A Lightweight Real-Time Data Post-Processing Framework for Legacy High-Speed Train Ground Maintenance Systems: Design and Performance Evaluation

Tianyu Xia[1, 2], Beichen Gong[1], Jingxian Ding[1*], and Jianyong Zuo[1, 2*]

[1] College of Transportation, Tongji University, Shanghai, China.
[2] Shanghai Key Laboratory of Rail Infrastructure Durability and System Safety, Tongji University, Shanghai, China.

* Corresponding author. Email: dingjx@tongji.edu.cn (J.D.); zuojy@tongji.edu.cn (J.Z.)
Manuscript submitted January 28, 2026; accepted April 2, 2026; published May 28, 2026.
doi: 10.17706/jsw.21.1.32-44

Abstract: High-speed train Electric Multiple Units (EMU) ground maintenance platforms generate critical telemetry data from sub systems such as pneumatics, braking, and traction. While real-time monitoring is essential for safety, many existing ground stations rely on monolithic Relational Database Management Systems (RDBMS) and lack the infrastructure for heavy big data stacks. This paper proposes a lightweight, non-intrusive real-time post-processing architecture based on Python and standard RDBMS mechanisms which requires no modification to the original data ingestion pipeline. The system is evaluated using telemetry schemas from EMU braking systems. Results show a saturation throughput of 1600 records per second with end-to-end latency of 20 ms on total data volumes of 10^6 records, which is sufficient to support the monitoring of over 200 concurrent subsystem units at standard sampling rates. A comparative analysis with the industry-standard tool Debezium reveals similar performance limits under identical hardware constraints. This study validates that, for specific legacy industrial scenarios, an RDBMS-native approach offers a superior trade-off between performance, cost, and maintainability.

Key words: digital twin, real-time processing, relational database, railway equipment maintenance, EMU

1. Introduction

Industrial digital twins generate high frequency time series data with low value density [1, 2]. To derive operational value, digital twin technology can transform these streams via consistent cleaning, analytics, and visualization [3–7]. In legacy industrial storage stacks, Relational Database Management Systems (RDBMS) are the de facto data stores owing to their mature ecosystems. For existing legacy systems like high-speed train ground maintenance platforms, a pragmatic strategy is to augment existing RDBMS-based architectures with low-cost, minimally invasive capabilities for post-processing and visualization, without disruptive refactoring. This enables traditional platforms to integrate into an efficient pipeline for digital twins, thereby supporting real-time monitoring, alerting, report generation, and historical analytics [8–11].

To address the storage and processing requirements for industrial data with low latency in railway equipment maintenance systems, a variety of technical solutions have been developed. These can be broadly categorized into Relational Database Management Systems (RDBMS), Time Series Databases (TSDB), stream

processing with message queues [12, 13], distributed stream processing [14], and traditional offline processing. Traditional offline methods typically store data in semi-structured text files rather than databases. For instance, Fu [15] utilized this approach and achieved high performance in scheduled or deferred batch processing. However, the significant latency incurred by such methods makes them unsuitable for monitoring and alerting scenarios that demand near instantaneous responses. Distributed stream processing represents a mainstream technology for handling modern industrial big data, well suited for workloads characterized by high throughput and computational complexity. Yu *et al.* [16], for example, designed a system based on Spark Streaming that effectively processes massive industrial data generated at high speeds. Nevertheless, these systems often necessitate deep integration with original data pipelines, complicating efforts to augment existing systems in a minimally intrusive manner. Their architectures are also complex and resource intensive, and their reliance on specific hardware often leads to prohibitive deployment and operational costs for small and medium sized industrial settings. Similarly, Zhu *et al.* [17] employed Netty and Kafka for high concurrency stream processing in the industrial internet, but the system's distributed architecture is suited for centralized cloud platforms, making it difficult to meet digital twin requirements for localized data processing and lightweight deployment. Zheng and Tian [18] modeled digital twin data using a time series database; however, the adoption of TSDBs typically requires modifying the prevalent relational database architectures in traditional industrial systems. Relational databases offer technological maturity, robust ecosystems, and strong ACID (Atomicity, Consistency, Isolation, Durability) guarantees [19]. They have been successfully applied in the monitoring of digital twin production lines for both historical data storage and online data exchange [20]. Furthermore, Tang *et al.* [21] proposed storing industrial time series data in a relational database using the JSON data format.

To address this gap, we propose a minimally invasive, RDBMS-native post-processing architecture that enables sub-20 ms latency and successfully processes all ingested records without external middleware or database schema refactoring. Our system operates entirely within standard relational databases (validated on MySQL 8.0) and offers two change-data-capture strategies: (i) a timestamp-based polling method, and (ii) a trigger-based approach that writes new records to an intermediate staging table, thereby decoupling ingestion from processing. The latter design ensures consistent performance even under degraded indexing: latency remains less than 11.1 ms across data scales from 10^3 to 10^6 records and throughput of 1600 records per second, with no loss in completeness. Statistical analysis confirms a significant interaction between indexing strategy and dataset size for polling ($p < 0.001$), but not for the trigger-based method ($p = 0.52$). Comparative experiments with Debezium, an industry-standard Change Data Capture (CDC) tool, under identical hardware constraints reveal similar performance limits, validating the throughput bottleneck lies in the RDBMS I/O latency and ACID transaction rather than the lightweight design of our system.

