Building a Cloud-Based E-Commerce Analytics Pipeline

Izzy Valdivia

Overview

As part of the Autumn 2025 Scalable Data Systems and Algorithms course, I built a production-grade analytics pipeline that processes large volumes of artificial e-commerce event data and transforms it for business analytics. I was given a starter CloudFormation template that generates a stack to set up the data generation pipeline. This data generation pipeline simulates a situation I might find when out in the proverbial professional ‘wild’ mimicking a high volume raw data that needs a scalable solution.

To tackle the issue, I extended the starter template to include resources to ingest, process, restructure and query the data. This involved an AWS glue job, glue crawler, two event triggers, an s3 bucket, and an athena work group. In the following blog post, I will discuss the pipeline in more detail as well as walk through my decision making process with respect to each of the resources I implemented.

I implemented the pipeline using AWS cloud services, the platform provided to me. However, part of our course involved discussing the generalizability of cloud computing techniques and the overlap with other product suites like Google Cloud and Azure.

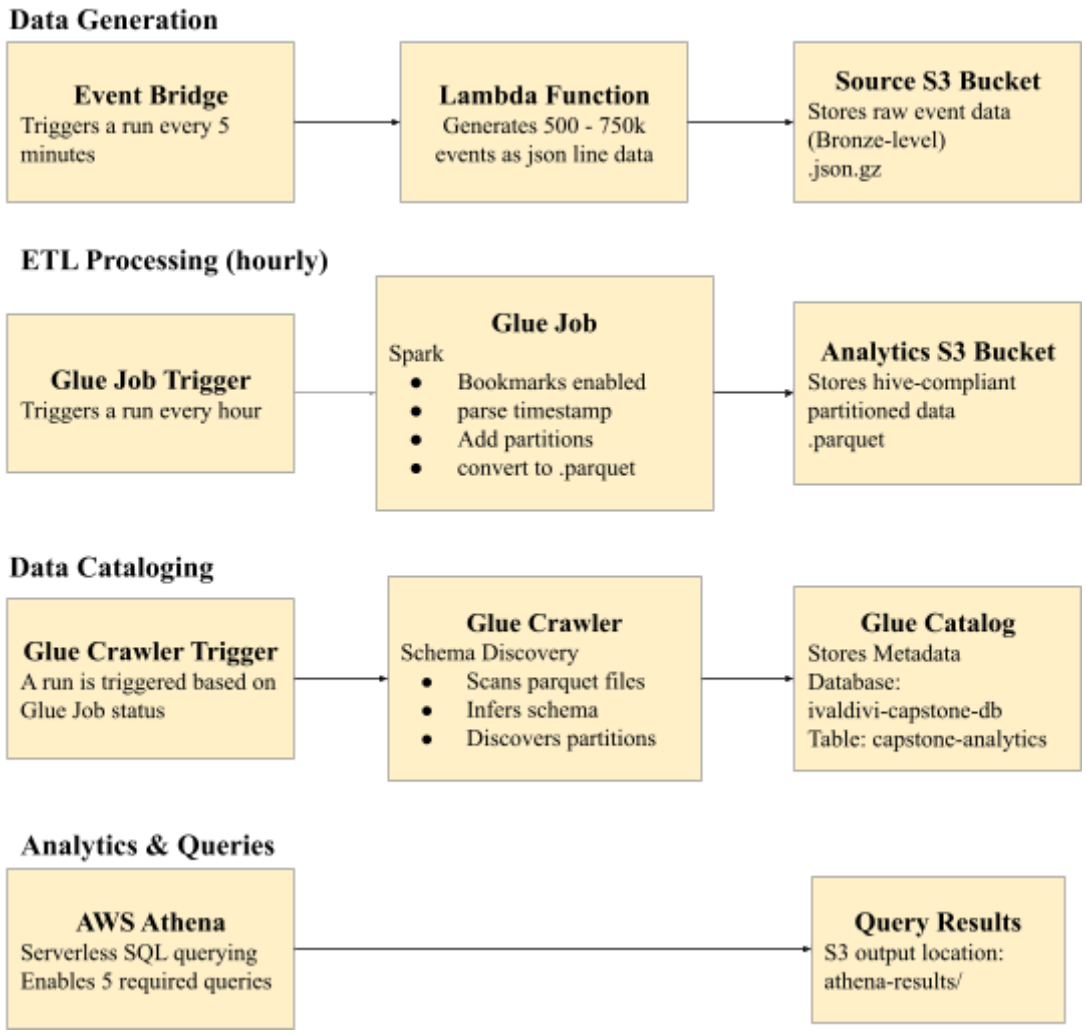
Business Context

The event data created by the template generates raw event logs, which are not queryable in their original form of gzipped JSON line files in S3. The data is structured, but not in a form that an analyst of BI engineer can query efficiently which is why it needs to be processed before it can be used. One file is generated every 5 minutes and each file contains between 500,000 and 750,000 events, which means the pipeline needs to handle processing the new data incrementally without reprocessing all previous data. The files are stored in Hive-compliant partitioning using the following partitions: year, month, day, hour, minute (eg. year=YYYY/month=MM/day=DD/hour=HH/minute=mm/). Each event has a variety of relevant fields like timestamp, session_id, product_id, category and price.

