# Enabling enterprise Telemetry on AWS: Handling massive data backfills using ECS and Databricks without exploding the bill

17 June 2026 - 6 min. read

When designing a data ingestion architecture for massive data volumes, theory and practice often clash. On paper, cloud-native services offer ready-to-use solutions for any scenario; in the real world, you have to deal with throughput limits, unexpected exceptions, and, most importantly, the financial impact of your configured resources.

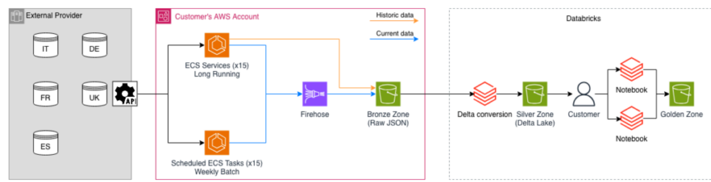
In this article, I want to share a deep dive behind-the-scenes of a data ingestion and transformation project we recently completed for one of our clients. We will focus on a specific architectural challenge: backfilling historical data and managing AWS Kinesis Firehose costs.

## The Context and Initial Architecture

The client needed to centralize telemetry data from their corporate vehicle fleet. This data was collected by an external provider and made available through a REST API. The goal was to build a pipeline capable of capturing this data in near real-time (effectively a CDC via API) and standardizing it for advanced analytics on Databricks.

The core ingestion strategy relied on a set of services designed to continuously poll the provider's endpoints. Once triggered, these components performed cyclic API calls to drain the available stream, downloading data continuously and pausing or entering a wait state only when there were no more records left to retrieve.

To provide a better context for the process, here is the architectural diagram illustrating the pipeline described in the following paragraphs.




For the compute infrastructure, we decided to leverage the then newly released **ECS Managed Instances**.

This Amazon ECS feature allows you to orchestrate and run Docker containers utilizing dedicated and optimized compute capacity, integrated directly inside an AWS-managed cluster. For us, it was the ideal solution: it enabled us to isolate our polling microservices securely in a dedicated environment, while maintaining all the flexibility, scalability, and ease of management typical of the AWS ecosystem.

If you want to dive deeper into how this approach works, check out the article by our dear colleague Damiano: When Serverless Runs on Servers: New Options for AWS Lambda and AWS Fargate with Managed Instances.

From day one, the architecture on ECS was structured to intelligently handle a mapping of data sources (divided by country and entity), separating the flows based on update frequency:

## 1. Real-time Services (CDC Long-Running)

For high-frequency data requiring a continuous flow, we configured **15 always-on ECS services**. This number resulted from combining 5 different databases divided by Country, each containing 3 high-update main entities. These containers handled constant Change Data Capture (CDC).

## 2. Scheduled Tasks (Weekly Batch)

At the same time, there were 3 other entities that updated much less frequently and contained smaller data volumes. Leaving services always running to poll these tables would have been a waste of resources. Therefore, we implemented 15 ECS tasks (5 Countries x 3 entities) akin to the previous ones, configured as scheduled tasks. These ran once a week to download the data and then shut down immediately.

The choice of Firehose was not accidental: for streaming current data, it is a formidable tool. As a fully managed (serverless) service, it autonomously handles scaling capacity based on incoming traffic, aggregating data in memory (buffering by

time or file size), and writing it to S3 already partitioned by date. This allowed us to avoid writing custom code for file management and wiped out infrastructure maintenance costs.

As long as the pipeline processed only current data, the system proved to be extremely stable and efficient.

## Historical Data Load and the Firehose Bottleneck

The issues emerged when, once the pipeline reached a mature state, we initiated the historical data backfill. We were dealing with a massive volume of data, totaling tens of billions of records and taking up more than one hundred Terabytes of overall storage.

