real robot odometry data obtained from ROS file recordings (called rosbags) with static plot geometries stored in GIS-compatible formats (PostGIS). The data reflect real conditions encountered in agricultural field deployments in France.

The novelty of this paper lies in: (1) providing end-to-end implementations of spatio-temporal robotic–GIS queries on real data, (2) systematically comparing three different processing paradigms under a common architectural framework, and (3) analyzing practical trade-offs in terms of development effort, flexibility, and suitability for GIS4IoRT-style deployments.

The remainder of the paper is organized as follows. Section 2 briefly recalls the GIS4IoRT architecture as the backbone of our experiments. Section 3 presents the three implementation frameworks and their design choices. Section 4 reports experimental results and qualitative comparisons based on real agricultural robotic data. Section 5 concludes the paper and outlines directions for future work.

2. Architecture

This paper builds on the GIS4IoRT architecture previously introduced as part of a CHIST-ERA project [12]. This section provides a condensed view of its key requirements and functional components, with a particular emphasis on those elements that are instantiated in the implementations described in this paper.

2.1. Goals and requirements

The GIS4IoRT architecture targets dynamic, heterogeneous, and spatio-temporal data processing, produced by mobile robots operating in environments such as agricultural fields. The requirements relevant to this study are as follows:

- heterogeneous stream integration: ingestion and correlation of robotic telemetry streams with static GIS datasets, including trajectories and spatial geometries [6];
- spatio-temporal query processing: support for spatial predicates combined with temporal semantics, enabling continuous real-time and historical queries [11];
- streaming and replay-based execution: low-latency processing of live streams and retrospective analysis via stream replay or batch execution;
- dynamic deployment: flexible query configuration in the presence of mobile data sources and changing connectivity [13];
- GIS-compatible outputs: emission of results in standard spatial formats consumable by GIS applications.

These requirements directly inform the architectural layering and underpin the choice of processing frameworks compared in Section 3.

2.2. Overview of the GIS4IoRT architecture

At a high level, the GIS4IoRT architecture follows a mediated data integration model [14, 15] deployed across an edge–fog–cloud continuum [6]. As illustrated in Fig. 1, the architecture consists of four main conceptual layers:

- IoRT data sources: mobile and stationary devices, including ground and aerial robots, sensors, cameras, and LiDARs. In this work, robots run ROS 2 and produce time-stamped localisation streams (e.g., `nav_msgs/Odometry` or `sensor_msgs/NavSatFix`) representing spatial trajectories. At the same time, these data are uploaded into a central repository that stores also metadata and ontologies for mapping data from multiple IoRT, i.e., data in different modalities.
- *Data integration and querying layer*: abstracts heterogeneous data sources and enables unified spatio-temporal query processing by translating high-level queries into executable operations on underlying processing engines [11], which are IoRT devices and a repository of historical static data.
- GIS4IoRT middleware: provides orchestration, data routing, and caching across the infrastructure, supporting spatio-temporal queries, QoS/QoD-aware execution, and transformation of results into GIS-compatible representations.
- GIS applications: issue queries, visualise results, and combine robotic data with external spatial datasets such as plot boundaries stored in GIS repositories.

Beyond data quality and real-time performance constraints, this architecture poses two significant challenges. First, the integration environment is highly dynamic. This dynamicity stems from: (1) the on-the-fly deployment of new IoRT devices and (2) intermittent connectivity, where limited or unavailable Wi-Fi causes mobile devices to temporarily disappear from the network. To manage this, the architecture must support automatic service discovery and registration, allowing devices to be incorporated immediately upon deployment or re-connection.

Second, the query mechanism must account for the intermittent availability of robotic data sources. For instance, while n devices may be present when a query is formulated, a few of them may disconnect before or when the query is executed. For this reason, the system must be capable of dynamic query routing to the remaining active devices. Furthermore, query results must be augmented with metadata describing their quality, specifically: (1) completeness metrics, indicating the percentage of missing data due to device unavailability, and (2) data fidelity indicators, reflecting quality degradation caused by network throughput limitations.