The data needed to be in a new format that was easily queryable to help the business answer questions about event data on a variety of topics like revenue and user activity. The queries include:

1. A conversion funnel: for each product, what is the conversion rate of views to purchases
2. Hourly Revenue
3. Top 10 Products by view count
4. Category Performance: daily event counts grouped by category
5. User Activity: number of unique users and sessions per day

Architecture Overview



Each row in the diagram connects to the one below it. So, Data Generation feeds into ETL Processing, and so on. The data generation part of the project is the automatic event generation that I have previously explained. One thing to note is that the .jsonl.gzip files are stored in S3.

I designed a Glue job that has bookmarking enabled to allow incremental data processing. The corresponding python ETL script parses each event's timestamp, adds partitions and converts the data into a compressed columnar format (parquet) which is then stored in an S3 bucket that I created. I chose to add year, month, day and hourly partitions. I will explain why I chose these partitions in the next section. The data output by the glue job is then scanned by a glue crawler to infer the schema and update the partitions if necessary. This information is then passed to a corresponding glue database called ivaldivi-analytics-db that includes a table called capstone-analytics. Additionally, there is an athena workgroup that allows the database and table to be queryable. I added a trigger to run the glue job at the top of every hour. Similarly, there is a trigger that sets off the glue crawler once the job has completed successfully.

Design Decisions

There were a host of factors that I needed to weigh when making decisions about the architecture of the pipeline. The following sections will walk you through the major decisions I made, explaining the options, reasoning and tradeoffs of each.

Decision 1: Partitioning Strategy

The partitioning strategy of the data that would be stored in the S3 output bucket was critical. If chosen correctly, they would help efficiently store and retrieve the data through optimized queries and high scalability. In a real-world setting data querying needs to be highly performant so considering multiple partitioning strategies is necessary. The optimal partitioning strategy depends heavily on the fields that filter and group the data in the queries as well as the frequency/load of the queries used to retrieve the data. So the first thing I considered when weighing the partition options was the 5 queries we were told the business wants to run. We were not given any explicit query load requirements, although some could be inferred. For example, it is unlikely that the daily user activity query would be run multiple times per hour because only the current day's data would change. No single field spanned all 5 of the queries, although 3 used some or all of the timestamp fields (year, month, day, hour) and 2 referenced product_id.

With these parameters in mind, I weighed two different partitioning strategies. The first was partitioning by product_id, year, month, day, and hour and the second was partitioning by

year, month, day and hour. Although the addition of product_id may have made a couple of the queries more efficient, it had the potential to slow all of the queries down if there are a large number of products that result in a large number of sub-folders that need to be scanned. Because of this, I decided to partition only by year, month, day and hour.

Decision 2: File Format

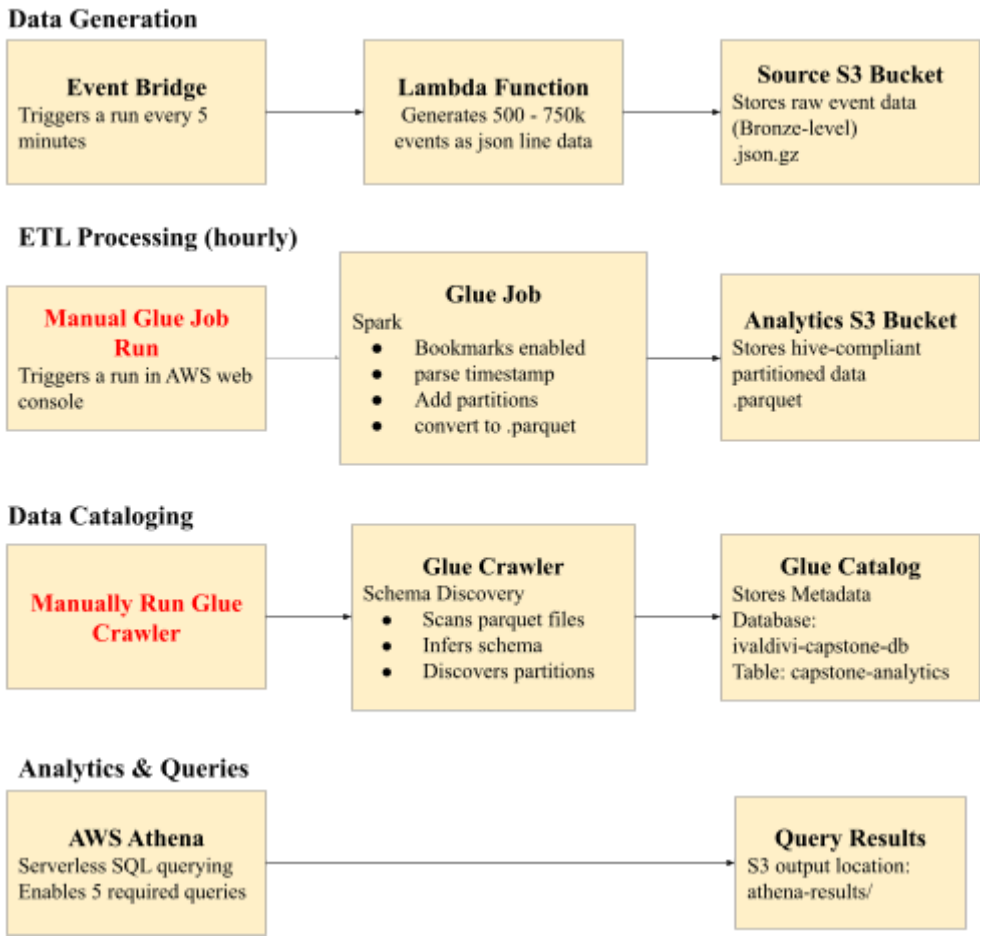
The next decision that I had to make was how to store the files in the analytics S3 bucket, post processing by the glue job. Depending on the amount of data you need to process, this decision can greatly affect query performance. I ended up deciding between storing the data in .csv files or in .parquet files. I wanted to consider both row-based and columnar-based file formats. While the row-based options are great for quick writes to the database, the columnar file types generally have faster reads because they only scan referenced columns. Additionally, the columnar files are able to be stored in a compressed format. These two advantages led me to choose to store the data in .parquet files.

