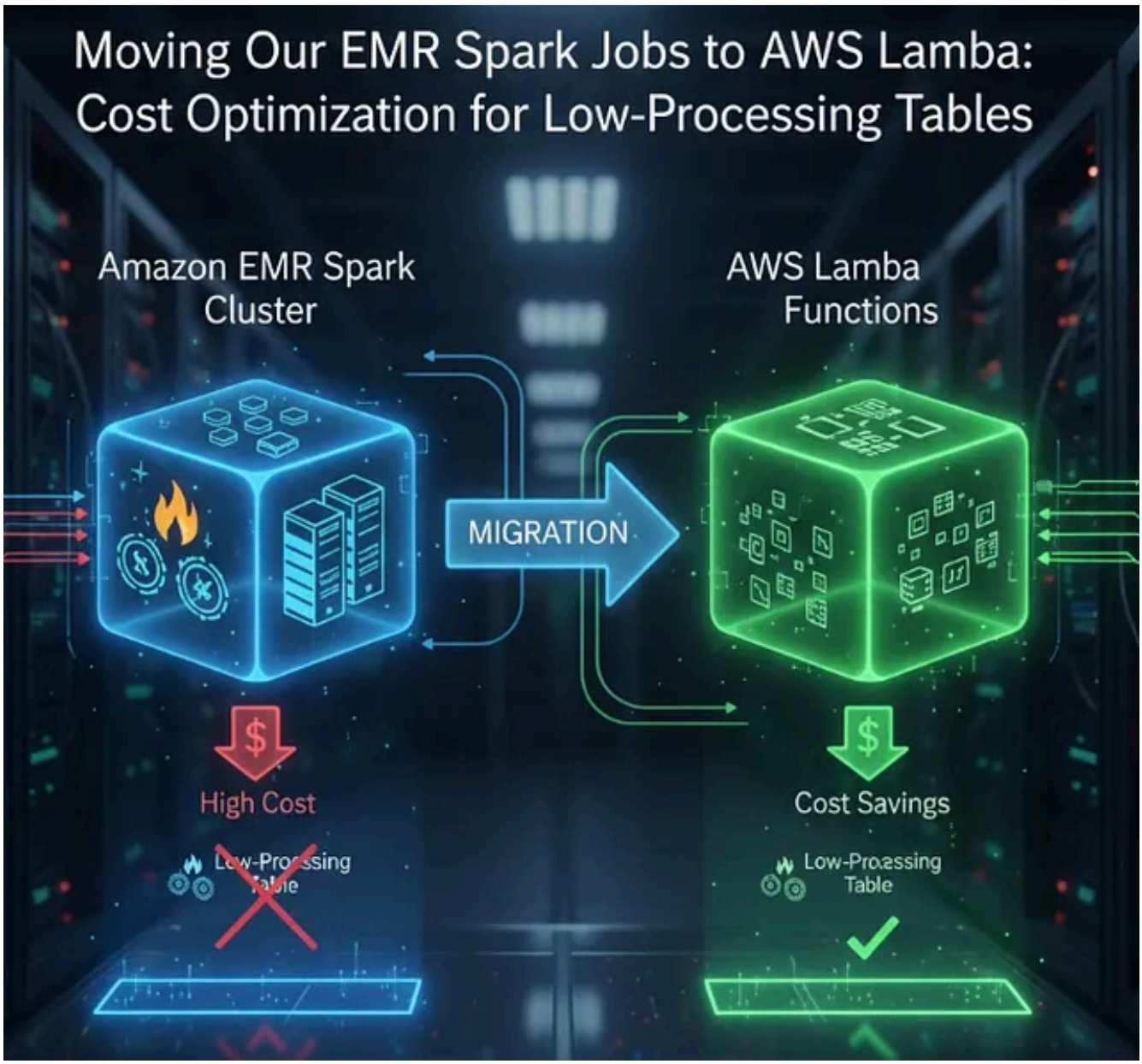


Moving Our EMR Spark Jobs to AWS Lambda: Cost Optimization for Low-

Processing Tables

Anurag Sharma 5 min read · Dec 16, 2025

In data engineering, few things are as persistent as the annual cost optimization conversation. For years, our DevOps team has been highlighting the steady 20% year-over-year growth in our AWS bill. Storage costs are largely fixed; we need to retain historical data for compliance and analytics, and the spotlight fell squarely on processing. The mandate was clear: reduce compute spend significantly while adopting a more modern, scalable approach that would keep these pipelines running smoothly for at least the next two years without requiring performance upgrades.

After a thorough review of our ETL landscape, we identified a large category of jobs that were massively over-engineered for EMR Spark: low- or no-processing tables. These included dimension tables, static references, mapping tables, and third-party vendor feeds that required little more than "pick up, lightly clean (if at all), and drop" into S3 and Redshift.