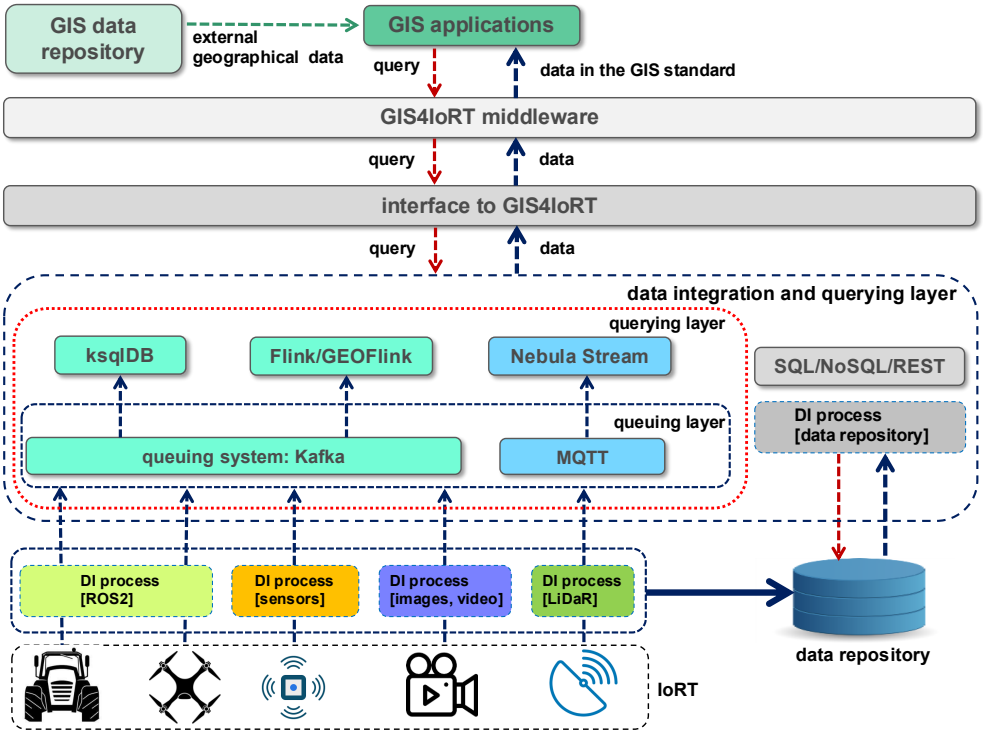

Figure 1: The GIS4IoRT architecture serving as the backbone of the CHIST-ERA project. IoRT data sources are ROS 2-enabled field robots replayed from rosbag files. The data integration and querying layer is instantiated by one of the three frameworks discussed in Section 3, while the middleware functionality is realized using Kafka or MQTT and lightweight REST/WebSocket APIs. Query results are consumed by GIS applications in GIS-compatible formats.

2.3. Architectural instantiation

The design, development, and experimental work presented in this paper instantiates the GIS4IoRT architecture using real agricultural robot data and three alternative implementations of the data integration and querying layer.

At the data source level, rosbag recordings are used to replay robot trajectories collected in real field deployments. These trajectories represent the dynamic data streams processed by the system. Static spatial data describing agricultural plots are provided in GIS-compatible formats (PostGIS geometry exported to CSV), serving as reference data for geofencing queries.

At the integration and processing level, the data integration and querying layer is instantiated using three alternative frameworks that differ in their execution model and system scope. These include: (1) a SQL-oriented streaming solution built on a messaging backbone, (2) a general-purpose distributed stream processing framework with native support for spatial indexing, and (3) an IoRT-focused stream processing system supporting distributed edge–cloud execution. Although all three frameworks realise semantically the same queries, they differ substantially in how data ingestion, state management, spatial evaluation, and windowing are handled.

