Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Real Time Processing of ITCH Market Data Using Cloud Services at Borsa Istanbul

Emine Dumlu Demircioğlu
Borsa Istanbul (Turkey)
Murat Toprak
Borsa Istanbul (Turkey)
Ahmet Can Ramazanoglu
Borsa Istanbul (Turkey)
Altan Arı
Borsa Istanbul (Turkey)
Berk Dursun Yılmaz
Borsa Istanbul (Turkey)
Mehmet Gokturk
gokturk@gtu.edu.tr

Gebze Technical Universty

Case Report

Keywords: Financial Data Processing, ITCH API Protocol, cloud middleware, real-time analytics

Posted Date: November 18th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10893075/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Real Time Processing of ITCH Market Data Using Cloud Services at Borsa Istanbul

Emine Dumlu Demircioğlu [1], Murat Toprak [2], Ahmet Can Ramazanoglu [3], Altan Arı [4] , Berk Dursun Yılmaz [5] and Mehmet Gokturk [6,*]

[1] Borsa Istanbul, Istanbul, Turkey; emine.dumlu@borsaistanbul.com
[2] Borsa Istanbul, Istanbul, Turkey; murat.toprak@borsaistanbul.com
[3] Borsa Istanbul, Istanbul, Turkey; ahmetcan.ramazanoglu@borsaistanbul.com
[4] Borsa Istanbul, Istanbul, Turkey; altan.ari@borsaistanbul.com
[5] Borsa Istanbul, Istanbul, Turkey; berkdursun.yilmaz@borsaistanbul.com
[6] Gebze Technical University, Gebze, Turkey; gokturk@gtu.edu.tr
* Correspondence: gokturk@gtu.edu.)

Abstract

As financial markets become increasingly complex and data volumes continue to grow, the efficient processing of high-frequency trading (HFT) data has become a critical re-quirement. This study proposes a cloud-based middleware solution for real-time financial data processing. The proposed Java- system parses sequenced binary ITCH API messages which are widely used by financial exchanges, into a human-readable format. The system streams them to Microsoft Azure Event Hub for downstream analytics and storage. The design enables scalable and low-latency data processing in a secure cloud environment through Azure Stream Analytics.

A proof-of-concept implementation was developed and evaluated in a real financial set-ting at Borsa Istanbul, demonstrating strong adaptability across ITCH-based systems and significantly reducing system integration effort. The results highlight best practices for managing sensitive financial data in the cloud and illustrate the advantages of event-driven architectures for real-time financial data management.

Keywords: Financial Data Processing; ITCH API Protocol; cloud middleware; real-time analytics.

1. Introduction

Financial exchanges generate continuous streams of high-frequency market events that must be processed with low latency, high reliability, and strict operational continuity. These event streams include order submissions, cancellations, executions, session updates, and reference data, all of which are consumed by downstream systems such as market monitoring interfaces, trading dashboards, web and mobile applications, compliance tools, and analytical services. The technical problem extends far beyond simple message transport in this setting. The main challenge is to transform protocol-specific exchange messages into structured, reusable, and analytics-ready representations without violating sequencing guarantees or causing operational fragility.

This becomes critical during infrastructure renewal projects in which the core trading or dissemination system changes while surrounding applications remain largely not changed. External and internal consumer systems highly depend on legacy assumptions about message fields, timing, data availability, and derived indicators in many capital market environments,. Replacing each point-to-point integration

separately is costly and difficult to maintain. Therefore, the integration burden shifts toward an intermediary middleware layer that can absorb protocol-specific complexity, regulate data exchange between the renewed core platform and dependent systems, and provide a stable interface for heterogeneous consumers. The technical problem becomes much more than protocol parsing. Basically, it is the design of a scalable data processing architecture that preserving source correctness and producing reusable outputs for multiple downstream use cases.

Binary market data protocols such as ITCH intensify this requirement. ITCH is optimized for compact transmission and high throughput, as opposed to text-based protocols. It is therefore suitable for high-frequency environments but difficult to inspect, analyze, and consume directly. Raw ITCH messages must first be decoded in accordance with protocol specifications. They can only then be converted into structured events before they can support dashboards, monitoring tools, or downstream analytics. In practice, this must take place continuously under sustained event pressure. It must preserve message ordering and timing consistency, and it must remain robust under restart, scaling, and failure conditions. General-purpose stream processing frameworks provide useful building blocks for processing structured event streams. However, they do not by themselves solve protocol-aware ingestion or deterministic transformation of exchange-native binary feeds.

More difficulty comes from the mismatch between traditional enterprise data-processing logic and streaming financial data. Functions that may previously have been implemented as database procedures or post hoc transformations cannot simply be transferred to a live event stream without redesign. Streaming pipelines require different assumptions about state, timing, incremental computation, and output routing. For this reason, real-time financial data transformation requires not only decoding logic but also an architectural separation between protocol-sensitive ingestion and scalable downstream computation.

Cloud-native messaging and stream analytics services provide an attractive foundation for such systems because they support scalable ingestion, elastic processing, monitoring, and multi-consumer distribution. At the same time, capital market software is mission-critical. It must satisfy strict requirements for consistency, continuity, low latency, and recoverability. This creates an architectural issue. Latency-sensitive parsing and sequencing control must remain tightly managed. On the other hand, enrichment, aggregation, and distribution benefit from cloud-side scalability and modularity. Bridging these requirements is a call for a hybrid design in which deterministic protocol-aware ingestion. It is performed close to the exchange environment, while transformation and orchestration are delegated to managed cloud services.

This study addresses that design problem through the development and evaluation of a cloud-based real-time financial data processing architecture centered on the ITCH protocol. Java-based parser applications are used by the proposed system to ingest binary market data feeds, to decode them into structured JSON events, and to publish them to Azure Event Hub raw topics. Then, Azure Stream Analytics jobs consume these raw streams. They compute derived market attributes through SQL-like queries and user-defined functions. Multiple transformed outputs are generated. They are tailored to downstream use cases, including instrument-based, market-based, trade-based, market-by-price, and market-by-order streams. This design keeps original exchange messages for replay, validation, and audit purposes. Analytics-ready event streams for operational use are simultaneously produced.