This work demonstrates that EMU ground platforms can achieve real-time post-processing capabilities through lightweight augmentation of existing RDBMS deployments, enabling rapid deployment of digital twin backends in settings where infrastructure overhaul is impractical. Compared to existing CDC solutions, the proposed architecture is distinguished by its avoidance of external middleware. This design enables deployment and debugging with minimal cost and complexity, requiring no downtime or modification of the original data ingestion pipeline. As a result, the system is particularly well-suited for legacy industrial environments, where intrusive changes and operational interruptions are prohibitive. The approach offers a practical path for rapid integration of real-time post-processing capabilities into legacy industrial ecosystems.

The remainder of this paper is organized as follows: Section 2 details the system architecture and design; Section 3 presents a case study with experimental evaluation and a comparative analysis with Debezium; and Section 4 concludes the paper.

2. Method

The proposed real-time post-processing system is designed to fulfill three primary functional objectives: 1) Ingesting and storing industrial data streams, with provisions for both transient and persistent data handling. 2) Executing real-time computations and analyses on incoming data and preserving the derived results. 3) Providing a unified query interface for both historical and real-time data to support downstream applications, such as analytics and visualization. In Fig. 1, the system employs an architecture with three layers structured around the data flow: 1) The Data Source Layer serves as the foundation, managing data ingestion, persistence, and query execution. 2) The Listening Layer continuously monitors the Data Source Layer to retrieve new data records as they become available for processing. 3) The Data Processing Layer consumes data supplied by the Listening Layer, executes specific logic for analysis and transformation, and writes the processed results back to the Data Source Layer. A separate User Interaction Module provides a centralized interface for managing system configurations and monitoring runtime logs, enabling administrators to control system operations. Externally, the system interfaces with data acquisition sources and exposes processed data to external consumers for applications such as report generation, real-time monitoring, and fault alarming.

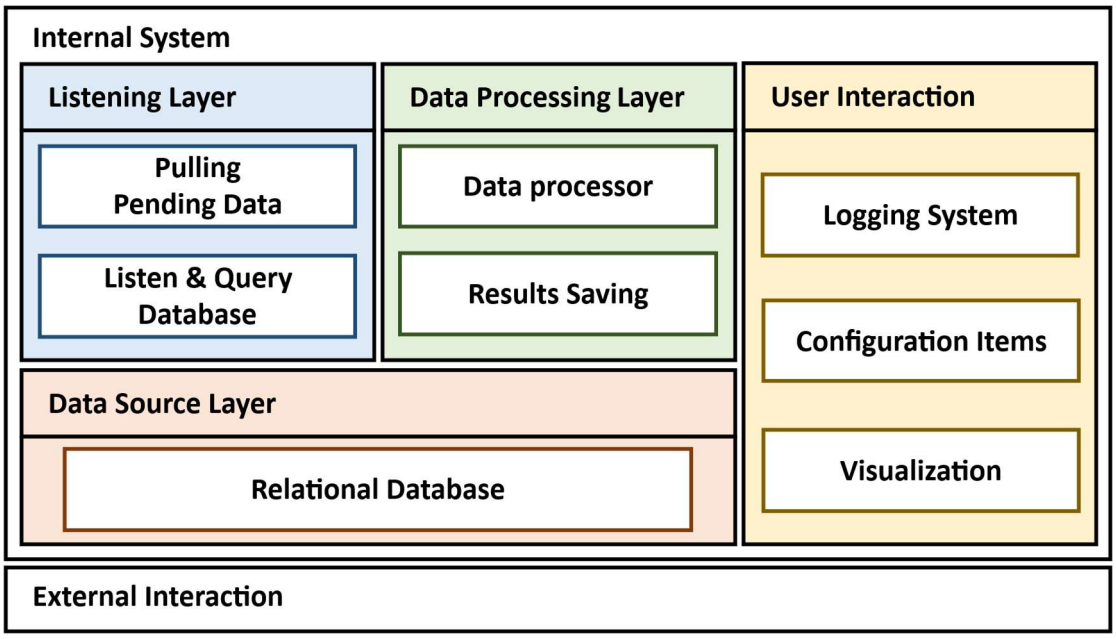


Fig. 1. System architecture.

The Data Source Layer is architected around a relational database that serves as the central repository for data persistence and the primary interface for data exchange. For storage, the database ingests data from upstream sources via standard drivers and employs indexing to ensure efficient organization and retrieval. For interaction, its query interface exposes data to the internal Listening Layer and to external consumers. The use of standardized database interfaces facilitates the decoupling of data flow stages. As depicted in Fig. 2, this layer may be implemented either with a dedicated internal database, as shown in Fig. 2(a), or with an existing database from a legacy workflow, as shown in Fig. 2(b), offering flexible deployment options.

