ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD109

FROM TERABYTES TO PETABYTE: SCALING OF THE ARCHIVING SYSTEM FOR FAIR

V. Rapp[†,1], K. Klimczyk[2], J. Kerssemakers[1], P. Salabura[3]
[1]GSI, Darmstadt, Germany
[2]S2Innovation, Krakow, Poland
[3]UJ, Krakow, Poland

Abstract

With the recent rise of AI and various machine learning models, the importance of storing and managing data generated by control systems is greater than ever before. In 2016, GSI began developing an archiving system to collect, store, and retrieve data from the diverse accelerator devices managed by the GSI control infrastructure. The system was successfully deployed in production in 2021. To evaluate its capabilities and suitability for operational needs, the system was initially launched with a limited storage capacity of 50 TB and reduced computing power. With a current data volume of over 100 GB per day, the archiving system quickly exceeded its initial limits. However, the experience gained in day-to-day operations thus far has allowed us to better understand our use-cases and identify areas for further improvement. In preparation for the anticipated start of FAIR operations in 2027, the system will require significant scaling to meet future demands. Therefore, this is an opportune moment to review and refine the system's architecture based on the experience gained so far. This paper outlines the challenges encountered with the current implementation and presents the solutions that will be incorporated into the system for FAIR operations.

INTRODUCTION

As part of the controls renovation for the FAIR project, GSI initiated the development of an archiving system in 2016 [1]. After several years of development, the first production-ready version was released in 2021. The primary objective of the system is the collection, storage, and presentation of data generated by control devices and other relevant infrastructure. This system was the first of its kind and scale at GSI and posed several challenges in terms of requirement analysis and assumptions regarding scalability. Consequently, the architecture was designed with a high degree of flexibility with respect to incoming data structures and the ability to accommodate high data rates. Elasticsearch was selected as the storage backbone, leveraging the department's prior experience with several self-hosted Elasticsearch installations at GSI, which provided the necessary operational expertise.

In order to build operational knowledge, evaluate system behavior under realistic conditions, and prepare for FAIR-scale operation, the system was initially deployed with limited resources. The storage backbone comprised three data nodes with a total capacity of 30 TB, while the application components were deployed on two dedicated nodes. As storage rapidly emerged as a bottleneck, capacity was extended to 50 TB. After resolution of initial operational issues, the system demonstrated stable performance and is currently employed in several important use cases [2]. Presently, the system ingests over 100 GB of data per day. To mitigate storage constraints, data is decimated after each beam time, which presently spans approximately six months. In 2024, following modifications to the storage format and migration from Elasticsearch to OpenSearch, the system was integrated with Grafana, thereby providing users with an efficient graphical interface for data access and visualization (Fig. 1).

Figure 1: One of the Grafana dashboards.

While the system has proven adequate for numerous use cases, operational experience has also revealed several architectural limitations. The design philosophy of maximum flexibility has resulted in comparatively high maintenance demands. This flexibility requires careful configuration and user discipline to ensure efficient operation. Furthermore, although Elasticsearch exhibits good performance with smaller datasets, query latency increases substantially with growing index sizes. In addition, the analytical capabilities of the system remain limited and do not fully meet contemporary expectations.

With the commencement of FAIR operation anticipated in 2027 and an extended beam intervention scheduled between 2025 and 2026, GSI has initiated a comprehensive refactoring of the system to meet future requirements. The primary objectives of this effort are to develop a simplified, maintainable architecture and to enable scalability to the petabyte regime.

SYSTEM ARCHITECTURE OVERVIEW

Similar to the current implementation, the refactored system will retain a layered architecture with a relatively straightforward data flow (Fig. 2). Data is collected by

† V.Rapp@gsi.de

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD109

multiple collectors, the number of which can be adjusted at runtime without requiring a system restart. Each collector subscribes to a defined set of sources via JAPC [3] subscriptions. The JAPC framework provides a unified interface that supports multiple communication protocols and device implementations.