The middleware functionality is implemented in a lightweight manner using messaging systems (Kafka or MQTT) and REST/WebSocket-based APIs [13]. These components handle configuration updates (e.g., plot definitions), dissemination of query results, and interaction with external clients or visualization tools. At the GIS application level, query results are produced in standard, GIS-friendly representations (e.g., JSON with spatial attributes), enabling straightforward integration with GIS tools and dashboards.

3. Implementation frameworks

This section presents three alternative implementations of the GIS4IoRT processing layer, each realizing the same functional requirements but using different tools, programming models, and execution paradigms [11]. All implementations process real robot data collected in agricultural fields and support all of the core queries introduced in Section 1, namely: a continuous geofencing query (denoted as *Query 1a*), a historical geofencing query (*Query 1b*), a collision detection query (*Query 2*), and an IoT sensor proximity query (*Query 3*).

The goal of this section is not to advocate a single "best" solution, but to demonstrate how different technological choices affect system design, flexibility, and operational complexity within the same architectural backbone.

3.1. Declarative stream processing with ksqlDB and Apache Kafka

The first approach adopts a declarative stream processing model, using Apache Kafka [16] as the messaging backbone and ksqlDB as the processing engine [11]. This design emphasizes simplicity, rapid development, and tight integration with event-driven architectures.

Figure 2 shows the data flow in which robot telemetry data is produced by a ROS 2–Kafka bridge that replays rosbag recordings and publishes odometry messages as JSON events into Kafka topics. Messages are keyed by robot identifier to enable partition-based parallelism [13]. Plot definitions and robot–plot assignments are managed through a control stream, allowing configuration updates to be applied dynamically without restarting the system.

Query 1a is implemented directly in ksqlDB using SQL-like continuous queries. Telemetry streams are joined with the configuration table and evaluated in 1-second tumbling windows. Since ksqlDB does not natively support spatial data types or operators, the point-in-polygon test is implemented as a custom User-Defined Function (UDF) based on a standard geometry library. If a robot is detected outside its assigned plot at least once within a window, an alert event is emitted to an output Kafka topic.

Query 1b is realized outside the streaming engine using batch processing. A Python-based tool parses the recorded ROS bag files, extracts odometry data, and applies the same geometric logic to identify all historical violations. While this results in a split streaming/batch architecture, it significantly simplifies implementation by reusing existing geospatial libraries.

Query 2 is implemented as an external Python program running alongside ksqlDB. This hybrid approach was necessitated by the limitations of the standard Kafka partitioning, where sharding by the robot identifier prevents the spatial co-locality required for collision detection. By comparing each new position with every other robot, the logic runs effectively with $O(N^2)$ complexity. The primary bottleneck of this implementation is its centralized state architecture. While it ensures accurate and fast collision detection for small fleets, it scales poorly for larger ones.

Query 3 is performed within ksqlDB utilizing the Broadcast Join strategy. Unlike the high-frequency robot collision scenario, the lower frequency of sensor updates permits replicating sensor messages across all partitions. This enables a local stream-stream join to pair robots with sensors. However, this logic is based on the regularity of sensor updates. Irregular intervals create temporal blind spots where telemetry may pass unchecked. Although the current approach prioritizes responsiveness, eliminating these gaps would require windowed aggregation, inherently introducing processing lag.

This approach fits well with the GIS4IoRT data integration and querying layer, offering moderate operational overhead along with high developer productivity. Its main limitations are the lack of native spatial abstractions and limited control over execution internals.