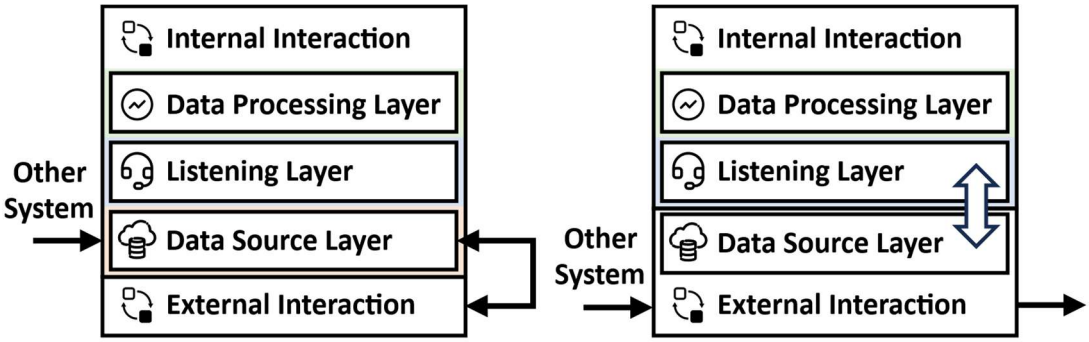
The database schema comprises two principal tables: a source table for raw data ingestion and a result table for storing processed data. To facilitate time series data handling, both tables utilize a millisecond timestamp as the primary key. The data payload is encapsulated within a JSON field to accommodate semi-structured data formats. The structures for these tables are detailed in Tables 1 and 2.

Table 1. Source table structure

Field Name	Description	Constraints
time	Timestamp in ms	Primary Key
data	Data payload in JSON	Not Null

Table 2. Result table structure

Field Name	Description	Constraints
time	Timestamp in ms	Primary Key
result	Processed data in JSON	Not Null
time res	Timestamp of result in ms	Not Null



(a) (b)

Fig. 2. Data source layer deployment modes. (a) Internal database; (b) Existing database.

The Listening Layer is responsible for detecting and retrieving new data from the Data Source Layer. It maintains a persistent connection to the database and implements two distinct listening strategies to capture data changes: a polling-based method and a trigger-based method. Upon detecting new records, the layer fetches them and places them into a task queue for consumption by the Data Processing Layer. These two mechanisms are designed to accommodate different data ingestion patterns and database constraints.

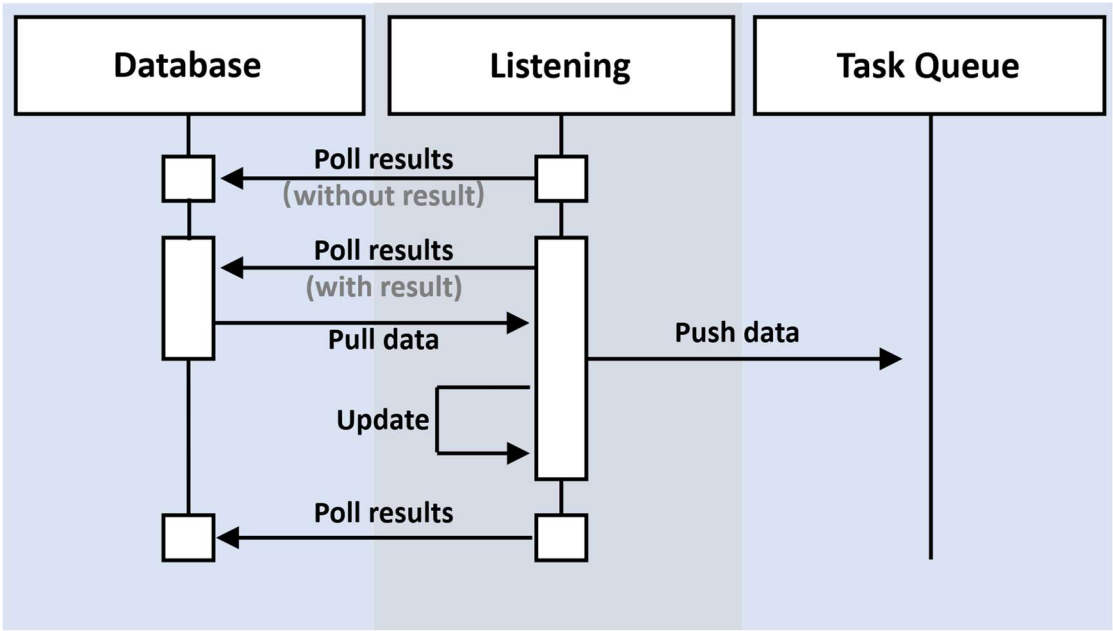


Fig. 3. Polling-based listening mechanism.

The first approach, the method based on polling, directly queries the source table at regular intervals, as illustrated in Fig. 3. This method maintains a polling cursor, which records the timestamp of the last successfully retrieved record. In each cycle, the listener queries for new records with timestamps greater than this cursor. If new data is found, it is enqueued for processing, and the cursor is updated to the timestamp of the newest record.

The second approach is a trigger-based method, designed for scenarios where direct, frequent polling of the source table is undesirable, such as in legacy systems. As depicted in Fig. 4, this method installs a trigger on the source table. This trigger automatically executes upon an INSERT operation, copying the new record to a dedicated intermediate table. The Listening Layer then polls this intermediate table to retrieve new data. After processing, the result is persisted to the result table. A second trigger, placed on the result table, then deletes the corresponding record from the intermediate table. This decoupled architecture avoids frequent queries on the primary source table, minimizing the performance impact on the existing system.