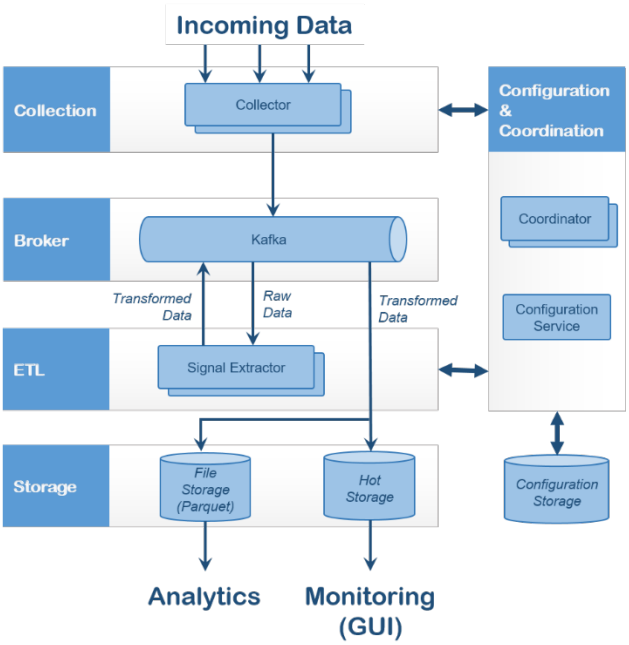

Figure 2: Architecture Overview.

Collectors are managed by a coordination service, which maintains an overview of all connected clients, especially collectors, and distributes subscriptions among them. If a collector becomes unavailable, the coordinator redistributes the remaining load across the active collectors. The coordination service itself is redundant: multiple instances run concurrently, electing a single leader, with the remaining nodes operating as hot standbys. Configuration data—including information about devices, properties, and subscription details—is stored in an Oracle database. Users configure subscriptions through a graphical interface. The current interface is a standalone JavaFX application, while a new web-based version is under development.

Once acquired, the data is published—without processing—to a Kafka broker. The broker temporarily stores messages until they are consumed, acting as an intermediate buffer. Data is processed by extraction and transformation (ETL) workers, which extract signal values and other relevant information from the incoming raw messages. The extracted values are then republished to Kafka in a separate topic, using a standardized, signal-data format, serialized with Avro [4]. Configuration details for transformation and extraction are stored in the Oracle database alongside other system settings.

Transformed messages are consumed by the storage layer and written to two destinations. The first is long-term, file-based storage: data is initially written into small files, which are subsequently compacted into larger files to improve performance and eliminate duplicates. Apache *Parquet* format is used for the files. In parallel, the data is also written to short-term (hot) storage, where recent data remains directly accessible to users. The hot storage is connected to Grafana, allowing users to monitor critical parameters in near real time. Since the data format follows a stable and simple time-series standard, the hot storage layer can be implemented on a variety of backends. At present, the plan is to utilize the existing *OpenSearch* cluster for 2026, with the intention of replacing it in the future with a more performant, dedicated solution.

IMPLEMENTATION DETAILS

Collecting and Transforming Data

The collection layer is responsible for acquiring values from the controls infrastructure and forwarding them to the Kafka broker in a stable and uniform format. In this layer only light normalization and optional filtering is performed. Signal extraction is deliberately excluded from this stage and is delegated to downstream ETL workers.

To accommodate heterogeneous device payloads, the collectors employ a schema-less serialization strategy. This approach preserves native data types while ensuring a consistent message format. Several binary, schema-less encodings—such as *MessagePack*, *BSON*, and *JSON* variants—were evaluated. Ultimately, *Smile* was selected as the preferred format due to its simplicity, solid performance characteristics, and compact payload size, while avoiding the operational overhead of a centralized schema registry.

Normalized and split records are placed into bounded internal queues, from which worker threads manage publishing to Kafka.