3.2. Distributed spatial stream processing with Apache Flink and GeoFlink

The second approach relies on Apache Flink [17] combined with the GeoFlink library [18], representing a low-level but highly scalable implementation of the GIS4IoRT processing layer [11]. Unlike the

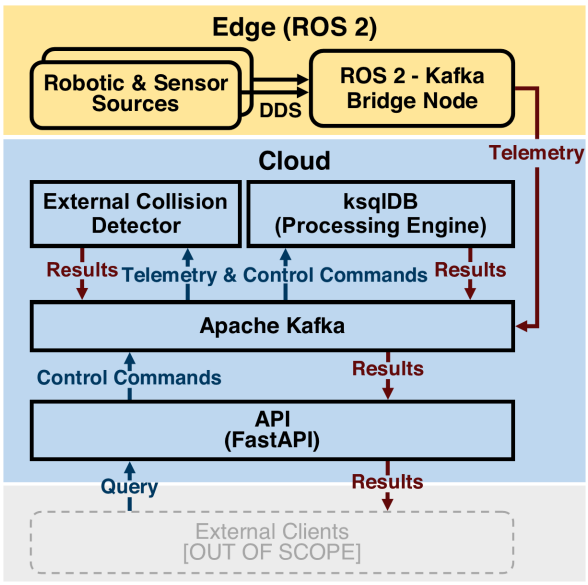

Figure 2: Data flow in the ksqlDB-based architecture.

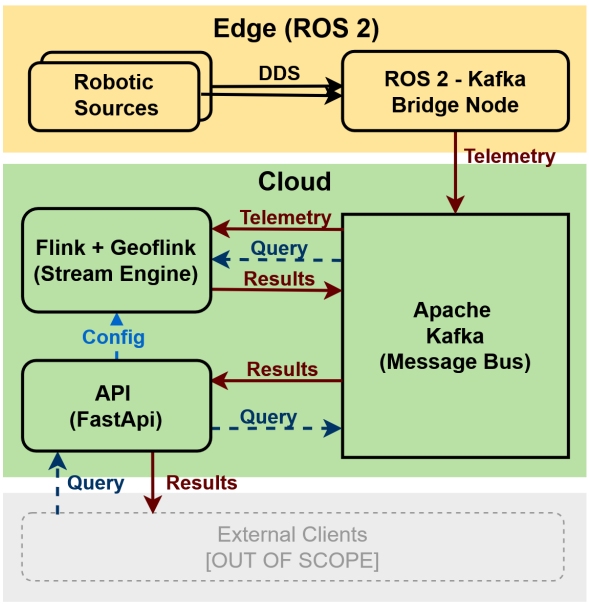

Figure 3: Data flow in the GeoFlink-based architecture.

declarative ksqlDB approach, this solution uses Flink DataStream API and explicit state management.

As shown in Fig. 3, robot telemetry and query configuration are ingested into the stream processing engine from Kafka topics. Configuration info is distributed to all processing nodes using a broadcast state pattern, ensuring local access to query rules. GeoFlink provides spatial data structures and indexing mechanisms, most notably a uniform grid index used to spatially partition incoming data [18]. This allows robot locations and complex plot definitions to be routed only to the relevant grid cells.

Query 1a is implemented as a stateful streaming operator. Incoming telemetry events are routed to grid cells, where local point-in-polygon checks are performed against the cached plot geometry. Results are aggregated in 1-second windows and emitted as alert events. The explicit spatial partitioning enables efficient scaling with increasing numbers of robots and higher data rates [11].

Query 1b can be implemented by replaying historical data through the same Flink job or by running a dedicated batch-style Flink pipeline, preserving a consistent processing model.

Query 2 is implemented without time windows to ensure immediate reaction. Instead, it uses a per-event processing model, where each incoming telemetry message is instantly compared against the cached positions of nearby robots routed to the same spatial partition. Cached positions are retained only for a limited duration (TTL) to prevent detection against obsolete data.

Query 3 combines real-time robot telemetry with environmental sensor readings. The status of each sensor, merging configuration data (e.g., thresholds) with the latest measurements, is maintained in the worker's local memory to track external conditions. This enables the system to trigger alerts based on the robot's proximity to a sensor and the simultaneous violation of specific safety limits.

