Implementation Summary:
As part of the **CareFirst data migration initiative**, I led the **first phase of the project**, which focused on automating EDI data ingestion and validation workflows using AWS services.

The pipeline architecture included:
S3 (EDI intake) → SQS (message queuing) → Lambda (Python-based JSON conversion) → AWS Glue (ETL and cataloging) → Athena (query and validation of valid/invalid claims).

This phase successfully established the **foundation for automated, scalable, and query-ready healthcare claims processing**, improving data availability and integrity for downstream analytics.

1) Storage setup (S3 Buckets)

Why:

Lading zone: need a landing zone for raw data and separate curated zones for processed data. Keeping raw and curated separate ensures traceability and easier debugging.

How:

Created buckets

1) hc_raw_bucket
    a. folder – edi
2) hc_processed_bucket
    a. folder – claims_raw
    b. folder – claims_validated
    c. folder – claims_rejected
    d. ref
        i. master-plan

Result: Files are safely stored and versioned (if enabled).

2) Creating a custom policy for glue to access S3

Console path: IAM → Policies → Create policy → JSON → paste the JSON below → Next → Name: ProjectS3Access-GlueJob → Create.

Json:

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Sid": "ListBucketsLimited",
   "Effect": "Allow",
   "Action": [
    "s3:ListBucket"
   ],
   "Resource": [
    "arn:aws:s3:::hc-raw-bucket",
    "arn:aws:s3:::hc-curated-bucket"
   ],
   "Condition": {
    "StringLike": {
     "s3:prefix": [
      "edi/*",
      "csv/*",
      "txt/*",
      "claims_raw/*",
      "claims_validated/*",
      "claims_rejected/*",
      "ref/plan_master/*"
   ]
  }
 }
},
{
 "Sid": "ReadFromRawAndCuratedInputs",
 "Effect": "Allow",
 "Action": [
  "s3:GetObject"
 ],
 "Resource": [
  "arn:aws:s3:::hc-raw-bucket/edi/*",
  "arn:aws:s3:::hc-raw-bucket/csv/*",
  "arn:aws:s3:::hc-raw-bucket/txt/*",
  "arn:aws:s3:::hc-curated-bucket/claims_raw/*",
  "arn:aws:s3:::hc-curated-bucket/ref/plan_master/*"
 ]
},
{
 "Sid": "WriteCuratedOutputs",
 "Effect": "Allow",
 "Action": [
  "s3:PutObject"
 ],
 "Resource": [
  "arn:aws:s3:::hc-curated-bucket/claims_validated/*",
  "arn:aws:s3:::hc-curated-bucket/claims_rejected/*"
 ]
},
{
 "Sid": "GlueCatalogRead",
 "Effect": "Allow",
 "Action": [
  "glue:GetDatabase",
  "glue:GetDatabases",
  "glue:GetTable",
  "glue:GetTables"
 ],
 "Resource": "*"
},
{
 "Sid": "CloudWatchLogsForJob",
 "Effect": "Allow",
 "Action": [
  "logs:CreateLogGroup",
  "logs:CreateLogStream",
  "logs:PutLogEvents",
  "logs:DescribeLogStreams"
```

```
   ],
   "Resource": "*"
  }
 ]
}
```

3) Creating Role which can access the storage and glue with the defined policies.

Console path: IAM → Roles → Create role

- **Trusted entity:** AWS service → **Glue**
- **Use case:** Glue
- **Attach policies:** select the **ProjectS3Access-GlueJob** policy you just created
- **Role name:** role_s3_access
- Create.

4) SQS + S3 Event (exact clicks & the one policy you need)

A) Create the queues

1. **SQS → Create queue → Standard**
    - Name: edi-intake-queue
    - Leave most defaults.
2. **Create a DLQ**
    - Name: edi-intake-dlq

    - After creating both, open **edi-intake-queue → Dead-letter queue** and attach edi-intake-dlq with **Max receives = 5**.

B) Allow S3 to send messages to SQS (queue access policy)

Open **edi-intake-queue** → **Permissions** → **Access policy** → **Edit** and paste this, changing only the bucket name if yours differs:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "AllowS3SendMessage",
    "Effect": "Allow",
    "Principal": { "Service": "s3.amazonaws.com" },
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:<REGION>:<ACCOUNT_ID>:edi-intake-queue",
    "Condition": {
      "ArnLike": { "aws:SourceArn": "arn:aws:s3:::hc-raw-bucket" }
    }
  }]
}
```