In earlier implementations, large records were handled through byte-level chunking on the collector side, with reassembly required by the writer. This approach introduced unnecessary complexity into an otherwise straightforward data pipeline. In the refactored design, chunking is replaced with semantic splitting: large records are divided into smaller, self-contained units that remain valid individually but together represent the original logical content. This ensures that messages remain within Kafka size limits while eliminating the need for reassembly logic on the consumer side.

The ETL pipeline converts normalized device records into structured, signal-centric outputs. Workers execute a Kafka Streams topology that performs lightweight validation, configurable extraction, and key/value shaping before publishing. Two complementary extraction modes are supported:

- **Scalar extraction**, which selects a single value from structured fields to emit one signals value record
- **Array extraction**, which expands array-like fields into multiple signal value records, suitable for time series or dense sensor arrays.

Extraction rules are configuration-driven, allowing new signals to be onboarded without code changes. Each signal is emitted as an Avro-encoded record, using an evolution-friendly schema to standardize interchange and support long-term archival.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD109

Kafka is used as a reliable persistence checkpoint: Collection is kept deliberately simple to minimize the chances of permanent data-loss before reaching this initial checkpoint. Similarly, the ETL processing and permanent storage writers are programmed to only fully commit their Kafka reads after the next stable checkpoint has been reached. This ensures any processing step can be retried, and that unexpected failures do not lead to loss of in-flight data. Additionally, Kafka's buffering capabilities ensure that any downtime (planned or unplanned) or capacity problems do not stall the entire pipeline. The same buffering also allows us to operate without cumbersome round-the-clock on-call shifts.

The system remains stateless with respect to extraction, enabling horizontal scaling; partitioning by key preserves per-key ordering. Input validation drops empty or malformed records, while Avro schemas enforce boundary checks. Outputs are written to a dedicated signals topic, allowing downstream services to scale independently.

Based on observed average throughput and the current production environment, at least two ETL instances are required to sustain ingestion with sufficient headroom while maintaining high availability.

To choose the serializer we compared Avro and Protobuf [5] in the end-to-end pipeline comparing sequential and parallel execution across two data types (scalar and array). The results are presented in the Table 1.

Following set up was used:

Input: 10M generated values, as scalar or packaged into arrays (1000 per array)

Output signal values: 10M per run

Parallel settings: thread pool with 15 threads

Table 1: Pipeline Throughput

Data	Serializer	Throughput Sequential [rec/s]	Throughput Parallel [rec/s]
Scalar	Avro	952 392	1 852 732
Scalar	Protobuf	921 784	1 997 806
Array	Avro	5 597 107	17 216 995
Array	Protobuf	3 197 834	11 032 803

The significant difference between the scalar and array times can be explained by the need to process each individual record, while the array allows a batch processing. For comparability reasons the tests were performed with generated data. On array device data (many signals per record), Avro sustains markedly higher throughput than Protobuf in both modes.

Lightweight Coordination Service

To achieve scalability and reliability, the system relies on multiple collector instances that forward data to Kafka. In earlier versions, coordination of these collectors was based on Apache *ZooKeeper*, which provided synchronization and load balancing. However, ZooKeeper introduced additional complexity and maintenance overhead, making it less suitable for domain-specific deployments. Moreover, the latest versions of Kafka no longer depend on ZooKeeper, rendering it an unnecessary external dependency.

As part of this work, a range of alternatives and related technologies were examined [6]. These included consensus-based systems such as *etcd* and *Consul*, common web load balancers such as *HAProxy* and *Traefik*, as well as distributed consensus algorithms including *Raft* and *Paxos*. Several Java-based implementations of Raft were also evaluated. While these solutions are powerful and widely adopted, they often introduce unnecessary complexity or are optimized for larger clusters than required in this context.