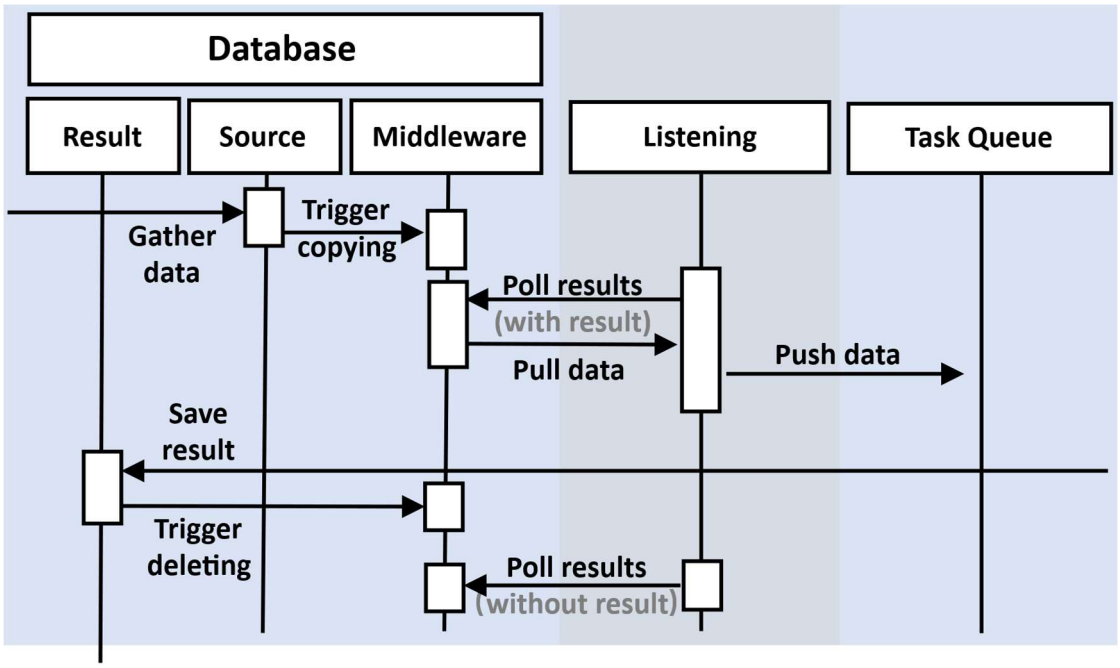


Fig. 4. Trigger-based listening mechanism.

The Data Processing Layer is composed of a task queue, a pool of worker threads, and a database connection pool. This layer implements a producer-consumer pattern [22, 23], where the Listening Layer acts as the producer by placing new data into a task queue. The data processor, acting as the consumer, utilizes multiple worker threads to retrieve tasks from the queue. Each thread processes the data and persists the result to the database using a connection from the connection pool. The operational logic of the data processor is detailed in the following pseudocode.

Algorithm 1: Actions of Worker Threads

```
global dbPool: DBConnectionPool
class Worker {
    inbox: BlockingQueue<Task>
    onIdle: BlockingQueue<Worker>
    run():
        onIdle.put(self)
```

```
        while true:
            task = inbox.take()
            result = compute(task.payload)
            conn = dbPool.acquire()
            executeInsertOrUpdate(conn, result)
            dbPool.release(conn)
            onIdle.put(self)
}
```

The operational flow is as follows: Upon initialization, each worker thread is added to a shared idle workers queue and waits for a task. The main data processor monitors the global task queue. When a new task arrives, the processor retrieves an available worker from the idle workers queue and dispatches the task to that worker's dedicated inbox. The worker executes the data transformation, acquires a connection from the database pool to persist the result, and releases the connection. After completing the task, the worker is returned to the idle workers queue to await the next assignment.

Algorithm 2: Actions of the Data Processor

```
global taskQueue: BlockingQueue<Task>
class DataProcessor {
    idleWorkers: BlockingQueue<Worker>
    workers: List<Worker>
    init(workerCount):
        idleWorkers = new BlockingQueue<Worker>()
        workers = []
        for i in 1..workerCount:
            w = new Worker(onIdle=idleWorkers)
            workers.add(w)
    start():
        for w in workers: w.start()
        spawn thread scheduleLoop()
    scheduleLoop():
        while true:
            task = taskQueue.take()
            worker = idleWorkers.take()
            worker.inbox.put(task)
}
```

The User Interaction Module provides centralized system management through a web interface, as shown in Fig. 5. All system parameters are stored in external configuration files. Administrators use a web browser to remotely view and modify these configurations, as well as control the lifecycle of system services. For runtime monitoring, the system's standard output is captured, buffered on the server, and streamed to the browser in real time. This design decouples the management interface from the core processing engine, ensuring that the system operates continuously if the administrative client is disconnected.