Decision 3: Incremental Data Processing

One explicit requirement of the project was that the pipeline had to incrementally process additional data, meaning that it should not reprocess the entire raw set of data when new files are added. I ended up deciding between using partition-based file tracking via s3 and glue job bookmarks with spark. I chose to use the latter option for a couple reasons. First being that the ETL script that would correspond with using glue job bookmarks is simpler because it does not require tracking of which files have already been processed. Similarly, if the partition structure of the input file ever changed the pipeline would fail. Therefore, I chose to use glue bookmarks to handle incremental data processing.

Decision 4: Pipeline scheduling

The final decision I had to consider when building the pipeline was how to handle pipeline scheduling. Originally, I had built my pipeline to have the following architecture:



As you can see, the main differences lie in the manual glue job and glue crawler requirements. This was great for testing the pipeline on small amounts of data. However, unless I remember to re-run the glue job and crawler, no more data will be processed. So, I ended up adding a glue job trigger that runs every hour and a glue crawler trigger that runs the crawler if the glue job is successful. If the run is not successful, the pipeline would require intervention, and there would be no need to run the glue crawler until that is complete.

The other considerations I had when designing the solution include the bronze, silver and gold levels of the solution architecture. The data is in ‘bronze’ format when it is in its raw event log state. I would argue that it is somewhere between the silver and gold level when it is in the glue database that is queryable using the athena workgroup. There is the potential to add 5

external tables that correspond to the 5 analytics queries the business requested, and if given more time and resources, I would have implemented those as well.

Testing & Validation

After deploying the updated stack that provisioned the additional resources that I specified, I then moved into the testing phase. First, I completed a variety of ad hoc sql queries on the Athena workgroup table (capstone-analytics) to make sure the data was loaded with the appropriate schema and the expected size with reasonable performance. For example, I ran a simple query to check the number of rows in the table. At the time of testing, it returned >171 million rows in 1.2 seconds. Once this was confirmed, I moved on to testing the 5 analytics queries I had written to make sure that they were syntactically correct and had reasonable performance. The performance results on the table at the time of having more than 171 million entries is below.

Query	Data Scanned	Run Time (seconds)
1 - Conversion Rates	290.58 MB	4.696
2 - Hourly Revenue	452.88 MB	1.781
3 - Top 10 Products	176.15 MB	1.528
4 - Category Performance	77.62 MB	1.997
5 - User Activity	1.78 GB	4.794

Next, I needed to test that the automated pipeline scheduling worked. The way I tested this was simple, but effective. I queried my database to check the number of rows present, waited for the beginning of the next hour, plus a few minutes to allow for the glue job and crawler to finish running and re-queried the database. The number of rows that my database increased was expected based on the fact that 500-750k events are generated every five minutes. I also checked the job run logs to confirm that the number of files being processed each time the job is run is not strictly increasing to indicate the bookmarks are enabled and working.

Reflection

This project demonstrates my ability to design, build and implement an analytics pipeline using Infrastructure as Code (IaC). By creating a pipeline that is able to process and store millions of additional rows of data per hour, I proved that I am able to implement a scalable, fully automated pipeline that meets real-world business requirements. Pipelines like these help enable data-driven decision making at a business level and help turn raw or messy data into actionable insights.