Building a Production-Ready Event Analytics Pipeline on AWS: A Data Engineer's Walkthrough

- *By Sanya Katiyar*

High volume event streams keep moving whether you are ready or not. They land in storage every few minutes and come in without structure. That is the kind of environment I recreated while building this pipeline: a continuous stream of e-commerce user events, each capturing a small detail about customer behavior. My job was to step into the role of a data engineer and turn that raw, noisy flow into something analysts could actually use.

Even though this scenario was hypothetical, the workflow felt very close to my experience building large-scale data engineering solutions in industry, most notably at my previous role at ZS Associates where stability, reproducibility, and thoughtful design were central to every project. Designing this pipeline brought many of those instincts back to the surface after seeing jobs fail in unexpected ways and debugging schemas that drifted quietly over time. This project became an exercise in building something functional and a space to reflect on how and why I make certain architectural choices.

This post walks through how I approached the problem, the reasoning behind the system I designed, and the steps I took to validate that it worked correctly.

- Understanding the Challenge: A Firehose of E-Commerce Events

The system simulated an e-commerce platform that logs every meaningful interaction a customer has with the site. An AWS Lambda function wrote roughly 500,000 to 750,000 events every five minutes into S3. The data was both high-volume and continuous, which created the need for an incremental processing strategy. The schema included common behavioral fields: timestamps in ISO-8601 format, user and session identifiers, event types, and event-specific attributes such as product ID, quantity, price, category, and search query.

Each Lambda invocation dropped a gzipped JSONL file into S3 using hive-style partitioning by year, month, day, hour, and minute. That partitioning structure made it possible to discover partitions automatically using Athena and Glue, but poorly chosen downstream partitioning decisions could increase the number of files and hurt performance. Working with raw JSON is flexible, but it does not support fast analytics. I needed an architecture that could transform the data, maintain consistency, and support large-scale queries.

- Why a Layered Architecture Made Sense

One of the clearest patterns I saw during my past work was that layered architectures create stability. For this project, a Bronze-Silver-Gold structure felt like a natural fit.

1. Bronze: The Raw Events Layer

The bronze layer served as the untouched landing zone. It stored exactly what the Lambda produced and maintained the minute-level S3 partitions. This layer acted as both the system of record anda reliable fallback. If anything downstream needed to be recomputed, the raw data was always available.

When something unexpected shows up in a report, the first question is often, "Is the input wrong, or did the transformation do something unexpected?" Having a clean raw layer makes that question much easier to answer.

2. Silver: The Clean, Structured, Analytics-Optimized Layer

The silver layer was where the ETL transformations happened. Using a Glue Spark job, I read the raw JSON, cleaned it, normalized the structure, and wrote Parquet files partitioned by year, month, day, and hour.

Choosing hourly partitions was intentional. Minute-level partitions would produce hundreds of small files in a short period, which would slow Athena queries. I have seen how too many small files can turn into a performance problem, and this becomes more visible when analysts build dashboards that trigger many queries in parallel. Hour-level partitioning provided the right granularity while keeping file sizes efficient.

I also added derived fields such as standardized timestamps, numeric casts for price and quantity, and a revenue column for purchase events. These transformations ensured consistent semantics and removed irregularities from the raw data.

3. Gold: The Analytics Layer

Instead of building a complex gold layer with many tables, I used Athena to expose the transformed Parquet dataset as a single query-ready analytics table. Traditionally, gold layers contain multiple aggregated analytical tables serving specific business use cases, for example, daily sales summaries, customer lifetime value tables, or product performance dashboards. From my experience, these multi-table gold architectures make sense when you have multiple silver tables that need to be joined and aggregated into use-case specific views for different teams.

Since in this case, the pipeline worked with a single event stream, creating additional aggregated tables would have added unnecessary complexity. For this use case, keeping the analytics surface lean made more sense.