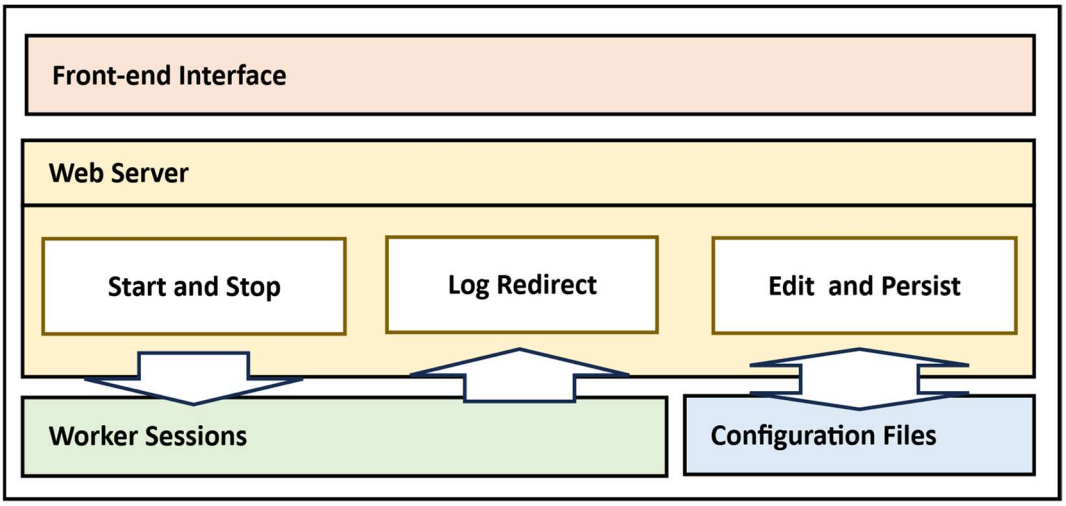


Fig. 5. User interaction module.

3. Case Study

In this case study, the system employs MySQL 8.0 as the Relational Database Management System (RDBMS). Method I operates on a schema including a source table and a result table. Method II requires an intermediate staging table with a schema identical to the source table additionally. The schema is based on simulated telemetry data from an EMU braking system including timestamps, pressure values and valve states. The time field serves as the primary key in both the source and result tables.

The listener layer, processing layer, and user interaction module are implemented in Python 3.9, and the web server is built using Flask. Data processing involves parsing input JSON, performing post-processing operations, and serializing the output back to JSON format.

Fig. 6 presents the user interface, comprising four components: (i) Database Configuration, which specifies all connection parameters and provides a connectivity test; (ii) Listener Configuration, which allows selection of the listening strategy and adjustment of the polling interval; (iii) System Control, which displays the current listener status and enables start/stop operations; and (iv) Logging, which renders runtime logs with options to refresh and clear the history.

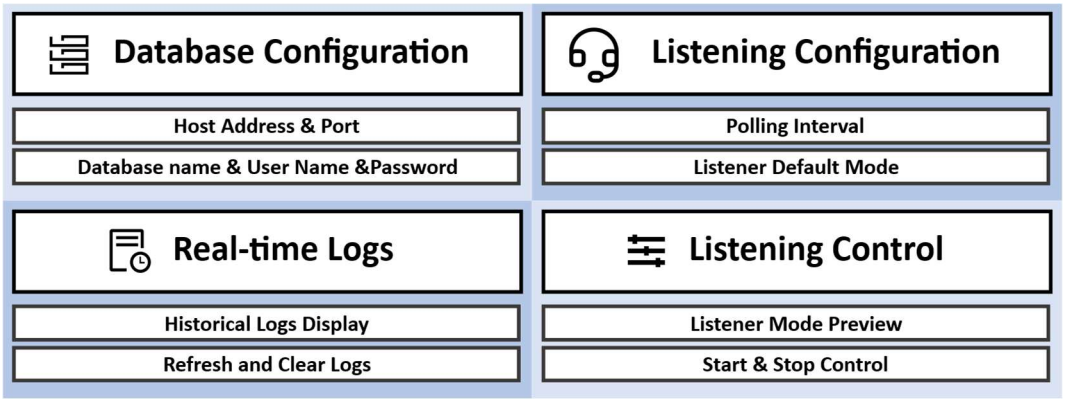


Fig. 6. User interaction components.

The experimental platform is configured with an Intel i5-12450H CPU, 16 GB RAM, running Ubuntu 24.04.

To assess the system's effectiveness and operational stability, controlled experiments under simulated workloads are conducted. Each trial was configured with a 5 ms polling interval and executed for 5 min. The

mean end-to-end processing delay computed over the last 1000 records of each run is reported as the primary latency metric. The initial data volume (preloaded records), input arrival rate, and indexing strategy are systematically varied for analyzing the resulting latency for both listening methods.

To evaluate the effects of different experimental factors on system latency, one-way and two-way Analysis of Variance (ANOVA) are employed.

One-way ANOVA is employed to assess the impact of a single experimental factor on processing latency. In Eq. (1), the *F*-statistic is calculated as the ratio of between-group variance to within-group variance:

$$F = \frac{MS_{between}}{MS_{within}} = \frac{SS_{between}/(k-1)}{SS_{with}\ /(N-k)} \tag{1}$$

where k is the number of groups, N is the total number of observations, and $SS_{between}$ and SS_{within} are the between-group and within-group sums of squares, respectively.

Two-way ANOVA is utilized to examine the main effects of two independent factors and their interaction effect. In Eq. (2), the F-statistics for the main effects (Factor A, Factor B) and the interaction (A × B) are given by:

$$F_A = \frac{MS_A}{MS_E} \tag{2a}$$

$$F_B = \frac{MS_B}{MS_E} \tag{2b}$$

$$F_{AB} = \frac{MS_{AB}}{MS_E} \tag{2c}$$

where MS_A, MS_B, and MS_{AB} are the mean squares for each factor and their interaction, and MS_E is the mean squared error.

