Telemetria Enterprise su AWS: Gestire backfill di dati massivi con ECS e Databricks senza far esplodere i costi

17 Giugno 2026 - 7 min. read

Quando si progetta un'architettura di data ingestion per grandi volumi di dati, la teoria e la pratica spesso si scontrano. Sulla carta, i servizi cloud nativi offrono soluzioni pronte all'uso per qualsiasi scenario; nel mondo reale, ci si trova invece a fare i conti con limiti di throughput, eccezioni impreviste e, soprattutto, con l'impatto economico delle risorse configurate.

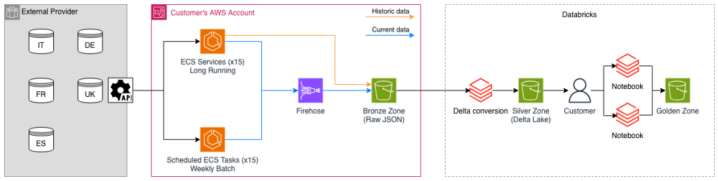
In questo articolo voglio raccontarvi il dietro le quinte di un progetto di ingestion e trasformazione dei dati che abbiamo completato di recente per un nostro cliente, concentrandoci su una sfida architetturale specifica: il recupero dello storico dei dati e la gestione dei costi di AWS Kinesis Firehose.

Il contesto e l'architettura iniziale

Il cliente aveva l'esigenza di centralizzare i dati di telemetria della propria flotta di veicoli aziendali. Questi dati venivano raccolti da un provider esterno e messi a disposizione tramite REST API. L'obiettivo era creare una pipeline capace di catturare questi dati in modalità quasi real-time (una sorta di CDC via API) e standardizzarli per l'analisi avanzata su Databricks.

La strategia adottata per lo scarico consisteva nell'implementare una serie di servizi logici configurati per effettuare polling continuo sugli endpoint del provider: una volta avviati, questi componenti effettuavano chiamate cicliche per scaricare i dati a flusso continuo, mettendosi in attesa solo quando non risultavano nuovi record da recuperare.

Per contestualizzare meglio il processo, ecco lo schema dell'architettura che illustra la pipeline descritta nei paragrafi seguenti.



Per l'infrastruttura di calcolo abbiamo deciso di sfruttare le **ECS Managed Instances**, allora appena uscite.

Questa funzionalità di Amazon ECS permette di orchestrare ed eseguire container Docker sfruttando una capacità di calcolo dedicata e ottimizzata, integrata direttamente all'interno di un cluster gestito da AWS. Per noi è stata la soluzione ideale: ci ha permesso di isolare i microservizi di polling in un ambiente sicuro e circoscritto, mantenendo tutta la flessibilità, la scalabilità e la semplicità di gestione tipiche dell'ecosistema AWS.

Se volete approfondire come funziona questo approccio, date un'occhiata all'articolo del nostro caro collega Damiano: Quando il Serverless gira sui Server: nuove opzioni per AWS Lambda e AWS Fargate con le Managed Instances.

Fin dal giorno uno, l'architettura su ECS è stata strutturata per gestire in modo intelligente un mapping di sorgenti dati (divise per country ed entità), separando i flussi in base alla frequenza di aggiornamento:

1. I Servizi Real-time (CDC Long-Running)

Per i dati ad alta frequenza che necessitano di un flusso continuo, abbiamo configurato **15 servizi ECS sempre attivi**. Questa numerica nasceva dalla combinazione di 5 database diversi divisi per Country (nazioni), ognuno dei quali conteneva 3 entità principali ad alto aggiornamento. Questi container gestivano la Change Data Capture (CDC) costante.

2. I Task Schedulati (Batch settimanali)

Allo stesso tempo, c'erano altre 3 entità che si aggiornavano molto meno frequentemente e che contenevano volumi di dati ridotti. Lasciare dei servizi sempre accesi a fare polling su queste tabelle sarebbe stato uno spreco di risorse. Abbiamo quindi implementato altri 15 task ECS speculari (5 Country x 3 entità), ma configurati

come task schedulati, eseguiti una volta alla settimana per scaricare i dati e poi spegnersi subito dopo.

La scelta di Firehose non è stata casuale: per lo streaming di dati correnti è uno strumento formidabile. Essendo un servizio completamente gestito (*serverless*), si occupa in totale autonomia di scalare la capacità in base al traffico in ingresso, aggregare i dati in memoria (buffering per tempo o per dimensione del file) e scriverli su S3 già partizionati per data. Questo ci ha permesso di evitare la scrittura di codice custom per la gestione dei file e di azzerare i costi di manutenzione infrastrutturale.

Finché la pipeline ha elaborato i soli dati correnti, il sistema si è dimostrato estremamente stabile ed efficiente.

Il caricamento dello storico e il collo di bottiglia di Firehose

I problemi sono emersi quando, raggiunto lo stato di maturità della pipeline, è stato avviato il recupero dello **storico dei dati**. Parliamo di una mole di dati imponente, quantificabile in decine di miliardi di record per una dimensione complessiva di più di cento Terabyte di storage occupato.