- Incremental Processing: Glue Bookmarks and Idempotency

The dataset grew every five minutes, so rerunning the entire ETL on all historical files would be inefficient and expensive. The pipeline needed a way to detect and process only new files, which is where Glue bookmarks became useful.

Bookmarks track which files have already been processed. When the ETL job runs again, it reads only new partitions and avoids rewriting existing data. This makes the job idempotent, stable, and safe to rerun. In my experience, incremental logic acts as a quiet backbone for reliable ETL systems. Jobs fail, clusters restart, unexpected spikes occur, and backfills are sometimes required. Bookmarks help absorb much of that unpredictability.

I validated this by running the Glue job once, checking the record counts in Athena, then running it again after more raw files arrived. The second run only picked up the new records, which confirmed that the bookmarks were working correctly.

- Transforming the Data: Designing the Glue ETL Script

Writing the Glue ETL script became one of the most interesting parts of the project. I wanted the script to handle the transformations and set up the entire analytic environment, a choice shaped by my work at ZS where reproducibility mattered.

The ETL script handled several responsibilities end to end: loading raw JSON using DynamicFrames, converting to Spark DataFrames for transformations, enforcing type correctness, cleaning malformed records, parsing timestamps, deriving time-based fields, normalizing strings, computing revenue for purchase events, writing partitioned Parquet files, registering Athena databases and tables through SQL, and running 'MSCK REPAIR TABLE' so Athena could discover partitions.

This approach removed the need for manual schema creation in the console. Tables were created automatically through Athena queries invoked through boto3, which allowed the script to stand alone. I also included validation checks such as printing schemas and sample rows to catch schema mismatches or unexpected nulls before they turn into confusing query results.

- Infrastructure-as-Code: Extending CloudFormation

After completing the ETL script, the next step was to enable full environment deployment through CloudFormation. This included provisioning the source events S3 bucket, the analytics S3 bucket, Glue databases for both staging and analytics, a Glue job definition pointing to the ETL script, IAM roles with right permissions, and parameters to avoid hardcoding names.

By defining everything in CloudFormation, the pipeline became reproducible and portable. Infrastructure-as-code helps remove environment inconsistencies and keeps deployments aligned.

- Architecture Overview: How the Pieces Fit Together

The architecture follows a clear flow:

1. Event generation and ingestion

EventBridge triggers a Lambda function every five minutes, and each run produces around 500k-750k events. Lambda writes these events as gzipped JSONL files into the Bronze S3 bucket, already organized by year, month, day, hour, and minute. This gives a clean structure right from the start.

2. Raw storage and partition organization

The Bronze S3 bucket stores the raw input exactly as it arrives. Keeping the untouched data makes it easy to check whether any issue comes from the source or from later transformations. The minute-level partitions also make it simple to isolate or reprocess specific time windows if something needs review.

3. Incremental transformation using Glue ETL

The Glue ETL job reads only the new files using bookmarks keeping processing fast and avoiding unnecessary cost. It then performs the required transformations on the raw dataset. The output is written as Parquet with hour-level partitions to avoid creating too many small files.