Replace <REGION> and <ACCOUNT_ID> with yours.
This is the **only policy** you need right now for S3→SQS; S3 does not use your role_s3_access.

C) Wire S3 event to the queue

S3 → **hc-raw-bucket → Properties → Event notifications → Create**

- Name: object-created-to-sqs
- Event type: **All object create events**
- Prefix: x12/ (or edi/ if that's your folder)
- Destination: **SQS queue** → pick edi-intake-queue

D) Quick test

- Upload any tiny file to hc-raw-bucket/edi/ (e.g., test.edi).
- Go to **SQS → edi-intake-queue → Monitoring** and confirm the **Number of messages** increases.

5) Lambda router/parser (SQS → S3 JSON)

Why

- Turn raw uploads into **structured** records your visual ETL can use.
- Only EDI needs parsing; starting with a simple extractor gets you moving fast.
- Using SQS as trigger makes this **reliable** (retries, DLQ).

How

A. Permissions policy (least privilege for Step 3)

Create a customer-managed policy (IAM → Policies → Create → JSON):

Name: ProjectLambda-EDIParser

```
{
 "Version": "2012-10-17",
 "Statement": [
  { "Sid": "ReadQueue",
   "Effect": "Allow",
   "Action": [
    "sqs:ReceiveMessage",
    "sqs:DeleteMessage",
    "sqs:GetQueueAttributes"
   ],
   "Resource": "arn:aws:sqs:us-east-2:<ACCOUNT_ID>:edi-intake-queue"
  },
  { "Sid": "ReadRawEDI",
   "Effect": "Allow",
   "Action": [ "s3:GetObject", "s3:ListBucket" ],
    "Resource": [
      "arn:aws:s3:::hc-raw-bucket",
      "arn:aws:s3:::hc-raw-bucket/edi/*"
    ]
  },
  { "Sid": "WriteClaimsRawJSON",
    "Effect": "Allow",
    "Action": [ "s3:PutObject", "s3:ListBucket" ],
    "Resource": [
      "arn:aws:s3:::hc-curated-bucket",
      "arn:aws:s3:::hc-curated-bucket/claims_raw/*"
    ]
  },
  { "Sid": "WriteLogs",
    "Effect": "Allow",
    "Action": [
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents",
      "logs:DescribeLogStreams"
    ],
    "Resource": "*"
  }
]
```

```
}
```

B) Create the IAM role for Lambda

- IAM → Roles → Create role → **Trusted entity: AWS service**
- **Use case: Lambda**
- This sets the trust policy to:

```
{
 "Version": "2012-10-17",
 "Statement": [{
  "Effect": "Allow",
  "Principal": { "Service": "lambda.amazonaws.com" },
  "Action": "sts:AssumeRole"
 }]
}
```

C) Create the Lambda function

- **Runtime:** Python 3.12
- **Role:** role-lambda-parser
- **Environment variables:**
- CURATED_BUCKET=hc-curated-bucket
- OUTPUT_PREFIX=claims_raw/
- **Memory/timeout:** 256 MB / 30 sec (fine for small files)

