International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P144
Eureka Vision Publication | Volume 7, Issue 1, 300-304 2026

Original Article

High-Throughput Data Processing Architectures on AWS for Logistics and Transportation Systems

Mohammed Abdul Mannan Ansari
Independent Researcher, USA.

Received On: 30/01/2026 ***Revised On: 02/03/2026*** ***Accepted On: 07/03/2026*** ***Published On: 13/03/2026***

Abstract - Modern logistics and transportation systems generate massive volumes of data from IoT sensors, GPS trackers, fleet management systems, and supply chain operations. This White paper presents comprehensive architectural patterns for implementing high-throughput data processing systems on Amazon Web Services (AWS) that handle both streaming and batch workloads. The proposed architectures leverage AWS services, including Amazon Kinesis Data Streams, AWS Lambda, and Amazon EMR to provide scalable, reliable, and cost-effective solutions. Key focus areas include real-time streaming processing for operational insights, batch processing for historical analysis, and fault tolerance mechanisms. Implementation patterns demonstrate processing capabilities exceeding 50,000 messages per second while maintaining sub-second latency for critical logistics operations.

Keywords - AWS, High-Throughput Processing, Logistics Systems, Stream Processing, Batch Processing, Amazon Kinesis, Scalability, Iot Data Processing.

1. Introduction

1.1. Background

Logistics and transportation systems are no longer merely about moving physical goods; they are about moving data. Modern industries face unprecedented challenges in managing and analyzing vast amounts of data generated by connected vehicles, IoT sensors, warehouse automation systems, and supply chain networks.

1.2. Problem Statement

Traditional on-premises data processing systems struggle with the scale, velocity, and variety of modern logistics data. Organizations require architectures capable of processing millions of events per second while providing real-time insights for operational decision-making and historical analysis for strategic planning [1] [2].

- Process high-velocity streaming data from fleet sensors and tracking devices
- Handle batch processing for historical analysis
- Scale elastically based on demand
- Maintain high availability and fault tolerance
- Integrate seamlessly with analytics and visualization tools

1.3. Objectives

This white paper outlines a modern cloud-native architecture and best practices for implementing high-throughput data processing systems on AWS designed for logistics and transportation use cases. The architectures address both real-time streaming requirements and batch processing needs while ensuring scalability, reliability, and cost optimization.

2. Architecture Overview

2.1. Modern Data Processing Architecture

A comprehensive data processing architecture for logistics systems consists of five key layers [6][9]

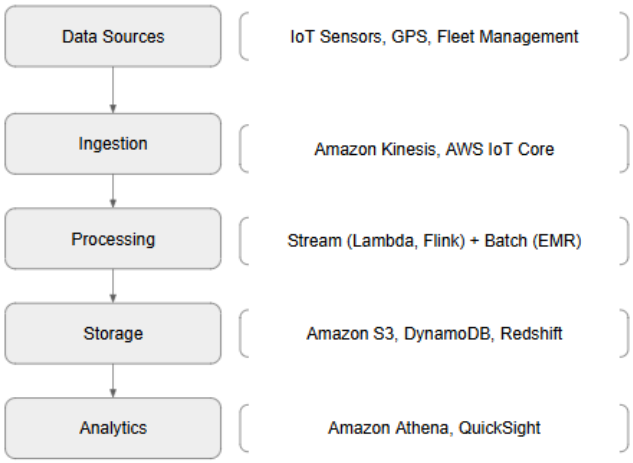

Figure 1. High-Level Architecture Components (source - author)

2.2. Architectural Pattern

The Architecture combines real-time stream processing (speed layer) with batch processing (batch layer):

- Speed Layer: Processes real-time data streams for immediate insights
- Batch Layer: Processes historical data for comprehensive analysis
- Serving Layer: Merges results from both layers for unified querying

3. Stream Data Processing Architecture

3.1. Real-Time Ingestion with Amazon Kinesis

Amazon Kinesis Data Streams serves as the backbone for real-time data ingestion, capable of handling thousands of transactions per second [2][6].

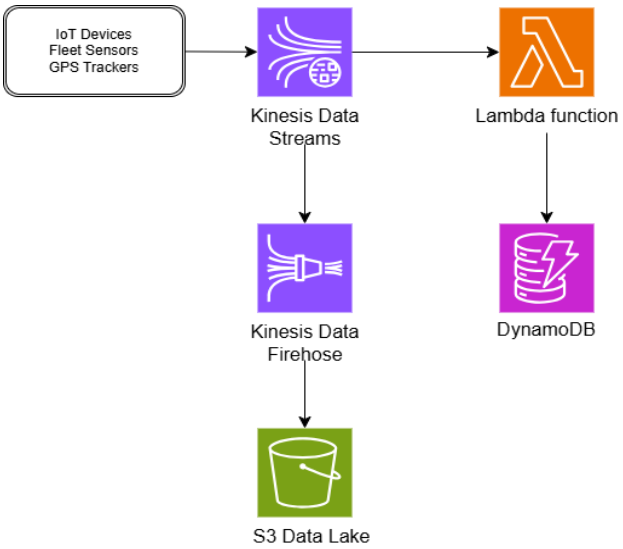

Figure 2. Kinesis Stream Processing Architecture (source - author)