The *p*-value quantifies the statistical significance of the results. It represents the probability of observing an *F*-statistic as extreme as, or more extreme than, the one computed from the data, under the assumption that the null hypothesis (H_0) is true. The null hypothesis in ANOVA posits that there are no significant differences between the means of the groups being compared. In Eq. (3), the p-value is derived from the *F*-distribution's Cumulative Distribution Function (CDF), calculated as:

$$p = P\left(F_{df_1,df_2} \geq F\right) = \int_{\mathrm{F}}^{\infty} f(x; df_1, df_2)\mathrm{d}x \tag{3}$$

where F is the calculated F-statistic, and $f(x; df_1, df_2)$ is the probability density function of the *F*-distribution with df_1 and df_2 degrees of freedom.

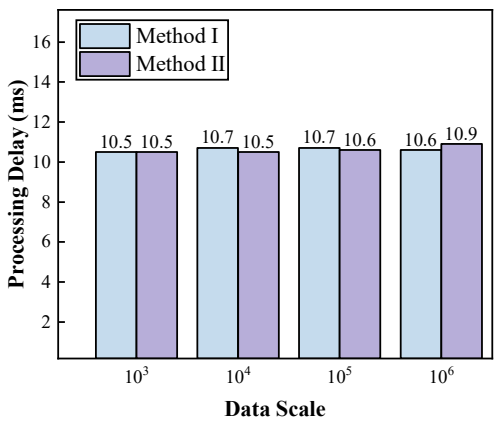
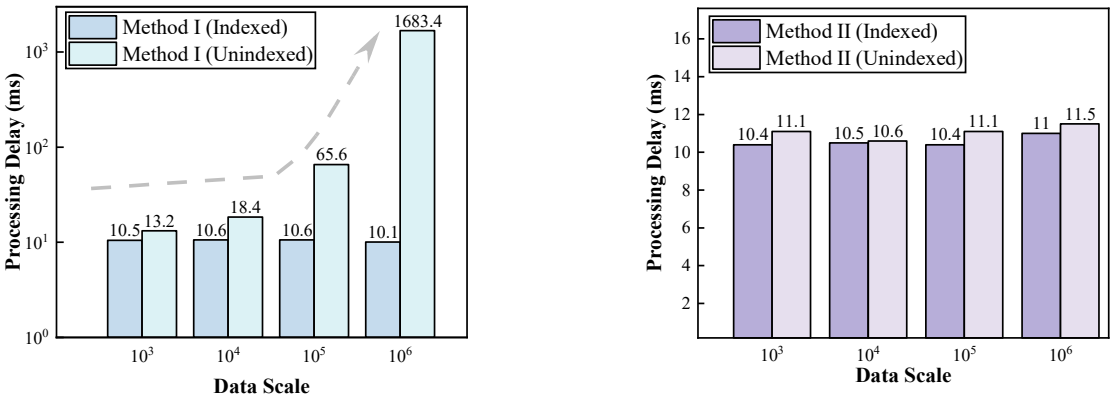


Fig. 7. System latency under different initial data volumes.

The initial data volume experiment evaluates performance when the system starts with a preloaded dataset and continues ingesting, emulating persistent accumulation. At an input rate of 10 records per second, four initial dataset sizes (10^3, 10^4, 10^5, 10^6 records) are tested. Across all conditions, both modes sustained a full processing completion rate, and the mean latency remained approximately 10.6 ms with no discernible trend across scales as shown in Fig. 7.

One-way ANOVA is applied on 100 records from each condition and results indicate no significant effect of existing data volume on latency for either method (Method I: $F(3, 396) = 0.81$, $p = 0.49$; Method II: $F(3, 396) = 0.76$, $p = 0.52$).

The indexing experiment disables primary key indexes on the source and result tables. With a 10 records per second input and varying dataset sizes, latencies for both methods are measured. As shown in Fig. 8(a), in Method I, latency increases rapidly with dataset size; at 10^5 records, real-time performance degrades, and at 10^6 records a backlog forms, the processing completion rate declines from 100% to 31.1%, and the system fails to meet real-time requirements. In contrast, Fig. 8(b) shows that Method II remains stable across dataset sizes.



(a) (b)

Fig. 8. System Latency under Different Indexing Mechanisms: (a) Method I, (b) Method II.

Two-way ANOVA indicates a significant interaction between indexing strategy and dataset size for Method I ($F(3, 72) = 78.08$, $p < 0.001$), while no significant interaction is found for Method II ($F(3, 72) = 0.76$, $p = 0.52$). Significant interaction between listening method and dataset size is also observed while indexing is disabled ($F(3, 72) = 78.07$, $p < 0.001$).

The performance collapse of Method I under degraded indexing is attributed to the $O(N)$ time complexity of full table scans on the massive source table. Conversely, Method II maintains stable $O(1)$ latency because the intermediate staging table consistently holds only unread records (near-zero size), functioning as a lightweight queue that naturally insulates the listening process from the historical data volume.

Overall, appropriate indexing and the proposed listening designs yield robust real-time performance.