Nel momento in cui abbiamo aperto i rubinetti per importare i dati pregressi, l'architettura basata su Firehose ha mostrato i suoi limiti strutturali ed economici sotto carichi massivi:

1. **Errori di PartitionCountExceeded**: Il volume di richieste e la granularità del partizionamento hanno saturato le quote di Firehose, generando eccezioni e rallentando il processo di ingestion.
2. **Esplosione dei costi**: Firehose addebita i costi in base ai GB di dati elaborati. Applicare questa metrica a decine di Terabyte in un lasso di tempo ristretto ha causato un picco di spesa insostenibile e ingiustificato per dei dati "freddi".

La soluzione: Il bypass per i dati massivi

Davanti a questo scenario, abbiamo capito che una pipeline ottimizzata per lo streaming real-time non poteva essere adatta a un caricamento massivo di tipo batch. Abbiamo quindi diviso la strategia di ingestion:

- **Dati correnti**: Sono rimasti sulla pipeline originale (ECS -> Firehose -> S3), dove i volumi ridotti rendono Firehose economico e scalabile.

- **Dati storici**: Abbiamo modificato la logica dei microservizi su ECS per effettuare un bypass totale di Firehose. I container scaricavano i JSON storici dalle API e li scrivevano direttamente sul bucket S3 tramite SDK AWS.

Per mettere in pratica questa nuova strategia senza sovraccaricare il sistema, abbiamo ripensato il modo di invocare i container: anziché configurarli come servizi continui, li abbiamo eseguiti come **task ECS *one-shot***. Passando una data di inizio e una data di fine come **variabili d'ambiente** a ciascun task, siamo riusciti a segmentare lo storico e a **parallelizzare l'ingestion** su più container contemporaneamente. In questo modo, i task scaricavano i JSON storici dalle API per lo specifico slot temporale assegnato e li scrivevano direttamente sul bucket S3 tramite SDK AWS, spegnendosi subito dopo.

Questo approccio ci ha richiesto un piccolo compromesso architetturale: **abbiamo dovuto sacrificare parte del *dynamic partitioning*** nativo di Firehose. Quando Firehose scrive su S3 partizionando i dati, rimuove i campi di partizione dal payload JSON per usarli solo come nomi delle cartelle. Scrivendo direttamente da ECS, invece, i nostri JSON mantenevano tutte le colonne all'interno del file.

Per capire il problema, basti pensare che un tipico path strutturato da Firehose si presentava così:

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

Per evitare che Databricks vedesse queste colonne come duplicate (una volta dal percorso della cartella di S3 e una volta dal contenuto del file), abbiamo deciso di non replicare la struttura ultra-granularizzata di Firehose, ma di salvare i dati storici direttamente al livello della partizione principale **year=**. In questo modo abbiamo evitato la necessità di dover fare un "drop" preventivo delle colonne da codice ECS, delegando la normalizzazione della struttura direttamente alla fase successiva.

Questa modifica architetturale ha risolto istantaneamente gli errori di partizionamento, azzerato i costi di transito di Firehose per lo storico e velocizzato sensibilmente il completamento del caricamento.

Da JSON a Delta Lake

Una volta archiviati i file JSON grezzi nella *Bronze Zone* su S3, il flusso si è spostato su **Databricks** per l'esecuzione delle pipeline di ETL e la strutturazione del dato.

Abbiamo implementato dei job Databricks che si occupano di standardizzare i file: i JSON grezzi vengono letti, tipizzati e convertiti nel formato **Delta Lake**, per poi essere memorizzati nella *Silver Zone*. L'adozione del formato Delta ha garantito al cliente transazioni ACID, performance di query ottimizzate e una "single source of truth" affidabile.

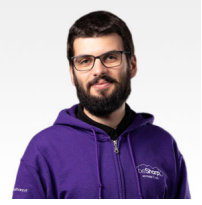
Da quel punto in poi, la nostra pipeline infrastrutturale ha ceduto il passo alle logiche di business: il cliente ha potuto iniziare a sviluppare in totale autonomia i propri job di analisi e reportistica su Databricks, sfruttando un patrimonio informativo finalmente pulito e accessibile.

Conclusione

Questo progetto ci ha confermato che nel Cloud Data Engineering non esiste una soluzione universale. Kinesis Firehose resta uno strumento eccellente per lo streaming di dati correnti, ma quando si affrontano migrazioni storiche nell'ordine dei Terabyte, l'ingegneria del software richiede flessibilità: a volte, eliminare un intermediario gestito è la chiave per salvare performance e budget.

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!



Keidi Xhafa

DevOps Engineer @ beSharp. Fin da piccolo la curiosità per la tecnologia e i computer, unita alla naturale ossessione per far funzionare le cose (e farle funzionare bene), mi hanno guidato fino a dove sono oggi. Quando non sono impegnato a mettere in piedi backend e job ETL, mi potete trovare a godermi del buon cibo, guardare film, videogiocare o lanciare magie in una partita a Magic: The Gathering.