This approach offers the greatest flexibility and performance potential, closely aligning with the scalability goals of GIS4IoRT. At the same time, it requires deep expertise in distributed stream processing and, in some cases, the implementation of custom spatial operators beyond those provided by GeoFlink.

3.3. Edge–cloud IoRT stream processing with NebulaStream

The third approach utilises NebulaStream [19], an end-to-end stream processing system designed specifically for IoT and IoRT environments spanning sensors, edge devices, and cloud infrastructure [6]. This framework is architecturally closest to the original GIS4IoRT vision of distributed, heterogeneous execution (Fig. 4).

In this setup, a NebulaStream Coordinator instance is initialised. The Coordinator is responsible for receiving, compiling, optimising, and dispatching partial query plans to NebulaStream Workers.

Multiple workers can be deployed across various machines, including edge devices. A worker is created for each robot or group of sensors, corresponding with a ROS 2-MQTT bridge to capture ROS 2 telemetry data. All queries are written in Java, contain Java UDFs [20], and are sent to the Coordinator via the REST API [6].

Query 1a is implemented as a continuous NebulaStream query that calculates each new robot's position and evaluates geofence violations. Plot geometries are provided in binary spatial formats. The query results are published to an MQTT topic and consumed by downstream services or visualisation components.

Since NebulaStream focuses primarily on streaming execution, *Query 1b* is emulated by replaying historical data through the same pipeline. While this preserves a unified execution model, it increases operational complexity.

Query 2 necessitates the real-time, simultaneous comparison of robot positions for every pair. Since data processing is strictly stream-based, there is no operator available to access the most recent historical data. This necessitates the use of a join operator that functions exclusively on tumbling windows. Consequently, the system must calculate average robot positions and subsequent distances within a 1-second window. This introduces additional processing latency due to the time required to aggregate all messages within that window.

Query 3 encounters a similar issue; to facilitate comparison, the maximum sensor value within a given window must be determined.

This approach demonstrates how GIS4IoRT-style queries can be deployed across edge–cloud infrastructures. However, it requires significant configuration effort, extensive use of UDFs, and struggles with the instantaneous comparison of distinct data streams. Furthermore, it currently lacks support for dynamically attaching new data sources to running queries.

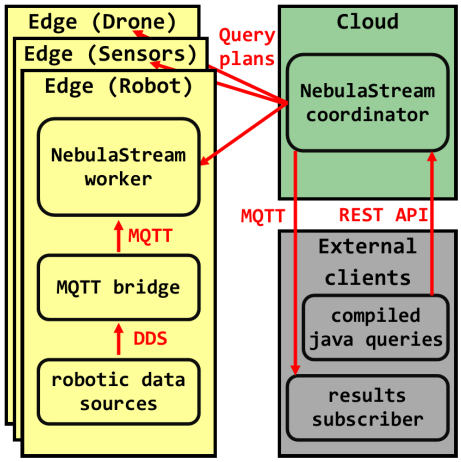

Figure 4: Data flow in the NebulaStream-based architecture.

3.4. Comparison of implementation approaches

Table 1 summarises and contrasts the three implementation approaches discussed above with respect to their programming model, execution scope, spatial support, and operational characteristics within the GIS4IoRT architecture. By presenting their key strengths and limitations side by side, the table highlights the trade-offs that arise from different technological choices when realising the same spatio-temporal queries. These differences motivate the experimental evaluation presented in the next section, which assesses how the approaches perform in practice on real agricultural robot data.

Table 1
Comparison of implementation frameworks for GIS4IoRT query processing.