The study is made in a real deployment context at Borsa Istanbul. Evaluation of the architecture as a practical solution to an industrial integration problem involving high-frequency exchange data became possible. The paper contributes a protocol-aware hybrid edge-cloud architecture, a middleware design pattern based on separation of raw and transformed streams. It makes an industrial case study showing

how managed cloud services can be incorporated into a financial data workflow without reducing the cloud layer to a passive transport channel.

The remainder of the paper is organized as follows. Section 2 reviews the relevant literature on financial market data, stream processing, distributed messaging, and hybrid cloud architectures. Then position of the study within that literature is set. Section 3 presents the proposed architecture and implementation logic. Section 4 provided the case setting and evaluation method. Section 5 reports the results. Section 6 discusses security, validity, design implications. Section 7 concludes the paper, outlines directions for future work.

2. Background and Related Work

2.1. Financial Market Data and ITCH-Based Systems

Modern electronic financial markets operate through continuous streams of order-level events that describe the evolution of the limit order book. These events include order submissions, cancellations, executions, and updates, and they collectively define market behavior at fine temporal granularity. Prior research in market microstructure shows that limit order book data is inherently sequential, state-dependent, and high-volume, requiring strict preservation of message ordering to maintain correctness in derived market states such as bid–ask levels, traded volume, and price evolution [11], [12].

Market data dissemination in such environments is commonly based on specialized binary protocols such as ITCH. Protocol specifications provided by exchanges define compact message formats for order-level activity and reference data [4], [13], [14]. While binary encoding improves transmission efficiency and supports high throughput, it introduces significant complexity in downstream processing. Raw ITCH messages are not directly suitable for analytics, visualization, or multi-application consumption. They must first be decoded based on protocol specifications, then transformed into structured representations before they can be used operationally.

Analysis of order book dynamics is usually the area of interest of existing research. Designing scalable infrastructures for real-time ingestion and transformation of binary market data, even though ITCH-based systems are central to modern exchanges are not well studied. This creates a gap between financial data semantics and the architectural requirements of systems where these systems have to process these data streams under strict latency and reliability constraints.

2.2. Stream Processing Systems and Latency Considerations

The processing of continuous event streams has been well studied in distributed stream processing area. The Dataflow model formalizes the trade-offs between correctness, latency and cost for the processing of unbounded and potentially "out-of-order" data streams [1], [15]. Systems such as MillWheel and Apache Flink extend these by providing stateful stream processing, fault tolerance, event-time semantics by scaling [16], [9], [17]. More recent works and system-level overviews further highlight the importance of managing time, state, and consistency in large-scale streaming architectures [23], [24].

Benchmarking studies emphasize throughput and latency must be both evaluated together to understand system performance under realistic workloads. Particularly, tail latency becomes a critical factor in high-frequency environments where occasional delays can affect downstream decision-making [2], [7]. Earlier stream processing systems such as Aurora and STREAM also established the importance of continuous query processing and real-time responsiveness in data stream management systems [28], [29], [30].

Most stream processing frameworks assume that input data is already structured despite these advances. In financial exchange environments, this assumption does not hold. Binary protocol messages must first be decoded, validated, and reconstructed before stream processing can begin. This introduces an additional architectural requirement that is not directly addressed in general-purpose stream processing literature.

Furthermore, traditional enterprise data-processing techniques, such as database procedures or batch-based transformations, cannot be directly applied to streaming financial data. Instead of this, computations must be reformulated as incremental, real-time transformations operating under strict latency constraints. This distinction determined the need for architectural separation between ingestion, transformation, and output generation stages..

2.3. Distributed Messaging and Event Ingestion Architectures

High-throughput event pipelines are typically supported by distributed messaging systems that provide scalability, partitioning, and fault tolerance. Log-based messaging platforms enable scalable ingestion by distributing events across partitions while maintaining ordering guarantees within each partition such as Apache Kafka [8], [18]. Coordination services support distributed state management, leader election, and recovery mechanisms in large-scale systems such as ZooKeeper [10], [19].

These technologies show that reliable event ingestion requires explicit management of ordering, replayability, and consumer independence. In another words, modern data infrastructure research emphasizes the importance of designing systems that support durable logs, event replay, and also loosely coupled processing pipelines [22].

However, these systems assume that upstream data is already structured and ready for consumption. This assumption is not valid in the context of financial market data. Binary protocol feeds such as ITCH require a protocol-aware ingestion layer that can decode and structure messages before they enter distributed messaging systems. Without the existence of such a layer, downstream consumers are required to implement their own parsing and reconstruction logic, leading to duplication and also increased system complexity.

2.4. Cloud-Based Stream Analytics and Edge–Cloud Architectures

Cloud-native services such as Azure Event Hub and Azure Stream Analytics extend distributed messaging and stream processing capabilities by providing managed infrastructures for event ingestion, transformation, and distribution [5], [6]. These platforms support scalable, elastic, and multi-consumer architectures in which different applications can consume different derived views of the same underlying data stream.

At the same time, financial data processing introduces constraints that are not always aligned with cloud-native design principles. Capital market systems require predictable latency, strict ordering guarantees, and operational continuity, whereas cloud platforms emphasize elasticity and abstraction. This tension has led to increased interest in hybrid edge–cloud architectures, where latency-sensitive operations are performed close to the data source and computationally intensive transformations are delegated to the cloud.

Recent work on edge–cloud integration demonstrates that such architectures can balance responsiveness and scalability by partitioning processing tasks across system boundaries [20], [21]. In the context of financial data pipelines, this approach enables deterministic protocol-aware ingestion at the edge while leveraging cloud-based services for scalable transformation and distribution.

In addition, modern streaming systems research highlights the importance of integrating incremental analytics and continuous query processing into distributed architectures [27], [31]. These capabilities are

particularly relevant for financial applications that require real-time computation of derived market indicators.