The Low-Hanging Fruit: 150+ Tables That Didn't Need Spark

We cataloged over 150 daily jobs that fell into this category:

- **Geographic mappings:** ZIP codes → city/state/county/latitude-longitude, city → state/province, country codes (ISO 3166) → full names and regions, time zones, area codes.

- **Standard code lookups:** Currency codes (ISO 4217), language codes (ISO 639), unit of measure conversions, industry classifications (SIC, NAICS, GICS), taxonomy codes, ICD-10 medical codes, and HS commodity codes.
- **Business reference tables:** Product categories and hierarchies, customer segments, user roles and permissions, department/organizational structures, holiday calendars, fiscal period mappings, feature flags, A/B test variants.
- **Other static/low-change tables:** Device type mappings, browser/OS version lookups, color palettes, error code descriptions, regulatory compliance categories (e.g., GDPR data classes), payment method codes.

These tables are small, most under 50k rows, many in the low thousands, and updated daily or weekly. Yet they were being processed on full EMR clusters alongside our heavy analytical workloads. More importantly, they power critical joins in dozens of fact and transaction tables, so reliability and timeliness matter, but raw compute power does not.

The decision was straightforward: migrate all 150+ of these jobs from EMR Spark to a lightweight, serverless architecture.

The New Architecture: Pandas on Lambda + Step Functions for Config-Driven Orchestration

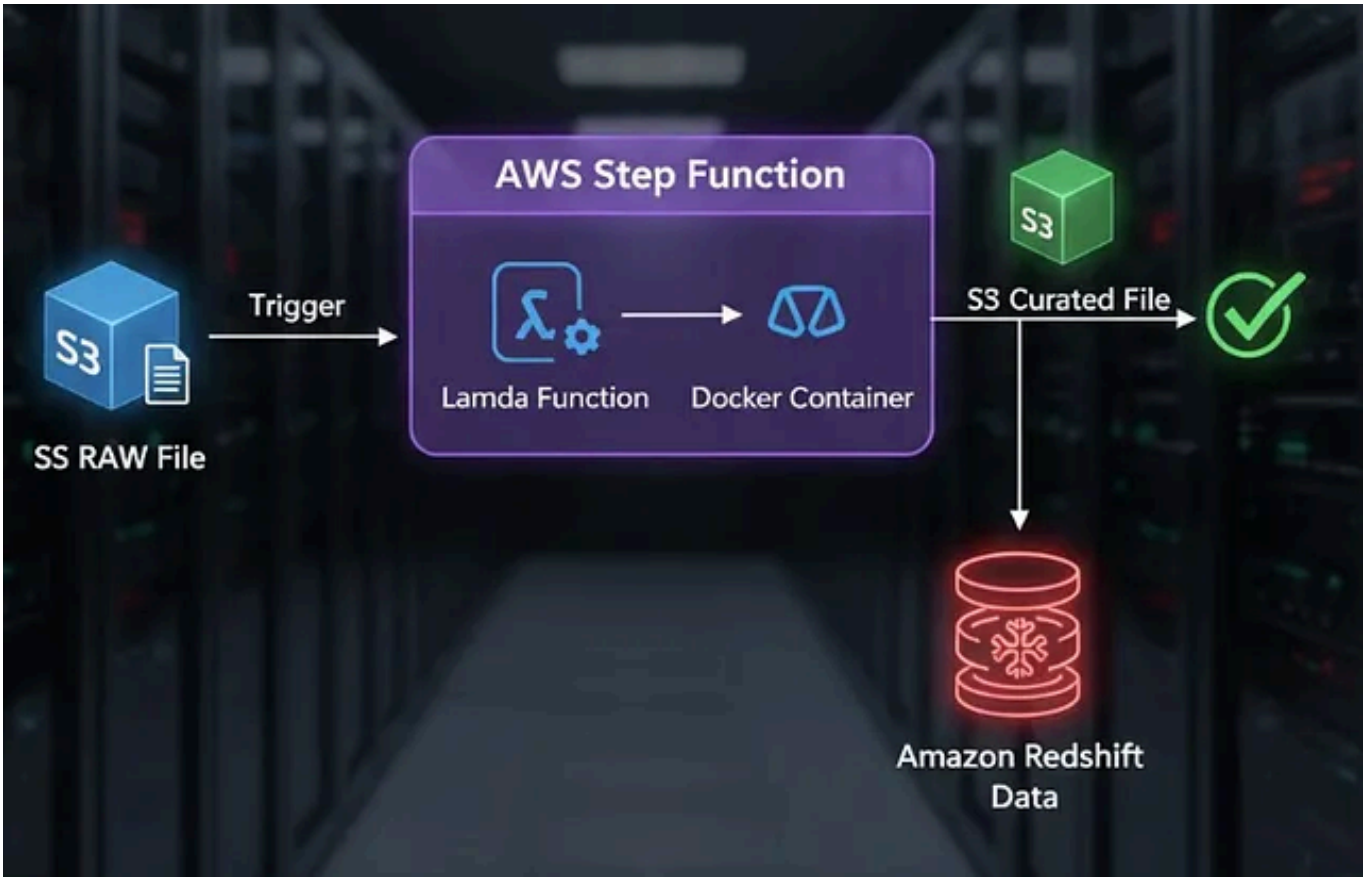
We rewrote the Spark jobs in Python using Pandas, deployed them as containerized AWS Lambda functions (via Docker images), and orchestrated them with AWS Step Functions for better configurability, monitoring, and testing.