Aspect	ksqlDB	Flink + GeoFlink	NebulaStream
Paradigm	SQL stream processing	Distributed data streams	Edge–cloud streaming
Programming model	Declarative SQL + UDF	DataStream API	Query plans + Java UDF
Execution scope	Cloud / broker	Cluster (fog/cloud)	Edge–fog–cloud
Spatial support	Custom UDF (PiP)	Native grid index	Custom Java UDF
State handling	Engine-managed	Explicit managed state	Coordinator-managed
Dynamic config	High	Medium	Low
Scalability	Kafka partitions	Spatial partitioning	Distributed workers
Development effort	Low	Med.–High	High
Key strengths	Simplicity, SQL	Performance, control	IoRT alignment
Key limits	Weak spatial support	High expertise	Complex setup
GIS4IoRT role	Lightweight DI layer	Processing core	Full middleware
Query 1a	Native	Native	Native
Query 1b	External batch	Replay or batch	Stream replay
Query 2	External Python, global state	Native spatial join	Window-based join
Query 3	Broadcast join, stream-based	Native stateful stream join	Window-based join

4. Experiments and results

This section presents an experimental evaluation of the three alternative GIS4IoRT processing pipelines presented in Section 3. Using real robot trajectories and GIS datasets, we analyse how different implementation choices affect query execution, latency, and system behaviour.

4.1. Experimental setup

Despite the distributed nature of the proposed architecture, the comparative tests were conducted on a single physical machine to ensure a controlled environment and eliminate network variability. The worker processes and processing engines were simulated within a Docker Compose environment. The hardware platform consisted of an Intel Core i5-14600K CPU, 64 GB of DDR5 RAM, and a Docker Engine environment running on WSL2.

The evaluation was performed using recorded robot trajectories replayed from ROS bag files, rather than live telemetry streams. This approach allowed for deterministic reproducibility of the experiments.

We focused on two key metrics, outlined below.

- **Latency**: defined as the difference between the data ingestion timestamp and the time the alert was received by the data collector. To ensure statistical reliability, extreme outliers were excluded from the final calculations.
- **Correctness**: verified by cross-referencing the system's output against a ground truth dataset derived directly from the raw telemetry logs. For queries involving time windows, the raw logs were pre-aggregated to match the windowing logic, enabling precise verification of the results.

4.2. Baseline performance: geofencing

The **geofencing** query (*Query 1a*) was selected as the primary baseline for quantitative performance comparison. The test measured the system's ability to detect boundary violations on a real-world trajectory, as illustrated in Fig. 5. In this scenario, the standard 1-second tumbling window requirement

was intentionally relaxed to *event-at-a-time* processing. This modification allowed us to isolate the inherent processing overhead of each engine without the masking effect of window aggregation delays.

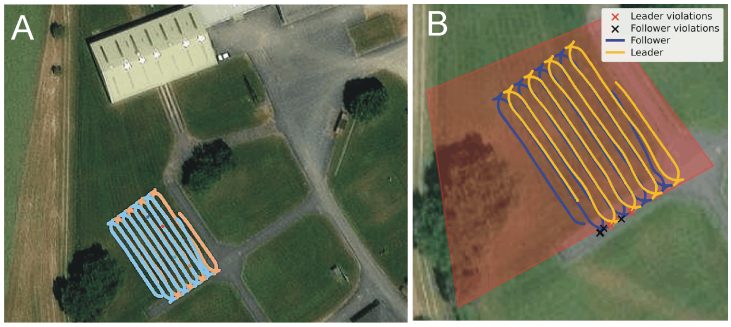

Figure 5: Geofencing example: (A) Trajectories of the leader (orange) and follower (blue) robots overlaid on satellite imagery of the study area. (B) Detail of the geofencing result, showing the plot boundary and the locations where the leader robot and the follower robot crossed the plot border. Satellite imagery © Esri, Maxar, Earthstar Geographics, and the GIS User Community.

Table 2 presents comparative latency results for the geofencing scenario. All three engines correctly detected 100% of the boundary crossing events with single-telemetry precision. The measurements, however, reveal substantial differences in processing overhead and execution behaviour.