2.5. Positioning of This Study

This study addresses a gap between financial market data processing, distributed stream processing, and cloud-based event architectures. Existing approaches to ITCH data processing generally are in three categories. These are batch-oriented pipelines designed for historical analysis, proprietary in-house parsers optimized for specific environments, and partial open-source decoding tools which do not provide scalable transformation and distribution capabilities.

These approaches do not completely study the combined requirements of protocol-aware ingestion, deterministic sequencing. They do not consider replayable raw data preservation, and generation of multiple analytics-ready outputs for heterogeneous downstream consumers. They do not provide a unified architecture that integrates binary protocol decoding with scalable, cloud-based stream transformation.

This study makes contribution by introducing a hybrid edge–cloud middleware architecture that explicitly separates protocol-aware ingestion from cloud-based transformation and distribution. Raw event streams for replay, validation, and audit purposes were preserved while multiple transformed output streams tailored to downstream applications are simultaneously generated. The architecture provides a reusable pattern for modernizing ITCH-based financial data infrastructures under real-time operating conditions by combining deterministic edge-side processing with scalable cloud-based analytics.

3. Proposed System Architecture

3.1. Architectural Overview

The proposed architecture was designed to process binary ITCH market data in real time and to expose the resulting information to multiple downstream applications through a scalable event-driven pipeline. A recurring integration problem in financial market infrastructures is discussed. Exchange-side data dissemination is protocol-specific and latency-sensitive, whereas downstream systems require structured, reusable, and analytics-ready data products. The architecture introduces a middleware layer that separates protocol-aware ingestion from scalable transformation and distribution to fulfill the need.

The system consists of four logical layers at high level. The first layer is the exchange data source disseminating market data through ITCH feeds. The second layer is an edge-side ingestion layer implemented as Java-based parser applications. These components receive binary ITCH messages, decode them according to protocol specifications, and convert them into structured JSON events. The third layer is a cloud-based messaging and transformation layer. It is built on Azure Event Hub and Azure Stream Analytics. It is responsible from event persistence, stream enrichment, and generation of multiple derived outputs. The final layer is the consumer layer, where structured streams are delivered to dashboards, monitoring systems, web and mobile applications, analytical services.

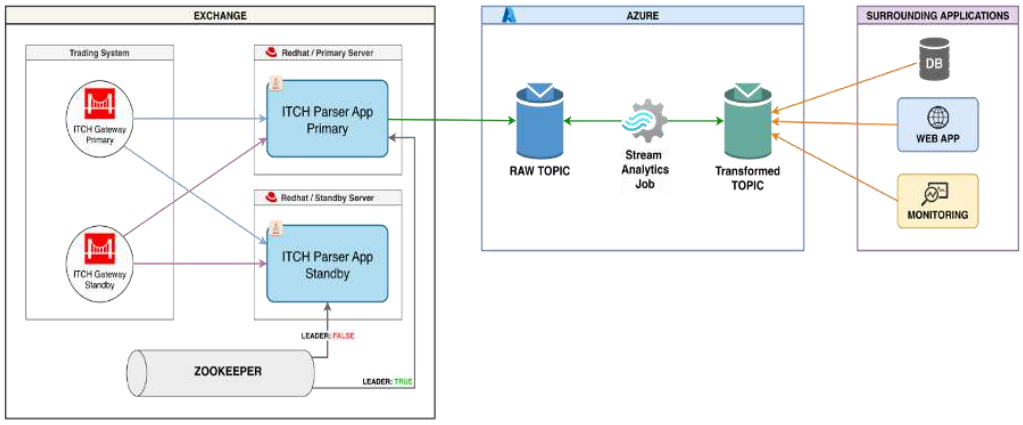


Figure 1. High-level architecture of the proposed hybrid edge-cloud middleware.

This layered design serves two complementary objectives. It keeps deterministic, low-latency processing at the ingestion boundary. At that point protocol correctness and message ordering are critical. It enables scalable, modular, and multi-consumer processing in the cloud layer at the same time, where derived data products are generated and also distributed. The resulting architecture converts exchange-native binary data into reusable event streams without requirement of repeated point-to-point integrations.

3.2. Design Rationale

The first requirement is protocol-aware ingestion. Rather than direct analytical use, ITCH messages are encoded in a compact binary format optimized for transmission efficiency. These messages must be decoded deterministically in accordance with protocol specifications before any downstream processing can happen. Any inconsistency at this stage would propagate to all dependent systems.

The second requirement is the separation of raw and transformed data. Transformation logic is tightly coupled with ingestion in many legacy systems, or it is implemented through database procedures after data persistence. This does not scale to streaming environments. Streaming pipelines require continuous transformation under time constraints. Therefore, raw message preservation and derived data generation must be handled as distinct concerns. The proposed architecture keeps raw event streams for replay, validation, and audit purposes while generating separate transformed streams tailored to downstream applications.

The third requirement is multi-consumer interoperability. Different consumer systems require different abstractions of the same underlying market activity. Some require instrument-level summaries, others require trade-level data, and others require full order-book representations. Implementing these transformations independently in each downstream system creates duplication and increases integration complexity. The architecture therefore centralizes transformation logic and produces multiple domain-specific output streams from a single ingestion pipeline.

A further design consideration concerns the mismatch between traditional data-processing approaches and streaming financial data. Transformation logic that may previously have been implemented as database procedures cannot be directly applied to high-frequency event streams. Instead, computations must be expressed as continuous, incremental transformations operating under latency constraints. This motivates the architectural separation between ingestion and transformation layers, allowing each to be optimized independently..

3.3. End-to-End Processing Flow

The end-to-end processing flow begins with the dissemination of market data through ITCH feeds from the exchange. Separate feeds are used for Market by Order (MBO) and Market by Price (MBP) streams in the current deployment,. Each feed is handled by an independent parser process configured with its own connection parameters and output destination. Fault isolation and operational control are improved by this separation.

Each parser continuously listens for incoming binary messages. They then decode them into structured internal representations with protocol-aligned message builders. Then the decoded messages are serialized into JSON format and augmented with processing metadata. The metadata includes parser-side timestamps. These events are immediately published to raw topics in Azure Event Hub. The architecture keeps source semantics at this stage. It avoids embedding domain-specific calculations into the ingestion layer.