Code

```
import json, os, urllib.parse, boto3, datetime, re


s3 = boto3.client('s3')
CURATED_BUCKET = os.environ['CURATED_BUCKET']
```

```
OUTPUT_PREFIX = os.environ.get('OUTPUT_PREFIX', 'claims_raw/')


def _parse_edi_minimal(text: str) -> dict:
    """
    Super-minimal, safe extractor for demo:
    - pulls a few fields if present; otherwise leaves them blank
    - YOU can replace this with a proper X12 parser later
    """
    segments = [seg for seg in text.split('~') if seg.strip()]
    claim_id = None
    svc_date = None
    plan_code = None

    for seg in segments:
        el = seg.split('*')
        tag = el[0].upper().strip()

        # Example: ST*837*0001 -> claim batch/control id in ST02
        if tag == 'ST' and len(el) > 2 and el[1] == '837':
            claim_id = el[2]

        # BHT*0019*00*0123*20250913*1200*CH -> date in BHT04
        if tag == 'BHT' and len(el) > 4:
            d = el[4]
            if re.fullmatch(r'\d{8}', d):
                svc_date = f"{d[0:4]}-{d[4:6]}-{d[6:8]}"
        # Placeholder: set a default fake plan code; replace with real mapping later
        if tag == 'HI':
            plan_code = plan_code or 'PPO01'

    return {
        "claim_id": claim_id or "",
        "svc_date": svc_date or "",
        "plan_code": plan_code or ""
    }

def lambda_handler(event, context):
    # SQS -> body contains S3 event JSON
    for rec in event.get('Records', []):
        body = rec.get('body', '')
        try:
            payload = json.loads(body)
        except json.JSONDecodeError:
            # Some S3->SQS setups wrap JSON in Message field; try that
            try:
                payload = json.loads(json.loads(body).get('Message', '{}'))
            except Exception:
                print("Could not parse message body:", body[:500])
                continue

        records = payload.get('Records', [])
```

```
for r in records:
    b = r['s3']['bucket']['name']
    k = urllib.parse.unquote(r['s3']['object']['key'])

    # fetch the file
    obj = s3.get_object(Bucket=b, Key=k)
    text = obj['Body'].read().decode('utf-8', errors='replace')

    # minimal parse
    parsed = _parse_edi_minimal(text)

    # add metadata
    today = datetime.date.today().isoformat()
    out_key = f"{OUTPUT_PREFIX}dt={today}/type=837/{os.path.basename(k)}.jsonl"

    # write a single JSON line (extend to multiple rows if your file holds many claims)
    line = json.dumps({
        "source_bucket": b,
        "source_key": k,
        **parsed
    })

    s3.put_object(
        Bucket=CURATED_BUCKET,
        Key=out_key,
        Body=(line + "\n").encode('utf-8')
        )

    return {"ok": True}
```

Deploy it

Set environment variables

1. Go to the Configuration tab → Environment variables → Edit → Add environment variable:
    - Key: CURATED_BUCKET → Value: hc-curated-bucket
    - Key: OUTPUT_PREFIX → Value: claims_raw/
2. Save.

Set memory and timeout

1. Configuration tab → General configuration → Edit.
2. Memory: set to 256 MB.
3. Timeout: set to 0 min 30 sec (30 seconds).
4. Save.

Add the SQS trigger

1. Add trigger (left pane or top button) → SQS.
2. SQS queue: choose edi-intake-queue.
3. Batch size: set 1 (easier to debug first).
4. Leave the rest default → Add.

(Your role already has sqs:ReceiveMessage/DeleteMessage; this just creates the event source mapping.)

This is intentionally simple: it writes **one JSON line** per file. If later your 837 has many claims, you can loop and emit multiple lines.

Quick test (end-to-end)

1. Upload test.edi to s3://hc-raw-bucket/edi/test.edi.
2. Wait a few seconds.
3. In **S3 → hc-curated-bucket → claims_raw/dt=…/type=837/** you should see a file like test.edi.jsonl.
4. Open it; you should see a single JSON line with source_bucket, source_key, and the parsed fields.