Table 2
Detailed latency metrics for geofencing scenario [ms]. Best results are given in bold.

Engine	Avg	Median (P50)	P99	StDev
NebulaStream	**2.1**	**2.0**	**7.0**	**1.4**
Apache Flink	13.0	13.0	21.9	4.3
ksqlDB	250.4	303.0	310.0	64.1

NebulaStream achieved the lowest latency across all metrics, reflecting the benefits of its compiled C++ execution model and streamlined query pipelines. Apache Flink exhibited higher but stable latency values, which are consistent with JVM-based execution and explicit state management. In contrast, ksqlDB incurred significantly higher latency and variance, primarily due to the additional abstraction layers introduced by its declarative processing model.

These results illustrate the trade-off between execution efficiency and abstraction level: lower-level stream processing frameworks offer minimal latency at the cost of increased development effort, whereas higher-level declarative systems favour simplicity and rapid development but introduce measurable performance overhead.

4.3. Functional evaluation: collision and proximity

For the **collision detection** (*Query 2*) and **IoT sensor proximity** (*Query 3*) scenarios, a direct latency comparison was not suitable due to significant differences in implementation.

Specifically, NebulaStream required window-based aggregations to handle complex joins, whereas ksqlDB SQL limitations necessitated offloading logic to external Python consumers. Comparing the latency of a windowed operation (which naturally waits for window closure) against continuous stream processing would be misleading. Therefore, these scenarios were evaluated based on functional correctness, as discussed below.

Collision detection: the system analyzed robot trajectories within the field and correctly identified all spatial violation events, fully matching the ground truth derived from raw GPS logs.

IoT sensor proximity: the engines successfully identified the intervals during which robots entered the sensor range (Fig. 6). However, minor discrepancies were observed at the level of individual telemetry packets. These artifacts resulted from the asynchronous nature of the data streams, where a sensor

Figure 6: IoT sensor proximity query result, showing robot positions (red) within the sensor range during intervals where the sensor reading exceeded the defined threshold.

state update was occasionally processed before a slightly delayed telemetry packet. Such temporal synchronization offsets are characteristic of distributed real-time systems and did not impact the overall reliability of the detected events.

5. Conclusions

In this paper we presented preliminary results comparing three alternative technologies for building an integration architecture for the IoRT. We compared the technologies with respect to their functionality, data latency, and correctness of query results. To this end we used real data from robotic devices and three types of the most common queries in robotic GIS. To the best of our knowledge, this is the only work that compares three alternative system designs based on Kafka/ksqlDB, Flink/GeoFlink, and NebulaStream, for the same business scenario. The work has been realized within the EU CHIST-ERA project.

Since the project is still in its early stage, future works will focus on: (1) evaluating scalability of each architectural design, (2) developing techniques for dynamically including robotic devices into the integration architecture, (3) developing mechanisms for querying intermittent robotic data sources, (4) extending the query mechanism to include parameters like the quality of service and the quality of data.

Acknowledgments

This research is supported from the National Science Centre, Poland, grant no. 2024/06/Y/ST6/00136, funded from the *CHIST-ERA call 2023* EU project *Development of a Plug-and-Play Middleware for Integrating Robot Sensor Data with GIS Tools in a Cloud Environment*. The authors would like to thank Dr Sandro Bimonte (INRAE) for sharing the field experiment data.

Declaration on Generative AI. The authors have not employed any Generative AI tools.

References

[1] R. Raja, Software architecture for agricultural robots: Systems, requirements, challenges, case studies, and future perspectives, IEEE Transactions on AgriFood Electronics 2 (2024).

[2] E. Dritsas, M. Trigka, Remote sensing and geospatial analysis in the Big Data era: A survey, Remote Sensing 17 (2025).