The raw topics serve as the authoritative input streams for downstream processing. Azure Stream Analytics jobs consume these streams and apply continuous transformations, including filtering, aggregation, enrichment, and routing. Multiple derived streams corresponding to different analytical and operational requirements generated by these transformations.

The transformed outputs are written to dedicated Event Hub topics. They are then delivered to downstream systems. Without requiring each consumer to perform protocol decoding or market reconstruction independently, this establishes a one-to-many distribution model where a single ingestion pipeline supporting multiple consumers. The processing flow then reduces duplication and centralizes interpretation of exchange events within the middleware layer. These are shown in Figure 2 with more detail explaining internal processing pipeline of the middleware architecture.

3.4. Edge-Side ITCH Parsing Layer

The ingestion layer is implemented as a set of Java-based parser applications. Thesedefine the deterministic boundary of the system. These components are responsible for receiving binary ITCH messages. They decode them according to the official protocol specification and forward in structured form to the cloud messaging layer.

The role of the parser is intentionally limited. It performs protocol decoding and message forwarding but does not execute computationally intensive market calculations. The proposed design keeps the ingestion layer lightweight, reducing latency at the point where ordering and timing guarantees are critical.

A producer–consumer pattern is used internally to separate message reception, decoding and transmission. This improves throughput stability and reduces contention between I/O and serialization operations. Configuration parameters such as connection endpoints, feed types, and output topics are externally defined. This allows flexible deployment across environments without modifying application code.

Message sequencing and decoding correctness are preserved before events enter the distributed processing environment through isolating protocol-aware ingestion in this layer. This separation is essential since errors introduced at this stage would propagate to all downstream outputs.

3.5. Cloud Messaging Layer

The cloud messaging layer is built on Azure Event Hub. It provides scalable, durable ingestion for both raw and transformed event streams. Separate raw topics are maintained for different feed types, preserving the original decoded ITCH messages as immutable event logs. These raw streams serve as the system's authoritative data source. They support replay, validation, and troubleshooting.

A second set of topics stores transformed outputs generated by the stream processing layer. These transformed topics contain enriched, application-ready data rather than protocol-level messages. The architecture avoids coupling protocol preservation with analytical computation by separating raw and transformed streams. Independent evolution of transformation logic is therefore possible.

Retention policies are configured to match with operational trading periods, eproviding that event streams remain available for recovery and validation within a bounded time horizon. While preventing cross-session data contamination this design supports consistent downstream behavior

3.6. Stream Transformation Layer

The transformation layer is implemented using Azure Stream Analytics and functions as the semantic core of the middleware. This layer consumes raw event streams. It applies continuous transformations to generate domain-specific data products.

Transformations are expressed using SQL-like queries combined with user-defined functions for domain-specific computations. These operations include filtering invalid messages, computing derived market indicators, aggregating event streams. They also route outputs to appropriate destinations. Examples of derived attributes include open price, high and low prices, traded volume, trade counts, and order-book depth.

Instead of producing a single aggregated output, proposed architecture generates multiple specialized streams which correspond to different analytical needs. The functional decomposition improves modularity. It also allows downstream systems to subscribe only to the data relevant to their use case.

The transformation layer therefore converts protocol-level events into structured, reusable representations that can support real-time analytics, monitoring, and application-level processing without requiring each consumer to implement its own transformation logic.

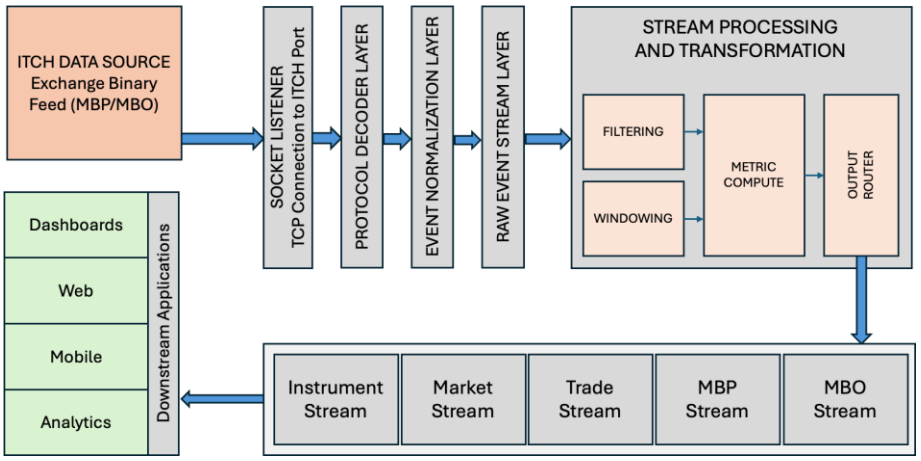


Figure 2. Internal Processing Pipeline of the Middleware Architecture.

3.7. Output Stream Model

The system produces five categories of transformed output streams as shown in Figure 2 and Figure 3 respectively.

The instrument-based stream organizes data at the level of individual financial instruments and includes attributes such as bid and ask levels, last price, traded volume, and daily price extrema. This stream supports market monitoring and instrument-level analytics.

The market-based stream provides aggregate indicators at the market or board level. These include total volume, total value, trade counts, and session state. This stream supports high-level monitoring of overall market conditions.

The trade-based stream captures executed transactions and includes execution price, volume, and trade metadata. This stream supports transaction monitoring, auditing, and downstream analytics.

The Market by Price stream aggregates order-book data by price levels, enabling analysis of market depth and liquidity.

The Market by Order stream preserves order-level granularity. It allows detailed reconstruction of order-book dynamics and advanced microstructure analysis.

These outputs show that the proposed architecture is not only limited to data transport. It functions as a transformation framework that generates multiple reusable abstractions of the same underlying market activity.