3.1.1. Architecture Components

- Data Producers: IoT devices, fleet sensors, and GPS trackers send data using the Kinesis Producer Library (KPL)
- Kinesis Data Streams: Collects and processes real-time streaming data with configurable shard capacity
- Stream Processing: AWS Lambda functions process events with a parallelization factor defined by the number of concurrent batches per shard. However, if you have multiple Lambda functions with a microservices-based serverless architecture, you would need to enable Enhanced fan-out
- Data Persistence: DynamoDB for real-time operational data, S3 for long-term storage

3.2. Enhanced Fan-Out for High Throughput

For applications requiring multiple consumers, Kinesis enhanced fan-out provides dedicated 2 MB/second throughput per consumer, enabling support for multiple Lambda functions simultaneously [1].

In Enhanced fan-out, your Lambda doesn't "poll" Kinesis; instead, Kinesis pushes data to your Lambda over a dedicated pipe (HTTP/2), providing higher throughput and lower latency.

Pseudocode: Lambda Streaming Processing with Enhanced Fan-Out

This pseudocode assumes the Kinesis record contains a JSON payload from a fleet sensor (e.g., Truck ID, Latitude, Longitude, Fuel Level)

```
import base64
import json
import boto3
from aws_kinesis_agg.deaggregator import deaggregate_records # Required for KPL

# Initialize clients
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('FleetRealTimeStatus')

def lambda_handler(event, context):
    """
    Triggered via Kinesis EFO.
    Handles de-aggregation of KPL records for a logistics fleet.
    """

    # 1. De-aggregate KPL records into individual sensor readings
    raw_records = event['Records']
    user_records = deaggregate_records(raw_records)

    processed_count = 0

    for record in user_records:
        # 2. Extract and decode the payload
        payload_data = base64.b64decode(record['kinesis']['data'])
        sensor_data = json.loads(payload_data)

        # Example Payload: {'truck_id': 'T100', 'lat': 40.7, 'long': -74.0, 'status':
'moving'}
        truck_id = sensor_data.get('truck_id')

        # 3. Business Logic: Check for anomalies (e.g., rapid deceleration)
        if sensor_data.get('speed', 0) > 80:
            trigger_alert(truck_id, "Overspeeding")

        # 4. Sink to DynamoDB (The 'Hot' Path for real-time tracking)
        try:
            table.put_item(
                Item={
                    'PK': f"TRUCK#{truck_id}",
                    'SK': record['kinesis']['approximateArrivalTimestamp'],
                    'location': {
                        'lat': sensor_data['lat'],
                        'long': sensor_data['long']
                    },
                    'fuel_level': sensor_data['fuel']
                }
            )
            processed_count += 1
        except Exception as e:
            print(f"Error writing to Dynamo: {e}")

    return {
        'statusCode': 200,
        'body': f"Processed {processed_count} fleet sensor records."
    }

def trigger_alert(truck_id, reason):
    # Logic to push to SNS or an emergency dispatch dashboard
    print(f"ALERT: {truck_id} - {reason}")
```

3.3. Stream Processing Best Practices

Table 1. Kinesis Configuration Recommendations

Parameter	Recommended Value	Purpose
Shard Count	Based on throughput (1 MB/s write, 2 MB/s read per shard)	Capacity planning
Batch Size	100–1000 records	Optimize Lambda invocations
Batch Window	0–300 seconds	Balance latency vs efficiency
Parallelization Factor	1–10	Concurrent processing per shard
Retry Attempts	3–5	Fault tolerance
Data Retention	24–168 hours	Replay capability

4. Batch Data Processing Architecture

4.1. Amazon EMR for Large-Scale Processing

Amazon EMR provides a managed Hadoop and Spark environment for processing large-scale batch workloads