Among the evaluated options, consensus algorithms—particularly Raft—appeared most promising. A proof-of-concept was developed using Apache *Ratis*, its Java implementation, which confirmed the feasibility of building a custom coordination layer. Nevertheless, the requirements of the archiving system favoured an even simpler and lighter solution. Consensus protocols excel in large clusters that demand strong fault tolerance through log replication and majority voting. In contrast, for small clusters, the additional latency, operational overhead, and system complexity outweigh the benefits.

In practice at the GSI and with limited available resources, the archiving system is often deployed on two nodes or other uneven counts. In such cases, quorum-based approaches offer limited added value and can even delay failure recovery. Instead, what is needed most is fast, predictable failover combined with behaviour that is straightforward to understand and operate.

The new coordination service for the Archiving System has therefore been designed with an emphasis on simplicity, reliability, and suitability for small clusters [7]. Its architecture deliberately avoids the overhead of a full Raft implementation, instead employing a lightweight leader-election approach based on the Bully algorithm. At the core, the Coordinator Service provides central orchestration, managing both lifecycle and configuration. Leadership is ensured through the Bully algorithm, which combines timeout-based failure detection with automatic re-election, while workload distribution is handled using a least-connections algorithm supported by state management.

The coordination mechanism is built on an event-driven message model. At its core, a message dispatcher functions as the communication hub, reliably handling both request–response interactions and event broadcasts. It employs a thread-safe queue and asynchronous execution via a bounded thread pool, with a defined fallback policy to handle worker saturation. Incoming requests are routed to type-specific handlers, while events are concurrently delivered to all subscribed components.

The following in-code classes integrate with the dispatcher to implement system behaviour:

- *Election Manager* – orchestrates leader election and announces leadership changes

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD109

- *Heartbeat Manager* – manages follower heartbeats to the current leader and processes heartbeat pings
- *Configuration Manager* – bridges configuration changes into internal events and serves the resource inventory
- *Load-Balancer* – maintains distribution state and (re)assigns resources
- *Instance Status Monitor* – tracks and publishes instance online/offline status

Node supervision is managed primarily by the *Heartbeat Manager*, which monitors system health through periodic heartbeat messages sent from followers to the leader. Configurable intervals and timeouts enable rapid failure detection, while status changes trigger real-time notifications. The *Election Manager* complements this by maintaining knowledge of all nodes, executing the Bully algorithm for leader election, and broadcasting results via the event system. In parallel, the *Load Balancer* oversees resource assignments and automatically redistributes them in the event of node failures.

On the client side the interface exposes a simplified API that provides automatic coordinator discovery among the configured nodes and transparent connection handling. As a result, clients can be integrated with minimal configuration, while maintaining awareness of the current topology and reducing overall complexity.

To assess robustness, the evaluation focuses on sustained and burst throughput as well as fault recovery under realistic operating conditions. The primary emphasis is on timely failure detection, correct leader re-election when required, and non-disruptive workload rebalancing.

Fault scenarios and expected behavior:

- Follower coordinator loss: Detected through missed heartbeats. Leadership remains unchanged and assignments stay stable. Detection occurs within the configured timeout, with no unintended reassignments and no slowdown in ingestion
- Leader loss and new-leader propagation: Missed heartbeats trigger a Bully election. An available follower assumes leadership and publishes the update. Collectors automatically discover the new leader, and assignments are recomputed idempotently
- Client churn (collectors connecting or disconnecting): On join/leave events, the load balancer reassigns resources. Reassignment completes without destabilizing throughput, and no duplicate processing occurs

The system consistently delivers stable throughput, with only rare and expected losses during scheduled rebalancing periods.

Storing Data

The new archiving system envisions a data warehouse based on a filesystem hierarchy of parquet files. This design seeks to reduce barriers for downstream re-use. *Parquet* has established itself as an industry standard, and is supported by all notable data analysis frameworks.

Compared to dedicated (timeseries) databases, filesystem hierarchies score highly on accessibility and ease of maintenance. Researchers wishing to analyse archived data also need not learn database-specific access strategies, and can use widely available data analysis libraries in their programming language of choice.