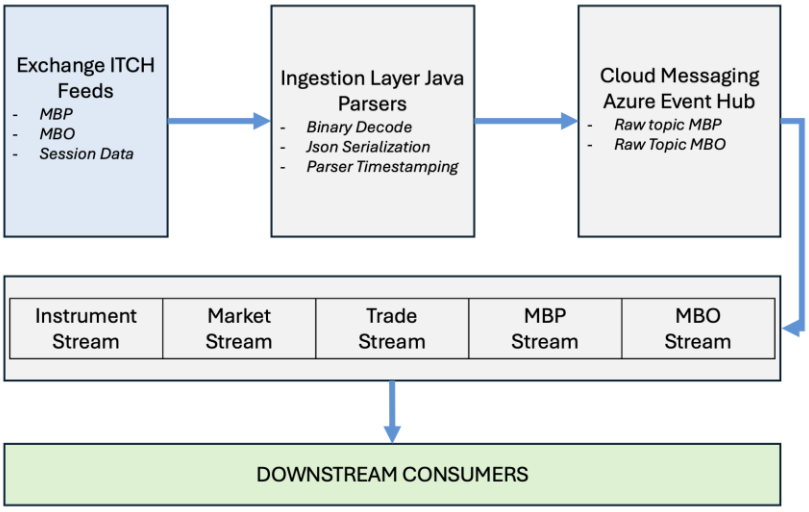


Figure 3. High-Level architecture of the proposed edge-cloud middleware.

The proposed architecture has four defining properties in brief. It keeps protocol awareness at the ingestion boundary through performing deterministic decoding before data enters the distributed environment. It separates raw event preservation from transformed data generation. This enables replayability and auditability. It supports one-to-many distribution through modular output streams, reducing integration overhead for downstream systems. Finally, it combines edge-side determinism with cloud-side scalability, making the system to operate under both latency-sensitive and high-throughput conditions.

4. Method

4.1. System Setup and Data Pipeline

The experimental setup was designed to evaluate a real-time financial data processing pipeline operating on binary ITCH market data. The system consists of an edge-side ingestion layer and a cloud-

based processing layer. At the ingestion boundary, Java-based parser applications were deployed to receive ITCH messages from dedicated exchange feeds. These parsers decode binary protocol messages into structured JSON events and immediately publish them to Azure Event Hub. Each message is augmented with a parser-side timestamp at the moment of ingestion.

Two independent input streams were used in the experiments: Market by Order (MBO) and Market by Price (MBP). Each stream was processed by a separate parser instance configured with its own connection parameters and output topic. This separation improves fault isolation and enables concurrent processing of different feed types without cross-interference.

The cloud-side infrastructure was implemented using Azure Event Hub and Azure Stream Analytics. Event Hub was used to store raw event streams as well as transformed outputs. Raw topics preserve the original decoded ITCH messages and serve as the authoritative input for downstream processing. Transformed topics store enriched, application-ready data generated by the transformation layer. The system was deployed on an Azure Red Hat Enterprise Linux (RHEL) with 64-bit operating system virtual machine with 28 CPU Core, 64GB memory, 250GB disk virtual machine CPU / memory configuration], and autoscaling was enabled for stream processing jobs to reflect production-like conditions.

4.2. Processing and Transformation Framework

Stream processing was implemented using Azure Stream Analytics jobs that continuously consume raw Event Hub topics and generate multiple derived output streams. Transformations were defined using SQL-like queries combined with JavaScript user-defined functions to support domain-specific computations that cannot be expressed declaratively.

The transformation layer performs filtering, aggregation, enrichment, and routing operations on the incoming event streams. Derived attributes include key market indicators such as open price, high and low prices, traded volume, trade counts, and order-book depth. All computations are executed incrementally on streaming data, without relying on post hoc database processing. This design reflects the requirements of real-time financial systems, where transformations must operate under strict latency constraints.

The system produces five distinct transformed output streams: instrument-based, market-based, trade-based, market-by-price, and market-by-order topics. Each stream corresponds to a specific analytical abstraction of the same underlying market activity. This design enables multiple downstream systems to consume different views of the data without duplicating transformation logic.

4.3. Evaluation Methodology

The evaluation was designed to measure system performance under sustained high-frequency workloads, focusing on latency, scalability, and reliability. End-to-end latency was defined as the time required for a message to travel from the ingestion point at the parser layer to the final transformed output stage. Latency was computed as the difference between the parser-side timestamp for each message. The timestamp recorded after transformation and publication to the output topic as follows:

$$\text{Latency_E2E}(i) = \text{Time_output} - \text{Time_input} \quad (1)$$

This measurement captures the cumulative delay introduced by parsing, event ingestion, stream processing, and output publishing. This measurement framework also allows decomposition of latency into these individual stages using system logs and Azure diagnostics tools.

In order to evaluate scalability of the solution, the input message rate was varied across multiple workload levels ranging from 1000 to 3000 messages per second. The system was operated for a sustained

period to observe stable behavior at each level. Metrics collected included average latency, maximum latency, percentile latency (p95 and p99), and throughput consistency. The identification of both the stable operating region of the system and the onset of performance degradation under higher load conditions became possible with this approach.

Reliability was evaluated by monitoring message continuity, ordering consistency, and potential data loss during processing. Raw Event Hub topics were used as the reference source to verify completeness of processed streams. Transformed outputs were compared against raw inputs to make sure that all relevant events were correctly propagated through the pipeline.

The system was monitored for message loss ordering anomalies and recovery behavior under continuous operation. Message counts in the raw topics are consistently compared with the parser's produced message counts to detect potential message loss. In addition to message count validation, end-to-end latency is continuously monitored using parser-injected timestamps. This made precise measurement of processing delays possible. Furthermore, message ordering is verified using ITCH sequence numbers to ensure that no reordering occurs during processing. In order to confirm that the system can resume processing without data loss, recovery scenarios, including parser or stream job failures, are also evaluated

Computed results in transformed topics are then periodically validated against raw data to ensure processing accuracy.

5. Results

After the system configuration was completed, the ITCH parser applications and Azure Stream Analytics jobs were executed to process incoming market data streams in real time. The results are presented from three complementary perspectives. Correctness of stream transformation, end-to-end latency under sustained load, and scaling behavior as the input rate increases. This structure reflects the practical objective of the middleware. The system is expected decode binary ITCH messages correctly and convert them into reusable downstream data products with controlled latency under operational workloads.