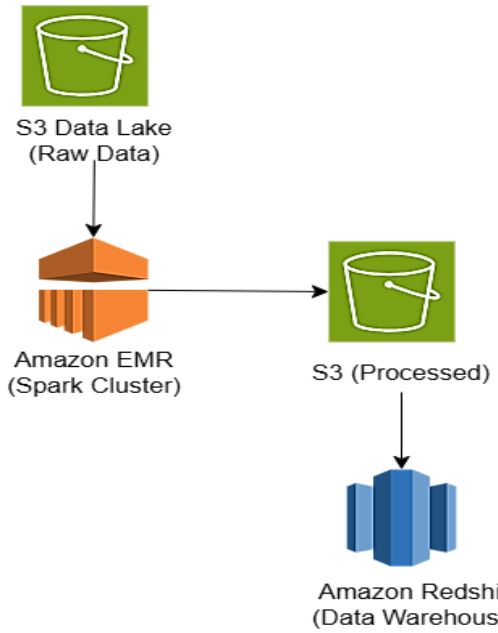

Figure 3. EMR Batch Processing Architecture (source - author)

4.2. Decoupled Storage and Compute

Modern best practices recommend separating storage (Amazon S3) from compute (EMR) for better scalability and cost efficiency [4] [9]

Benefits:

- Independent scaling of storage and compute resources
- Cost-effective storage with S3 lifecycle policies
- Ability to terminate clusters without data loss
- Multiple processing engines can access the same data

4.3. EMR Optimization Techniques

Pseudocode: Spark Batch processing for Fleet Analytics

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

# Initialize Spark with 2026 Native Redshift Integration & AQE
spark = SparkSession.builder \
    .appName("Fleet_Batch_Analytics") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .config("spark.dynamicAllocation.enabled", "true") \
    .config("spark.shuffle.service.enabled", "true") \
    .config("spark.sql.adaptive.skewJoin.enabled", "true") \
    .config("spark.sql.adaptive.localShuffleReader.enabled", "true") \
    .getOrCreate()

# Paths & Connectivity
# Note: jdbc:redshift:iam uses the EMR Role to authenticate to Redshift
S3_RAW_PATH = "s3://fleet-data-raw/logistics-iot-stream/"
S3_PROCESSED_PATH = "s3://fleet-data-processed/fleet-archive-parquet/"
REDSHIFT_TEMP_DIR = "s3://fleet-data-temp/redshift-staging/"
REDSHIFT_JDBC_URL = "jdbc:redshift:iam://fleet-cluster-2026.xyz.us-east-
1.redshift.amazonaws.com:5439/fleet_db"

def run_fleet_pipeline():
    # 1. READ: Native Spark JSON reader
    df_raw = spark.read.json(S3_RAW_PATH)

    # 2. TRANSFORM: Clean IoT data to match your Lambda schema
    df_refined = df_raw.select(
        F.col("truck_id").alias("vehicle_id"),
        F.to_timestamp("approximateArrivalTimestamp").alias("event_time"),
        F.col("fuel").cast("double"),
        F.col("speed").cast("double"),
        F.col("lat"),
        F.col("long"),
        F.col("status")
    ).filter("vehicle_id IS NOT NULL")

    # 3. ANALYTICS: Calculate Fleet KPIs
    # AQE handles partition management here to avoid 'small file' overhead
    df_daily_metrics = df_refined.groupBy("vehicle_id",
F.to_date("event_time").alias("log_date")) \
        .agg(
            F.avg("speed").alias("avg_speed_kmh"),
            F.max("speed").alias("max_speed_kmh"),
            F.avg("fuel").alias("avg_fuel_level"),
            F.count("event_time").alias("ping_count")
        )

    # 4. STORAGE: Traditional S3 Parquet Sink
    df_daily_metrics.write \
        .mode("append") \
        .partitionBy("log_date") \
        .parquet(S3_PROCESSED_PATH)

    # 5. SINK: Native Redshift Connector (No external JARs needed)
    # The 'redshift' format is now built into the EMR runtime.
    df_daily_metrics.write \
        .format("redshift") \
        .option("url", REDSHIFT_JDBC_URL) \
        .option("dbtable", "analytics.daily_fleet_summary") \
        .option("tempdir", REDSHIFT_TEMP_DIR) \
        .option("aws_iam_role",
"arn:aws:iam::123456789012:role/RedshiftS3AccessRole") \
        .mode("append") \
        .save()