A writer instance converts extracted signals into persistent, analysis-ready artefacts. It ingests Avro-formatted records from broker topics via a Kafka Streams pipeline [8], performs validation and buffering, and commits them as columnar Parquet files optimized for query workloads. Each signal is managed by a dedicated writer instance to maximize parallelism and isolate potential faults.

In comparative testing, Avro outperformed Protobuf for Parquet workflows, delivering approximately 28% faster write performance and 35% faster read performance. The measured throughput of the pipeline is about 650,000 records per second when writing to Parquet using Avro encoding.

Storage is organized by signal identity. Each signal maps to a dedicated directory through a deterministic scheme (identity → directory; time window → file). Within each directory.

Reliability follows the streaming model: broker-backed consumption provides backpressure through offset progression. Errors during writes or flushes are logged and counted, and failed appends affect only the corresponding signal. The system scales horizontally through partitioning and per-signal isolation. In the event of an instance failure, another instance can resume processing with at-least-once delivery guarantees.

For consumers, Parquet provides compact storage, effective column pruning, and vectorized reads across analytics engines. The per-signal directory layout combined with hourly file rotation enables efficient retrieval of specific signals over defined time ranges.

CONCLUSION AND FUTURE WORK

The refactored Archiving System for FAIR builds on a layered architecture that separates collection, transformation, coordination, and storage to balance scalability with maintainability. At the front of the pipeline, collectors act as lightweight proxies between the control infrastructure and Kafka, using Smile serialization to handle heterogeneous payloads without schema registry overhead. A streamlined coordination service—combining heartbeat supervision, Bully leader election, and least-connections load balancing—ensures predictable failover and resilience without the complexity of full consensus protocols. Downstream, ETL workers transform raw device records into structured, signal-centric outputs. Writer instances persist these signals as Parquet files, organized per signal and time window, delivering compact storage, efficient retrieval, and analysis-ready data. Together, these components form a robust, high-throughput pipeline capable of sustained ingest, reliable fault recovery, and scalable data access.

Compared to the current design, each component has been simplified to remove excess flexibility, and chiefly:

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUPD109

data is transformed for efficient downstream use, instead of stored in the raw incoming format.

Looking ahead, the system provides a solid foundation for future extensions and improvements that will further align it with FAIR's operational needs.

Currently the system is still under the development with an anticipated production start in the beamtime 2026. The main focus in the near future would be the finalisation of particular aspects of the system like the compaction of the data for a longer storage as well as developing of a flexible and user-friendly GUI to edit the settings. The next step would be a provision of powerful API for users to work and analyse the data.

These developments will ensure that the Archiving System not only meets immediate operational requirements but also evolves into a sustainable platform for FAIR's long-term data management needs.

REFERENCES

[1] V. Rapp and V. Cucek, "Concept and First Evaluation of the Archiving System for FAIR", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 486-489. doi:10.18429/JACoW-ICALEPCS2017-TUPHA043

[2] O. Geithner, R. Assmann, W. Geithner, F. Herfurth, V. Rapp, and S. Reimann, "Introduction of key performance indicators for the GSI accelerator facility", presented at the IPAC'25, Taipei, Taiwan, Jun. 2025, paper MOPS008, unpublished.

[3] V. Rapp and W. Sliwinski, "Controls Middleware for FAIR", in *Proc. PCaPAC'14*, Karlsruhe, Germany, Oct. 2014, paper WCO102, pp. 4-6.

[4] Apache Avro, https://avro.apache.org

[5] Protocol Buffers, https://protobuf.dev

[6] K. Klimczyk, Replacement of Zookeeper synchronisation service in the Archiving System, GSI, Darmstadt, Germany, GI-231726K-PL-SIE, 2023.

[7] K. Klimczyk, Implementation of a Multi-Node Coordination Service for the Archiving System, GSI, Darmstadt Germany, GI-242630K-PL-PIP, 2024.

[8] Kafka Streams, https://kafka.apache.org/documentation/streams