5.1. Stream Transformation Outputs

The first result concerns the correctness and completeness of the transformation pipeline. The parser layer successfully converted binary ITCH messages from both Market by Price and Market by Order feeds into structured JSON records stored in raw Event Hub topics. Representative MBP and MBO raw messages are presented in Tables 1 and 2, respectively. These examples show that the parser preserves the structural content of protocol-level messages while converting them into a format suitable for downstream stream processing.

Table 1. Sample MBP ITCH Message in JSON Format.

MBP Raw Topic
{ "orderBook": 2092, "attributeValue": "1150", "attributeType": "L", "reason": " ", "type": "X", "seqNum": 538, "systemTimestamp": "1688519155295658000", "EventProcessedUtcTime": "2023-07-05T08:59:34.0641266Z",

```
    "PartitionId": 0,
    "EventEnqueuedUtcTime": "2023-07-05T01:06:35.2170000Z"
  }
```


Table 2. Sample MBO ITCH Message in JSON format.

MBO Raw Topic


```
{
    "orderNumber": "202306160032654397",
    "side": "BUY",
    "quantity": "30",
    "orderBook": 102,
    "price": "6300",
    "yield": "9223372036854775807",
    "value": "189000",
    "mMUser": "N",
    "eventType": "ADD",
    "type": "A",
    "SeqNum": 2483,
    "messageTimestamp": "1686896441051310000",
    "EventProcessedUtcTime": "2023-06-16T09:57:09.9141705Z",
    "PartitionId": 0,
    "EventEnqueuedUtcTime": "2023-06-16T06:20:41.0500000Z
}
```


The MBO and MBP raw topics serve as the input sources for the Azure Stream Analytics jobs. Once these jobs are executed, the incoming streams are transformed into five distinct output families: instrument-based, market-based, trade-based, market-by-price, and market-by-order streams. The transformed topic structure and representative example outputs are summarized in Table 3. These outputs confirm that the proposed architecture performs more than protocol translation. It also functions as a semantic transformation layer that converts exchange-native binary data into multiple application-ready views of market activity.

Table 3. The transformed topic structure and representative example outputs.

Transform Topics	
Transform Topic Name	**Transform Topic Content**

Instrument/Symbol-Based	**Transformed Topic values:** (derived from the MBP Raw Topic)
	orderBook, securityCode, securityName, isin, currency, board, sessionState, assetType,
	underlying, suspended, bidVolume, bidPrice, askPrice, askVolume, lastPrice, previousClose, change, changePercentage, value, volume, trades, openPrice, highPrice, lowPrice,
	messageTimeStamp
	Example TransformedTopic Message in JSON format:
	{
	"orderbook": 102,"securityCode": "ABC","securityName": "ABC Bank","isin": "ABC1213",
	"currency": "USD","board":"EQTY","sessionState": "OPEN",
	"assetType":"E",
	"underlying":0,"suspended":"NO","bidVolume": 40,"bidPrice":1.1,"askPrice":0.99,
	"askVolume":10,"lastPrice":0.99, "previousClose":1.1,"change":0.99,
	"changePercentage":null,"value": 62.9,"volume": 60,"trades": 6,"openPrice": 1.1,
	"highPrice":1.1,"lowPrice":0.99,"theoreticalOpeningPrice":1,
	"messageTimestamp":"2023-07-18T06:10:03.0000000Z",
	"EventProcessedUtcTime": "2023-07-18T07:56:14.0669755Z",
	"EventEnqueuedUtcTime": "2023-07-18T06:10:13.7070000Z"
	}

Market-Based	**Transform Topic values:** (derived from the MBP Raw Topic)
	board, totalVolume, totalValue, totalTrades, sessionState, messageTimestamp
	Example TransformedTopic Message in JSON format:
	{
	"board": "EQTY","totalVolume": 20,"totalValue": 20,"totalTrades": 2,
	"sessionState":"OPEN","messageTimestamp":"2023-07-17T07:54:50.0000000Z",
	"EventProcessedUtcTime": "2023-07-17T12:01:00.1742160Z",
	"EventEnqueuedUtcTime": "2023-07-17T07:54:50.6920000Z"
	}

Trade-Based	**Transform Topic values:** (derived from the MBP Raw Topic)
	board, securityCode, securityName, executionYield, executionPrice, executedVolume,
	tradeIndicator, matchNumber, tradeValue, messageTimestamp
	Example TransformedTopic Message in JSON format:
	{
	"board": "EQTY","orderbook": 102,"securityCode": "ABC","securityName": "ABC Bank",
	"executionYield": 9223372036854776, “executionPrice": 1.1,"executedVolume": 10,
	"tradeIndicator": " ","matchNumber": "202307180000002601",
	"tradeValue": 11,
	"messageTimestamp": "2023-07-18T06:10:00.0000000Z",
	"EventProcessedUtcTime": "2023-07-18T07:56:07.3012295Z",
	"EventEnqueuedUtcTime": "2023-07-18T06:10:01.4240000Z"
	}

MBP (Market By Price)	**Transform Topic values:** (derived from the MBP Raw Topic) orderBook, securityCode, securityName, board, assetType, side, orderPrice, totalVolume, numberOfOrders, messageTimeStamp **Example TransformedTopic Message in JSON format:** { "orderbook": 102,"securityCode": "ABC","securityName": "ABC Bank","board": "EQTY", "assetType": "E","side": "B", "orderPrice": 990,"totalVolume": 30,"numberOfOrders": 1, "messageTimestamp": "2023-07-18T06:10:00.0000000Z", "EventProcessedUtcTime": "2023-07-18T08:06:24.9825175Z",

	"EventEnqueuedUtcTime": "2023-07-18T06:10:01.5320000Z" }

MBO (Market By Order)	**Transform Topic values:** (derived from the MBO Raw Topic)
	type, eventType, orderNumber, securityCode, securityName, side, board, orderPrice,
	orderYield, orderVolume, executedVolume, executionPrice
	Example Transformed Topic Message in JSON format:
	{
	"type": "add","eventType": "add","orderNumber": 4587321,"securityCode": "ABC",
	"securityName": "Bank","side": "B","board": "EQTY","orderPrice": 990,"orderYield": 0,
	"orderVolume": 30,"executedVolume": 30,"executionPrice": 990
	}

The instrument-based stream provides security-level indicators such as bid and ask levels, last price, previous close, value, volume, trade count, open price, and daily price extrema. The market-based stream provides aggregate board-level indicators such as total volume, total value, total trade count, and session state. The trade-based stream captures execution-level information including execution price, executed volume, trade value, and match identifiers. The market-by-price and market-by-order outputs keep depth-oriented and order-level representations required by downstream monitoring systems and analytical applications. These outputs demonstrate that a single ingestion pipeline can support heterogeneous consumer requirements without forcing each downstream system to implement its own ITCH parsing and market reconstruction logic.