if __name__ == "__main__":
    run_fleet_pipeline()
    spark.stop()
```

5. Scalability and Reliability

5.1. Horizontal Scaling Patterns

- The horizontal unit of scale in Kinesis is the ‘Shard’. Each shard provides a fixed capacity. To scale horizontally, you must increase or decrease the number of shards.
- For workloads with unpredictable spikes, you can utilize Kinesis Data Streams On-Demand, where

AWS automatically manages the shard count based on the observed throughput.

5.2. Fault Tolerance Mechanisms

Multi-AZ Deployment:

- Distribute resources across multiple Availability Zones
- Implement automatic failover for critical components

Checkpointing and Recovery:

- Implement checkpointing in stream processing applications
- Store state snapshots in Amazon S3 for recovery

Retry Strategies:

- Configure exponential backoff for transient failures
- Implement dead-letter queues for failed messages
- Set appropriate timeout values for Lambda functions

5.3. Scalability Metrics

Table 2. Performance Benchmarks for Logistics Workloads

Metric	Target Value	Service
Stream Ingestion Rate	50,000+ msg/sec	Kinesis Data Streams
Processing Latency	< 1 second	Lambda + Kinesis
Batch Processing Time	< 30 minutes	EMR Spark
Query Response Time	< 5 seconds	Athena
Data Availability	99.99%	S3, DynamoDB
Concurrent Consumers	8+ per stream	Kinesis Enhanced Fan-out

6. Conclusion

6.1. Summary

This white paper presented a comprehensive architecture for implementing high-throughput data processing systems on AWS tailored for logistics and transportation applications. The proposed solutions leverage AWS managed services to provide:

- Scalability: Horizontal scaling capabilities handling 50,000+ messages per second
- Reliability: Multi-AZ deployments with automatic failover and 99.99% availability
- Performance: Sub-second latency for real-time processing, efficient batch processing for historical analysis
- Cost Efficiency: Pay-per-use pricing models and optimized resource utilization
- Integration: Seamless connectivity with analytics and visualization platforms

6.2. Data Flow Description

- Data Collection: Fleet vehicles equipped with IoT sensors transmit telemetry data every 5-30 seconds
- Ingestion: The producer application sends data to Kinesis Data Streams using the KPL library to handle the heavy lifting
- Stream Processing: Lambda functions process real-time data
- Batch Processing: Historical data in S3 is processed by EMR for trend analysis and predictive maintenance
- Storage: Real-time operational data in DynamoDB, historical data in Redshift

6.3. Key Recommendations

Organizations implementing these architectures should:

- Start with stream processing for immediate operational insights, then add batch processing for historical analysis
- Implement proper monitoring using Amazon CloudWatch to track performance metrics and identify bottlenecks
- Design for failure with retry mechanisms, checkpointing, and multi-AZ deployments
- Optimize costs through right-sizing, auto-scaling, and appropriate data retention policies
- Iterate and improve based on actual workload patterns and business requirements

6.4. Future Considerations

As logistics and transportation systems continue to evolve, organizations should consider integrating machine learning, leveraging Amazon SageMaker for predictive maintenance and route optimization [8].

References

[1] "AWS Tools for Batch and Stream Data Transformation", AWS for Engineers Blog, 2025. [Online] Available: https://awsforengineers.com/blog/aws-tools-for-batch-and-stream-data-transformation/

[2] "Data Pipelines on AWS: 12 Services & How to Use Them Effectively", Dagster Guides, 2024. [Online]. Available: https://dagster.io/guides/data-pipelines-on-aws-12-services-how-to-use-them-effectively

[3] "Reference architecture", AWS Well-Architected Framework - Analytics Lens. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/reference-architecture-2.html

[4] "BUILD A REALTIME DATA PIPELINE SCALABLE APPLICATION DATA ANALYTICS ON AMAZON WEB SERVICES (AWS)", ResearchGate, 2024. [Online].Available:https://www.researchgate.net/publication/384200712

[5] "Building a 4x faster and more scalable algorithm using AWS Batch for Amazon Logistics", AWS HPC Blog, 2023. [Online]. Available: https://aws.amazon.com/blogs/hpc/building-a-4x-faster-and-scalable-algorithm-using-aws-batch-for-amazon-logistics/

[6] "Let's Architect! Designing systems for stream data processing", AWS Architecture Blog, 2023.

[Online].Available:https://aws.amazon.com/blogs/architecture/lets-architect-designing-systems-for-stream-data-processing/

[7] "Building a Modern Data Engineering Architecture on AWS with Databricks", Medium, 2025. [Online]. Available: https://medium.com/@hamidpmp/building-a-modern-data-engineering-architecture-on-aws-with-databricks-639781d8f4ab

[8] "How AWS Data Analytics Drive Operational Efficiency & Supply Chain Resilience", SourceFuse Blog, 2025. [Online]. Available: https://www.sourcefuse.com/resources/blog/powering-the-future-of-logistics

[9] "AWS Serverless data analytics pipeline reference architecture", AWS Big Data Blog, 2020. [Online]. Available:https://aws.amazon.com/blogs/big-data/aws-serverless-data-analytics-pipeline-reference-architecture/

[10] S. K. Sunkara, A. I. Ashirova, Y. Gulora, R. R. Baireddy, T. Tiwari and G. V. Sudha, "AI-Driven Big Data Analytics in Cloud Environments: Applications and Innovations," *2025 World Skills Conference on Universal Data Analytics and Sciences (WorldSUAS)*, Indore, India, 2025, pp. 1-6, doi: 10.1109/WorldSUAS66815.2025.11199123.