4. Metadata and table management

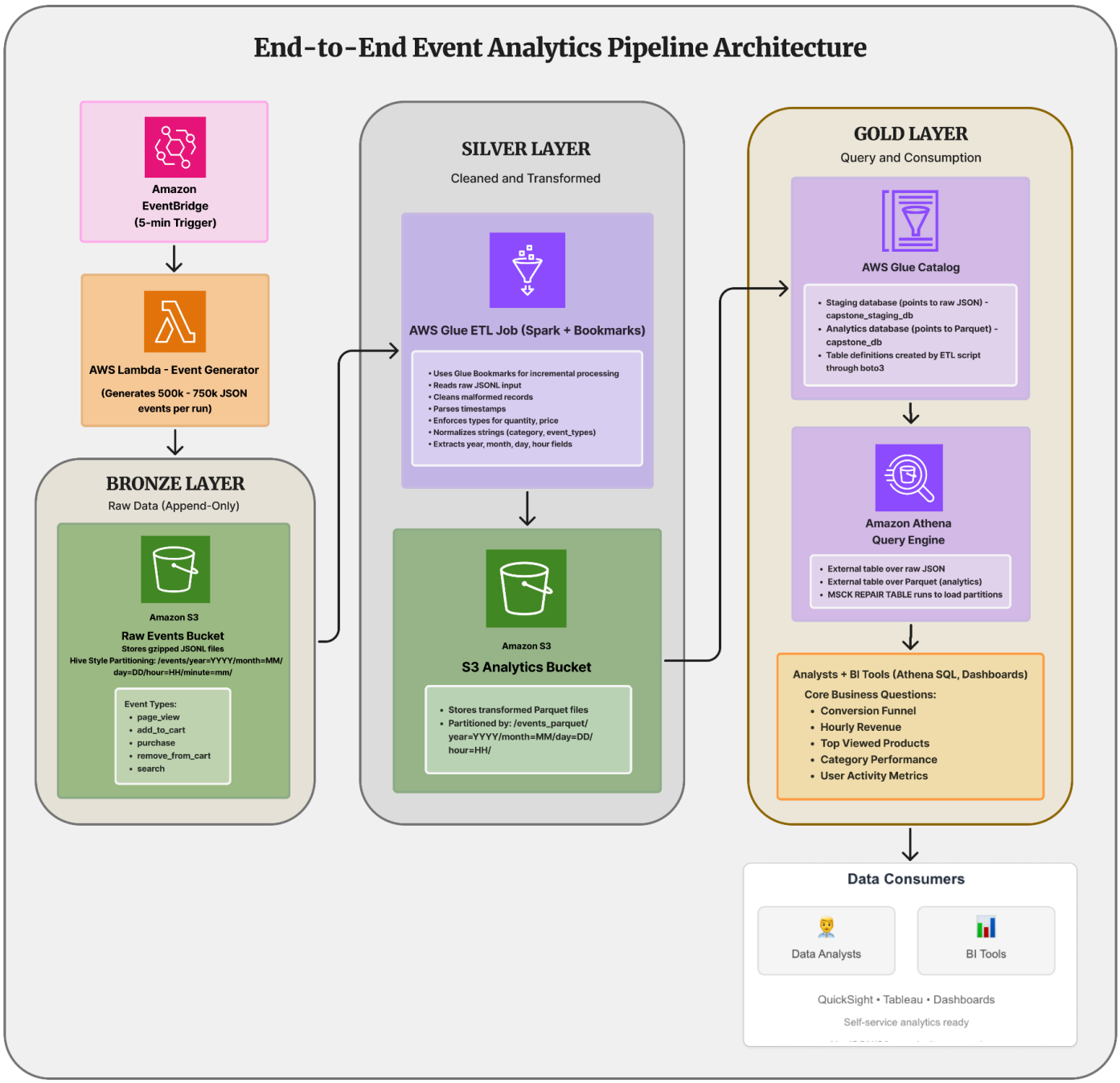
The ETL script also sets up the Athena databases and tables automatically using boto3. This removes the need for any manual table creation. Running MSCK REPAIR TABLE ensures that Athena discovers all new partitions and makes the cleaned data quickly available for querying.

5. Analytics storage layer (Parquet in S3)

The analytics S3 bucket stores the transformed dataset in Parquet format. Parquet reduces scan costs and speeds up queries since Athena only reads the necessary columns.

6. Query interface through Athena

Athena exposes the Parquet dataset as a single external table that analysts can query with SQL. Because the data is clean and partitioned well, analytical queries run quickly and cost less.



The pipeline works well because each service does its part. Lambda produces the data, S3 holds it, Glue cleans it, and Athena lets analysts query it. Good partitioning helps everything stay fast and efficient.