[3] A. Gillet, É. Leclercq, N. Cullot, Lambda+, the renewal of the lambda architecture: Category theory to the rescue, in: Int. Conf. on Advanced Information Systems Engineering (CAiSE), volume 12751 of *LNCS*, Springer, 2021.

[4] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, Robot Operating System 2: Design, architecture, and uses in the wild, Science Robotics 7 (2022).

[5] S. Bimonte, G. Bellocchi, F. Pinet, G. Chalhoub, M. A. Sakr, P. Skrzypczynski, Data engineering for sustainable agriculture: developments, challenges, and case studies of a novel IoRT architecture, Journal of Big Data 12 (2025) 195.
[6] N. V. B. Yogeswaranathan Kalyani, R. Collier, Digital twin deployment for smart agriculture in cloud-fog-edge infrastructure, Int. Journal of Parallel, Emergent and Distributed Systems 38 (2023).
[7] M. Villari, A. Celesti, M. Fazio, A. Puliafito, AllJoyn Lambda: An architecture for the management of smart environments in IoT, in: International Conference on Smart Computing Workshops (SMARTCOMP Workshops), 2014.
[8] G. S. Thakur, B. L. Bhaduri, J. O. Piburn, K. M. Sims, R. N. Stewart, M. L. Urban, PlanetSense: a real-time streaming and spatio-temporal analytics platform for gathering geo-spatial intelligence from open source data, in: SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems, 2015.
[9] S. Kamburugamuve, L. Christiansen, G. Fox, A framework for real time processing of sensor data in the cloud, Journal of Sensors 2015 (2015).
[10] CHIST-ERA, Development of a plug-and-play middleware for integrating robot sensor data with GIS tools in a cloud environment (GIS4IoRT). Chist-Era Project Call 2023, 2025. URL: https://www.geoscity.uliege.be/cms/c_13470217/en/gis4iort.
[11] S. A. Errami, H. Hajji, K. A. E. Kadi, H. Badir, Spatial big data architecture: From data warehouses and data lakes to the lakehouse, Journal of Parallel and Distributed Computing 176 (2023).
[12] J. Kasprzyk, R. Billen, S. Bimonte, L. d'Orazio, D. Sacharidis, P. Skrzypczynski, R. Wrembel, On integrating robotic data with GIS tools in a cloud environment, in: Workshops of the EDBT/ICDT Joint Conference, volume 3946 of *CEUR Workshop Proceedings*, 2025.
[13] A. Prountzos, E. G. M. Petrakis, Defog: dynamic micro-service placement in hybrid cloud-fog-edge infrastructures, Int. Journal of Web and Grid Services 20 (2024).
[14] P. Brezany, A. M. Tjoa, H. Wanek, A. Wöhrer, Mediators in the architecture of grid information systems, in: Int. Conf. on Parallel Processing and Applied Mathematics (PPAM), volume 3019 of *LNCS*, Springer, 2003.
[15] G. Wiederhold, Mediators in the architecture of future information systems, Computer 25 (1992).
[16] M. J. Sax, Apache Kafka, in: Encyclopedia of Big Data Technologies, Springer, Cham, 2018.
[17] A. Katsifodimos, S. Schelter, Apache Flink: Stream analytics at scale, in: IEEE International Conference on Cloud Engineering Workshop (IC2EW), 2016, pp. 193–193.
[18] S. A. Shaikh, K. Mariam, H. Kitagawa, K.-S. Kim, Geoflink: A distributed and scalable framework for the real-time processing of spatial streams, in: ACM Int. Conf. on Information and Knowledge Management (CIKM), 2020.
[19] M. M. G. Duarte, D. P. A. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Tas, E. Zimányi, M. Sakr, S. Zeuch, V. Markl, Mobility stream processing on NebulaStream and MEOS, in: Companion of the Int. Conf. on Management of Data (SIGMOD/PODS), 2025.
[20] Nebula Stream documentation, 2025. URL: https://web.archive.org/web/20250115180134/https://docs.nebula.stream/.