In the throughput experiment, we imported a comparison group using Debezium as the Change Data Capture (CDC) tool, Kafka as the message queue, and the same Python consumer for processing.

The system is evaluated under varying input rates (400, 800, 1600, 1800, 2000 records per second) with a fixed dataset size of 10^6 records.

As shown in Fig. 9, both methods maintain sub-20 ms latency up to 1600 records per second.

Beyond this rate, latency increases sharply, and processing completion rates drop below 100% (in our method, records that can't catch up are dropped while in Debezium they accumulate in the Kafka queue).

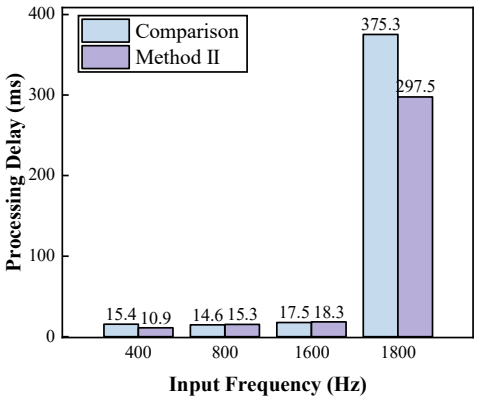


Fig. 9. System latency under different input rates compared with Debezium.

A standard EMU braking control unit typically uploads status packets at a frequency of 1 Hz to 10 Hz. The system can simultaneously monitor the braking status of over 30 typical 8-car trainsets (assuming 8 braking units per motor train, 12 braking units per trailer train). This capacity exceeds the operational requirements of most regional ground maintenance depots.

As shown in Fig. 10, when the input rate increases from 400 to 2000 records per second, the CPU utilization of the Debezium approach rises from 20% to 35%, while our method's CPU usage increases from 15% to 25%. RAM usage remains stable around 2000 MB for Debezium while it rises from 100 MB to 200 MB for our method. The distinct resource profiles observed in Fig. 9 stem from architectural footprints. Debezium, operating on the Java Virtual Machine (JVM) and coupled with Kafka clusters, introduces significant background memory allocation and JVM garbage collection overhead. In contrast, our Python-based worker pool directly manipulates RDBMS natively, avoiding intermediate serialization layers and heavy messaging protocols, thus resulting in a memory footprint an order of magnitude smaller.

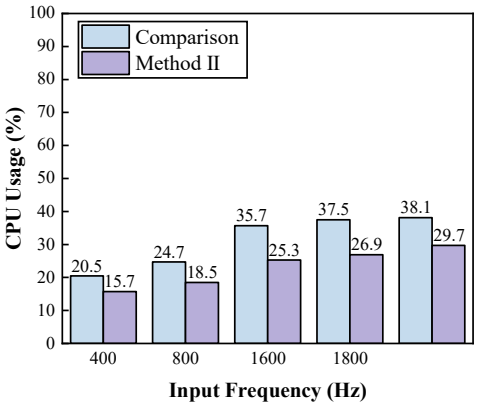


Fig. 10. CPU usage under different input rates compared with Debezium.

CPU usage remained below saturation levels for both methods across all tested input rates, indicating that the bottleneck is attributed to the ACID transaction requirements of MySQL (specifically redo log flushing and row-level locking) during high-concurrency write-back operations.

We compare the system with existing industrial data processing paradigms discussed in Section 1. As summarized in Table 3, while distributed stream processing offers superior extreme throughput for cloud-level big data, they require substantial infrastructure refactoring and incur high maintenance costs, which is

highly intrusive to legacy systems. Similarly, adopting TSDBs necessitates a complete overhaul of the historical storage schema.

Table 3. Comparison of industrial data processing paradigms

Methodology	Latency	Throughput Limit	Infrastructure	Intrusiveness
Offline Batch [15]	Minutes/Hours	High	Low (Files)	Low
Distributed Stream [16]	<10ms	Ultra-High	High (Spark/Kafka)	High
TSDB [18]	<10ms	High	Medium (TSDB)	High
Debezium	<20ms	Medium-High	High (Kafka/JVM)	Medium
Proposed System	~20ms	Medium(1600/s)	Low (Pure RDBMS)	Low (None)

4. Conclusion and Future Work

This paper presented a pragmatic real-time data post-processing system for EMU ground platforms. The architecture combines listeners based on polling and triggers with a producer–consumer pipeline to monitor, transform, and persist streaming industrial data with low latency. The system achieves full processing with latency below 20 ms at input rates up to 1600 records per second. The system exhibits no discernible trend ($p > 0.3$) across different initial data volumes and input rates. While the extreme throughput is bounded by the RDBMS I/O rather than distributed big data solutions, our comparative analysis highlights that the proposed system achieves similar low-latency (<20 ms) edge processing to robust CDC middleware like Debezium, but with merely 10% of the memory overhead and zero external infrastructure dependencies. Given the operational profile of subsystems like traction and braking, this performance is sufficient. This capability is readily transferable to other domains. For instance, anomaly detection for industrial robotic arms by several Inertial Measurement Unit (IMU) sensors with 10-Hz sample rate [24]. The enablers for the generalization are the framework's reliance on standard database mechanisms and its independence from domain-specific data structures, which can be adapted through schema customization rather than architectural changes. The proposed architecture offers a viable middle path for industrial digitization: enabling real-time capabilities on legacy infrastructure with minimal intrusion and rapid deployment. Several limitations of the system should be acknowledged. First, the system's reliability under variability or exceptional conditions is not guaranteed. Second, the discussion mainly focuses on localized environments. Multi-tenant scenarios or integration with external analytics platforms may introduce new bottlenecks. Future research could address these limitations by approving exception handling mechanisms against bursty traffic, schema evolution or network latency to guarantee reliability and online adaptability. Moreover, case studies with real conditions and data payloads from other industrial domains beyond railway maintenance can be made to test the generalizability.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Tianyu Xia: Writing—Review & Editing, Conceptualization, Methodology; Beichen Gong: Writing—Original Draft, Software, Validation; Jingxian Ding: Supervision, Project administration; Jianyong Zuo: Supervision, Project administration, Funding acquisition. All authors had approved the final version.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (Grant No.2024YFB4303302), and the Science and Technology Research and Development Programme Topics of China State Railway Group Co., Ltd (Grant No. N2024J012).