Why Pandas + Lambda (Containerized)?

- Datasets fit easily in memory (Lambda supports up to 10 GB RAM).
- Pandas is simple, expressive, and familiar to our team — no need for the overhead of distributed frameworks.
- We avoided Koalas (or PySpark Pandas API) because it still requires a Spark context and brings unnecessary complexity and runtime overhead for single-node workloads.
- Container images allowed us to bundle exact dependencies (Pandas, openpyxl for Excel mapping sheets, boto3, psycopg2 for Redshift interactions) and keep package sizes manageable.

Moving to a Framework-Driven, Configurable Design

Instead of hard-coding logic per table, we built a generic ingestion framework driven by configuration. This made onboarding new tables trivial and reduced code duplication dramatically.



Core components:

- **Configuration file** (JSON stored in S3): Contains table metadata source format, delimiter, header row, target Redshift schema/table, truncate-vs-upsert behavior, etc.
- **Column mapping file** (Excel/CSV stored in S3): Defines column renaming, simple transformations (e.g., uppercase, strip whitespace), data type casting, and default values.
- **Orchestration**: AWS Step Functions coordinates the multi-step process.
- **Trigger**: S3 event notification when a new export file lands in the raw bucket.

Example Flow: Daily ZIP Code Table Ingestion

File Arrival Vendor ZIP code export (CSV) lands in s3://raw-bucket/zipcodes/yyyy-mm-dd/file.csv. This triggers an S3 event, which in turn executes Step Functions, passing the bucket/key and table identifier.

Step 1: Data Ingestion & Transformation Lambda

- Reads the table-specific JSON config from S3.
- Downloads and parses the mapping Excel sheet (using openpyxl/pandas).
- Reads the source CSV into a Pandas DataFrame.
- Applies mappings: rename columns, cast types, add derived columns if needed, drop duplicates, and basic validation.
- Writes cleaned data as Parquet to curated zone: s3://curated-bucket/zipcodes/part-00000.parquet.
- Outputs the curated S3 path and row count for downstream steps.

Step 2: Redshift Load Lambda

- Executes Redshift COPY command from the curated Parquet file(s).
- Optionally truncates the target table first or uses a staging table + merge for upserts.
- Logs row count loaded.

Step 3: Data Quality Testing Lambda

- Runs predefined tests from config: → Row count comparison (source vs. target) → Uniqueness on primary keys —> Completeness (NOT NULL checks on required columns) —> Referential integrity spot checks —> Custom SQL assertions
- Inserts a row into a central data_quality_tests audit table in Redshift with pass/fail status, metrics, and execution timestamp.
- On any failure: → Sends detailed alert email via Amazon SES. → Creates an incident ticket in ServiceNow via API.

```
{
  "table_name": "dim_zipcode",
  "redshift_schema": "public",
  "redshift_table": "dim_zipcode",
  "source_format": "csv",
  "header_row": 1,
  "delimiter": ",",
  "mapping_file": "s3://config-bucket/mappings/zipcode_mapping.xlsx",
  "truncate_before_load": true,
  "quality_tests": [
    {"type": "row_count_match", "tolerance_pct": 0},
    {"type": "uniqueness", "columns": ["zipcode"]},
    {"type": "not_null", "columns": ["zipcode", "city", "state"]}
  ]
}
```

Results and Benefits

- **Cost Impact**: Processing costs for these 150+ pipelines dropped by 40–60%. No more paying for EMR cluster hours; we now pay only for seconds of Lambda execution and Step Functions state transitions.

- **Operational Simplicity**: One generic codebase handles all tables. Adding a new mapping table is now a matter of dropping config + mapping files →no code deployment needed.
- **Reliability & Observability**: Built-in quality checks catch vendor issues early; audit trail in Redshift makes compliance audits easier.
- **Scalability**: If a table unexpectedly grows, we can increase Lambda memory/timeout or selectively migrate it back to EMR without affecting others.
- **Developer Velocity**: Team members can iterate faster with pure Python/Pandas than with Spark SQL.

This migration proved that not every ETL job needs a big data hammer. For the dozens of small-but-essential tables in every data warehouse, a lightweight serverless approach with thoughtful orchestration can deliver massive savings and cleaner operations.

If your organization is grappling with similar cost pressures, start by auditing your EMR job runtime and dataset sizes.

You might be surprised how many clusters are spinning up just to move a few thousand rows.

AWS | Cost Optimization | Big Data | Spark | AWS Lambda