5.2. End to End Latency

The second result concerns timing behavior across the pipeline. The system processes high-frequency ITCH market data streams at an average rate of 3,000 messages per second. Raw ITCH messages are first parsed by the Java-based parser applications and published to source Azure Event Hub topics. Azure Stream Analytics jobs then consume these raw events, apply transformation logic, and publish the processed results to downstream Event Hub topics.

End-to-end latency was defined as the time required for an ITCH message to travel from the ingestion point at the parser layer to the final transformed output topic. Each ITCH message is augmented with an ingestion timestamp at the parser level before being sent to the source Event Hub.

$$\text{Time}\{\text{input}\} = \text{SystemTime}\{\text{parser}\}, \tag{2}$$

After the transformation stage, the output message includes a second timestamp recorded at the downstream Event Hub.

$$\text{Time}\{\text{output}\} = \text{SystemTime}\{\text{eventHub}\}, \tag{3}$$

The end-to-end latency for each message is therefore computed as the difference between these two timestamps.

$$\text{Latency}\{\text{E2E}\}(i) = \text{Time}\{\text{output}\} - \text{Time}\{\text{input}\}, \tag{4}$$

This formulation captures the cumulative delay introduced by parsing overhead, Event Hub ingestion delay, stream processing time, and downstream publishing latency.

The system maintained stable end-to-end latency by leveraging Azure Stream Analytics for scalable stream transformation, under a sustained ITCH message rate of 3,000 events per second,. The measured latency values under different workload levels are summarized in Table 4.

Table 4. Latency Results.

Input Rate (msg/s)	Avg Latency (ms)	Max Latency (ms)
2000	170	380
3000	220	500
4000	280	650

.

At 3,000 messages per second, the measured average end-to-end latency was 220 ms and the maximum observed latency was 500 ms. These values indicate that the architecture maintains controlled timing behavior while executing continuous parsing, transformation, and downstream publication in the cloud-based pipeline. This result is important because it shows that the proposed separation between lightweight edge-side decoding. Furthermore, cloud-side transformation can support real-time market data processing without introducing excessive delay at the target operational traffic level.

5.3. Scaling Behavior Under Increasing Load

The third result concerns scalability under increasing traffic. As shown in Table 5, average latency increased from 170 ms at 2,000 messages per second to 220 ms at 3,000 messages per second and 280 ms at 4,000 messages per second. Maximum latency increased from 380 ms to 500 ms and then to 650 ms across the same workload levels.

This pattern shows that the pipeline degrades in a controlled manner as load increases. Latency rises as expected when ingestion and processing pressure grow. However, the increase remains bounded within the evaluated range. The system remained operational at 4,000 messages per second, while the evaluated deployment was configured to guarantee stable sustained processing at 3,000 messages per second. This distinction is important because it indicates that the architecture can tolerate higher short-term traffic intensity while maintaining a validated steady-state operating point at 3,000 messages per second.

The results therefore support the central architectural claim of the paper. The proposed middleware functions not only as a protocol-aware ingestion mechanism but also as a scalable transformation and distribution layer for high-frequency financial data. The observed latency characteristics show that the architecture remains usable under increased load and can serve as a practical modernization layer for ITCH-based market data environments.

5.4. Overall Interpretation

The results cumulatively suggest that the proposed system achieves three concrete outcomes. First, it correctly transforms binary ITCH feeds into five reusable downstream stream families, demonstrating that the middleware can serve multiple application classes from a single ingestion process. Second, it sustains an average processing rate of 3,000 messages per second with an average end-to-end latency of 220 ms under the evaluated deployment. Third, it exhibits controlled latency growth as traffic increases from 2,000 to 4,000 messages per second.

These findings indicate that the architecture is suitable as a practical middleware layer for financial data environments that require protocol-aware ingestion, modular downstream data products, and controlled real-time performance. The current results are strongest in demonstrating architectural feasibility and operational stability in a real exchange-oriented setting.

6. Security and Cloud Considerations

Storing and processing it securely in the cloud is a priority, given the sensitive nature of financial data. Azure Event Hub provides robust access controls and encryption at rest and in transit. Stream Analytics integrates seamlessly while maintaining compliance with industry standards. This makes the overall

solution suitable for production-grade financial systems. Moreover, access to the Event Hub is restricted using its Shared Access Policy feature. A shared access signature (SAS) provides a way to grant limited access to resources in Event Hubs namespace. SAS guards' access to Event Hubs resources based on authorization rules. These rules are configured either on a namespace, or an event hub. All communication between EventHub and Stream Analytics is encrypted, providing compliance with financial security standards.

6.1. Threats to Validity

The case study was conducted within a single trading company and focused exclusively on the ITCH protocol. While ITCH is widely used, non-trivial API protocols in a financial domain, future studies should evaluate additional protocols to improve generalizability. Therefore, our evaluation is subject to an external validity threat.

The ITCH API data is specification-oriented. Any changes occurring in the associated dependency must be incorporated into the application to ensure consistency with the API specification.

7. Discussion

This study presented a cloud-based, real-time financial data processing system built using the ITCH API protocol. The middleware solution that is proposed, parses high-frequency binary ITCH messages using a Java-based parser and streams the structured output to Microsoft Azure Event Hub. The system enables scalable, low-latency processing of market data in a secure cloud environment using Azure Stream Analytics. The use of Azure Event Hub and Stream Analytics provides a robust, event-driven data pipeline capable of handling high-throughput financial data with low latency. Key market indicators as trade volumes, execution prices, orderbook depth, open/high/low prices and market state metrics were computed in realtime.