- Making Decisions: Trade-Offs and Alternatives

No data pipeline design is free of trade-offs. Here are the key decisions and their rationale:

- Explicit schemas over Glue crawlers - Crawlers can be convenient, but they sometimes change the schema when the raw data shifts, which can create confusion later. Defining the schema myself in Athena felt safer and gave me full control.
- Parquet over JSON - JSON is flexible, but it is expensive and slow for Athena to scan at scale. Parquet's compression and column pruning made queries faster and much more cost-efficient.
- Hourly partitioning over minute-level - Minute partitions would create too many small files and slow queries. Hourly partitions kept the dataset organized without overwhelming Athena.
- Single analytics table over multiple gold tables - Traditional gold layers contain aggregated analytical tables for specific business use cases. Since this pipeline worked with one event stream, additional aggregated tables would have added unnecessary complexity.

These choices came from past experience and from seeing what can go wrong in real pipelines. Sometimes the simplest design works best. Clean schemas, fewer partitions, and a straightforward data model make the system easier to maintain and easier for analysts to use.

- Debugging Failures and Key Takeaways

Building this pipeline came with a few debugging moments worth noting. One issue came from the Athena integration inside the Glue ETL script, where the job failed with an *InvalidRequestException* because I forgot to pass a database name into the query context. Adding the database parameter consistently fixed it.

Another challenge was aligning S3 paths with the table definitions. The staging table used year, month, day, hour, and minute partitions, while the analytics table used only year, month, day, and hour. When these didn't match, "MSCK REPAIR TABLE" couldn't discover partitions correctly.

Glue bookmarks also required care since they only pick up "new" files. I kept this stable by printing record counts in each run and checking them against expected file numbers. IAM permissions needed attention as well, so defining role ARNs clearly in CloudFormation helped avoid access issues.

These debugging steps shaped how carefully the pieces were connected and validated.

- Validating the Pipeline Through Business Queries

I checked that the pipeline worked as expected by running five basic queries that matched common business questions.

1. Conversion Funnel Analysis counted page views, add-to-cart events, and purchases for each product to confirm that event types were labeled correctly and that product IDs were filled in properly.

2. Hourly Revenue calculated revenue by using price and quantity for purchase events to verify that timestamps were parsed correctly and that time-based partitioning made the query fast.
3. Top Viewed Products ranked products by page views to confirm that product details were stored consistently and that the number of view events looked reasonable.
4. Category Performance counted events by category and event type for each day to check that category names and date fields were cleaned and standardized the right way.
5. User Activity counted unique users and sessions per day to make sure that user IDs and session IDs were tracked correctly across events.

All five queries returned accurate results and ran smoothly on the Parquet dataset. That gave me confidence that the pipeline was producing clean, reliable data that analysts could use without extra fixes.

- Reflections: What This Project Reinforced

Building this pipeline taught me a few lessons that stood out clearly. I learned that partitioning matters because it directly affects how fast analysts can run time based queries. I also saw how keeping raw and cleaned layers separate makes it much easier to trace where a problem started. Using explicit schemas helped avoid unexpected issues later when more complex queries were needed. Having a clean analytics layer makes the data easier for analysts to explore and for tools like Tableau to use without extra work. Reproducibility also became important because it makes the pipeline easier to fix and rebuild when needed.

I also noticed how much my past experience guided my decisions. I avoided too many partitions because I have seen slow Athena queries caused by tiny files. I skipped crawlers because I have dealt with schema drift before. I wrote a self contained ETL script because I have worked in environments where being able to rebuild the system from scratch was essential. Even though this project was simulated, the choices felt grounded in real work and the same ideas that help larger systems stay healthy as they grow.

- Conclusion

This pipeline took a fast stream of semi-structured events and turned it into a clean and reliable analytics setup. With careful choices around partitioning, layered storage, incremental processing, and infrastructure as code, the system became both stable and easy to reuse.

It is the kind of pipeline I would feel comfortable running in a real production setting because it is easy to understand, handles changes well, manages cost, and can be explained to both technical and non-technical stakeholders. For me, that is what good data engineering looks like: building systems that are practical, maintainable, and trustworthy without adding unnecessary complexity.