References

[1] Zhou, J., Li, P., Zhou, Y., *et al.* (2018). Toward new-generation intelligent manufacturing. *Engineering*, 4(01), 11–20.

[2] Zhang, J. X., Wu, X. L., & Yang Z. (2018). Research and application of industrial data acquisition based on industrial internet of things. *Telecommunications Science*, *34(10)*, 124–129

[3] Ji, Z. (2015). Intelligent manufacturing—Main direction of "made in China 2025". *China Mechanical Engineering*, *26(17)*, 2273–2284.

[4] Zhang, P., Liu, H., *et al*. (2018). Industrial intelligent network: Deepening and upgrading of industrial internet. *Journal of Communications*, *39(12)*, 134–140.

[5] Grieves, M., & Vickers, J. (2016). *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Cham: Springer International Publishing.

[6] Xia, T., Zuo, J., Wang, T., *et al*. (2025). Optimizing blank time in discrete-time sampling: Time-scheduling method for near real-time simulation. *Journal of Simulation*, 1–10.

[7] Sun, L., Shang, Z. Q., & Xia, Y. (2019). Development and prospect of bridge structural health monitoring in the context of big data. *China Journal of Highway and Transport*, *32(11)*, 1–20.

[8] Liu, Q., & Qin, S. J. (2016). Perspectives on big data modeling of process industries. *Acta Automatica Sinica*, *42(02)*, 161–171.

[9] Berlato, M., *et al.* (2025). Digital platforms for the built environment: A systematic review across sectors and scales. *Buildings*, *15(14)*, 2432.

[10] Yu, X., Liu, M., Jiang, X., *et al*. (2019). Industrial internet architecture 2.0. *Computer Integrated Manufacturing Systems*, *25(12)*, 2983–2996.

[11] Yu, Y. (2024). Design and test analysis of data acquisition and processing system in industrial big data. *Technology Innovation and Application*, *14(33)*, 22–25.

[12] Chintapalli, S., *et al*. (2016). Benchmarking Streaming computation engines: Storm, flink and spark streaming. *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).*

[13] Chaves, A. J., Martín, C., & Díaz, M. (2024). Towards flexible data stream collaboration: Federated learning in Kafka-ML. *Internet of Things, 25*, 101036.

[14] Zaharia, M., Das, T., Li, H., *et al.* (2013). Discretized streams: Fault-tolerant streaming computation at scale. *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (pp. 423–438).

[15] Fu, R. (2019). *Design And Implementation of Data Processing System for Industry*. Master's Thesis, Dalian University of Technology, China.

[16] Yu, X., *et al.* (2020). Industrial equipment management system for predictive maintenance. *Computer Science, 47(S2)*, 667–672.

[17] Zhu, K., He, C., Liang, W., *et al*. (2021). High concurrent streaming data processing technology and its application in industrial internet. *Control and Information Technology, 38(5)*, 20–25.

[18] Zheng, M., & Tian, L. (2021). Digital product twin modeling of massive dynamic data based on a time-series database. *Journal of Tsinghua University (Science and Technology), 61(11)*, 1281–1288.

[19] Mastropaolo, A. (2025). When databases age: How SQL server and MySQL handle the test of time. *Computer*, *58(9)*, 4–6.

[20] Guo, S., Chen, S., Li, Q., & Pei, F. (2026). Digital twin-driven approach to multidisciplinary real-time monitoring of flexible production lines. *Journal of Mechanical & Electrical Engineering*, *43(01),* 128–37.

[21] Tang, Y., *et al.* (2025). Efficient storage method on industrial real-time time-series massive measurement point data. *Forging & Stamping Technology, 50(06)*, 268–276.

[22] Tan, H. B., Yi, J. L., Xie, S. L., *et al*. (2023). Mechanical vibration fault diagnosis system based on a producer-

consumer model. *Journal of Hunan University of Technology, (04)*, 34–41.
[23] Huang, S., & Jin, X. (2017). Research on producer/consumer pattern based on labview. *Electronic Science and Technology, 30(09)*, 75–77+81.
[24] Kayan, H., Heartfield R., Rana, O., *et al.* (2025). Real-time anomaly detection for industrial robotic arms using edge computing. *IEEE Internet of Things Journal*, *12(15)*, 29696–29712.