The moment we started pulling the historic data, the Firehose-based architecture hit its structural and financial limits under heavy load:

- **PartitionCountExceeded** Errors: The request volume and partitioning granularity saturated Firehose's quotas, throwing exceptions and slowing down the ingestion process.
- **Cost Explosion:** Firehose charges based on the gigabytes of data processed. Applying this metric to tens of Terabytes in a short timeframe caused an unsustainable and unjustified spending spike for "cold" data.

## The Solution: A Strategic Bypass for Bulk Data

Faced with this scenario, we realized that a pipeline optimized for real-time streaming was fundamentally a poor fit for bulk batch loading. We therefore split our ingestion strategy:

- **Current Data:** Remained on the original pipeline (ECS -> Firehose -> S3), where lower volumes make Firehose cost-effective and highly scalable.
- **Historical Data:** We modified the microservices' logic on ECS to completely bypass Firehose. The containers downloaded the historical JSONs from the APIs and wrote them directly to the S3 bucket using the AWS SDK.

To put this new strategy into practice without overloading the system, we rethought how the containers were invoked: instead of configuring them as continuous services, we ran them as **one-shot ECS tasks**. By passing a start date and an end date as **environment variables** to each task, we managed to segment the history and

parallelize ingestion across multiple containers simultaneously. This way, the tasks downloaded the historical JSONs from the APIs for their specific assigned time slot, wrote them directly to the S3 bucket via the AWS SDK, and shut down immediately after.

This approach required a small architectural compromise: **we had to sacrifice part of Firehose's native dynamic partitioning**. When Firehose writes to S3 using dynamic partitioning, it removes the partition fields from the JSON payload to use them exclusively as folder names. Writing directly from ECS, however, meant our JSONs kept all columns inside the file.

To put this into perspective, a typical storage path generated by Firehose looked like this:

**s3://bucket-name/raw/database=fleetdb_de/entity=FaultData/year=2025/month=01/day=01/file.json**

To prevent Databricks from seeing these columns as duplicates (once from the S3 folder path and once from the file content), we decided not to replicate Firehose's ultra-granular structure. Instead, we saved the historical data directly at the root level of the main year= partition. This avoided the need for a preventive column "drop" within the ECS code, leaving the management of the structure directly to the next phase on Databricks.

This architectural tweak instantly resolved the partitioning errors, eliminated Firehose transit costs for the historical data, and significantly sped up the completion of the backfill.

## From JSON to Delta Lake

Once the raw JSON files were safely stored in the S3 Bronze Zone, the workflow moved to Databricks for the ETL and data structuring phase.

We implemented Databricks jobs to standardize the files: the raw JSONs were read, schema-enforced, converted into **Delta Lake** format, and then stored in the Silver Zone. Adopting the Delta format provided the client with ACID transactions, optimized query performance, and a reliable single source of truth.

From that point on, our infrastructure pipeline made way for business logic. The client was able to independently develop their own analytics and reporting jobs on Databricks, finally leveraging a clean, structured, and highly accessible data asset.

## Conclusion

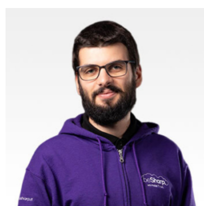
This project reinforced that there is no one-size-fits-all solution in Cloud Data Engineering. Kinesis Firehose remains an excellent tool for streaming live data. However, when tackling historical migrations in the Terabyte range, software engineering requires flexibility: sometimes, removing a managed intermediary is the key to saving both performance and budget.

---

## About Proud2beCloud

**Proud2beCloud** is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

---



**Keidi Xhafa**

DevOps Engineer @ beSharp. Driven since childhood by a deep curiosity for technology and computers, combined with a natural obsession for making things work smoothly, which is pretty much how I ended up here today. When I'm not building backends and ETL jobs, you can find me enjoying good food, watching movies, playing video games, or casting spells in a Magic: The Gathering match.

---

Copyright © 2011-2026 by beSharp spa - P.IVA IT02415160189