The system is adaptable to different exchanges using the ITCH protocol. The results confirm that the solution is suitable for production-grade financial systems operating in high-frequency trading environment. The proposed solution offers a practical and scalable modernization path for exchanges that are targeting to increase trading volume, improve investor engagement and meet international data standards.

Several enhancements are planned for future versions of the system including schema versioning for evolving ITCH formats, alerting and monitoring via Azure Monitor and Log Analytics, AI-based anomaly detection identifying unusual trading patterns in realtime.

Abbreviations

The following abbreviations are used in this manuscript:

ITCH	High performance binary protocol for outbound market data dissemination
MBO	Market By Order
OUCH	Low-latency binary protocol for inbound order entry and management
MBP	Market By Price
SAS	Shared Access Signature

.

Funding Declaration

There was no specific funding for this work. We thank our colleagues for their emotional support during this wotk.

Ethics, Consent to Participate, and Consent to Publish Declaration

Ethics, Consent to Participate, and Consent to Publish declarations: not applicable

Clinical Trial Number

The work did not include clinical trials.

References

1. Akidau, T., Balikov, A., Bekiroğlu, K., Chambers, C., Chernyak, S., Fernández-Molina, J., ... & Whittle, S. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive Scale, Unbounded, Out-of-Order Data Processing. Proceedings of the VLDB Endowment.
2. Das, S., Agrawal, D., & El Abbadi, A. (2014). Turmeric: Metrics for Characterizing Latency of Streaming Data Systems. ACM Symposium on Cloud Computing.
3. Xu, X., Yu, J., & Cao, B. (2019). Performance Modeling and SLA Compliance in Cloud Stream Processing. IEEE Transactions on Cloud Computing.
4. ITCH-API Protocol. Available online: https://www.borsaistanbul.com/files/bistech-itch-protocol-specification.pdf.
5. Azure Event Hubs. Available online: https://azure.microsoft.com/en-us/products/event-hubs.
6. Shared Access Signatures. Available online: https://learn.microsoft.com/en-us/azure/event-hubs/authorize-access-shared-access-signature
7. Fu, T., Kwon, Y., & Sadoghi, M. (2017). StreamBench: Towards Benchmarking Modern Distributed Stream-Processing Platforms. ICDE
8. Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: a Distributed Messaging System for Log Processing. NetDB Symposium
9. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 38(4), 28–38.
10. Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). ZooKeeper: Wait-free coordination for Internet-scale systems. Proceedings of the 2010 USENIX Annual Technical Conference, 145–158.
11. Gould, M. D., Porter, M. A., Williams, S., McDonald, M., Fenn, D. J., & Howison, S. D. (2013). Limit order books. Quantitative Finance, 13(11), 1709–1742. https://doi.org/10.1080/14697688.2013.803148
12. Cartea, Á., Jaimungal, S., & Penalva, J. (2015). Algorithmic and High-Frequency Trading. Cambridge University Press. ISBN 9781107091146.
13. Nasdaq. (2014). NASDAQ TotalView-ITCH 4.1 Specification. Nasdaq Trader.
14. Borsa İstanbul. (2025). BISTECH ITCH Protocol Specification. Borsa İstanbul.
15. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., & Whittle, S. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. Proceedings of the VLDB Endowment, 8(12), 1792–1803. https://doi.org/10.14778/2824032.2824076
16. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., & Whittle, S. (2013). MillWheel: Fault-Tolerant Stream Processing at Internet Scale. Proceedings of the VLDB Endowment, 6(11), 1033–1044. https://doi.org/10.14778/2536222.2536229
17. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin, 38(4), 28–38.
18. Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A Distributed Messaging System for Log Processing. In Proceedings of NetDB'11.

19. Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In USENIX Annual Technical Conference (USENIX ATC 10).
20. Wang, X., Yang, P., Gill, S. S., Zhou, H., & others. (2022). An edge–cloud integrated framework for flexible and adaptive stream analytics. Future Generation Computer Systems. https://doi.org/10.1016/j.future.2022.06.020
21. Esteves, S., Veiga, L., Leitão, J., & others. (2017). Empowering Stream Processing through Edge Clouds. In Middleware '17 Industry Track. https://doi.org/10.1145/3156655.3156661
22. Kleppmann, Martin. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017..
23. Akidau, Tyler, Slava Chernyak, and Reuven Lax. Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. O'Reilly Media, 2018.
24. Isah, Haruna, et al. "A Survey of Distributed Data Stream Processing Frameworks." IEEE Access, vol. 7, 2019, pp. 154300–154316.
25. Gulisano, Vincenzo, et al. "StreamCloud: An Elastic and Scalable Data Streaming System." IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, 2012, pp. 2351–2365.
26. Zaharia, Matei, et al. "Discretized Streams: Fault-Tolerant Streaming Computation at Scale." Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 423–438.
27. Chandramouli, Badrish, et al. "Trill: A High-Performance Incremental Query Processor for Diverse Analytics." Proceedings of the VLDB Endowment, 2015.
28. Abadi, Daniel J., et al. "Aurora: A New Model and Architecture for Data Stream Management." The VLDB Journal, vol. 12, no. 2, 2003, pp. 120–139.
29. Arasu, Arvind, et al. "STREAM: The Stanford Stream Data Manager (Demonstration Description)." Proceedings of the ACM SIGMOD International Conference on Management of Data, 2003, p. 665.
30. Stonebraker, Michael, Uğur Çetintemel, and Stanley B. Zdonik. "The 8 Requirements of Real-Time Stream Processing." SIGMOD Record, vol. 34, no. 4, 2005, pp. 42–47.
31. Chandramouli, Badrish, et al. "Quill: Efficient, Transferable, and Rich Analytics at Scale." Proceedings of the VLDB Endowment, vol. 9, no. 14, 2016, pp. 1623–1634.