6) Glue Catalog & Crawlers

1 Create the Glue Database

Why

You need a logical place (schema) for your tables so Glue and Athena can find them.

How

- Console → **AWS Glue** → **Data Catalog** → **Databases** → **Add database**
  - **Name:** hc_claims_db
  - Leave the rest default → **Create**

Result

Empty database hc_claims_db exists (tables will appear after crawlers run).

2 Create the Crawler Role and Policy (one-time)

Why

Crawlers need permission to read your S3 paths and write table metadata to the Data Catalog.

How

1. **Policy** (IAM → **Policies** → Create → JSON)
   Name: ProjectGlueCrawler-Base

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadCuratedBucketList",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": "arn:aws:s3:::hc-curated-bucket"
    },
    {
      "Sid": "GetObjectsCuratedAndRef",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::hc-curated-bucket/ref/plan_master/*",
        "arn:aws:s3:::hc-curated-bucket/claims_raw/*",
        "arn:aws:s3:::hc-curated-bucket/claims_validated/*",
        "arn:aws:s3:::hc-curated-bucket/claims_rejected/*"
      ]
    },
    {
      "Sid": "GlueCatalogReadWrite",
      "Effect": "Allow",
      "Action": [
        "glue:GetDatabase",
        "glue:GetDatabases",
        "glue:GetTable",
        "glue:GetTables",
        "glue:CreateTable",
        "glue:UpdateTable",
        "glue:GetPartitions",
        "glue:GetPartition",
        "glue:GetPartitionIndexes",
        "glue:BatchGetPartition",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:UpdatePartition"
      ],
      "Resource": "*"
    },
    {
      "Sid": "CrawlerLogging",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Resource": "*"
    }
  ]
}
```

3.**Role** (IAM → **Roles** → Create role)

  - **Trusted entity:** AWS service → **Glue**
  - **Attach policy:** ProjectGlueCrawler-Base
  - **Role name:** role-glue-crawler → **Create**

Result
role-glue-crawler exists and can read ref/plan_master/ + claims_raw/, and write tables/partitions in the Catalog.

4 Crawler A — plan_master (reference CSV)

Why
Registers your reference lookup so ETL can join on plan_code (no code needed).

How
Glue → **Crawlers** → **Create crawler**

- **Name:** crawler-plan-master
- **Data sources: S3** → **Include path:** s3://hc-curated-bucket/ref/plan_master/
- **IAM role:** role-glue-crawler

- **Target database:** hc_claims_db
- **Table name prefix:** *(blank)*
- **Recrawl behavior:** Crawl new and changed partitions only
- **Schema change policy:** Update table definition and add new partitions
- **Schedule:** On demand (run manually now)

Click **Run crawler**. Wait until **Ready**.

Result
Table **hc_claims_db.plan_master** appears with columns:
plan_code, plan_name, effective_from, effective_to.

5 Crawler B — claims_raw (your Lambda JSONL)

Why
Registers the parsed claims so Glue Studio & Athena can read them. Also detects **partitions** from folder names (dt=YYYY-MM-DD, type=837).

How
Glue → **Crawlers** → **Create crawler**

- **Name:** crawler-claims-raw
- **Data sources: S3** → **Include path:** s3://hc-curated-bucket/claims_raw/
- **IAM role:** role-glue-crawler
- **Target database:** hc_claims_db
- **Recrawl behavior:** Crawl new and changed partitions only
- **Schema change policy:** Update table definition and add new partitions
- **Grouping behavior:** Keep as default (Glue will infer partitions from dt=.../type=...)
- **Schedule:** On demand (run now)

Click **Run crawler**.

Result
Table **hc_claims_db.claims_raw** appears with JSON-derived columns (e.g., source_bucket, source_key, claim_id, svc_date, plan_code) and **partitions**: dt, type.

7. Glue Studio Visual ETL (claims_raw → validated/rejected)

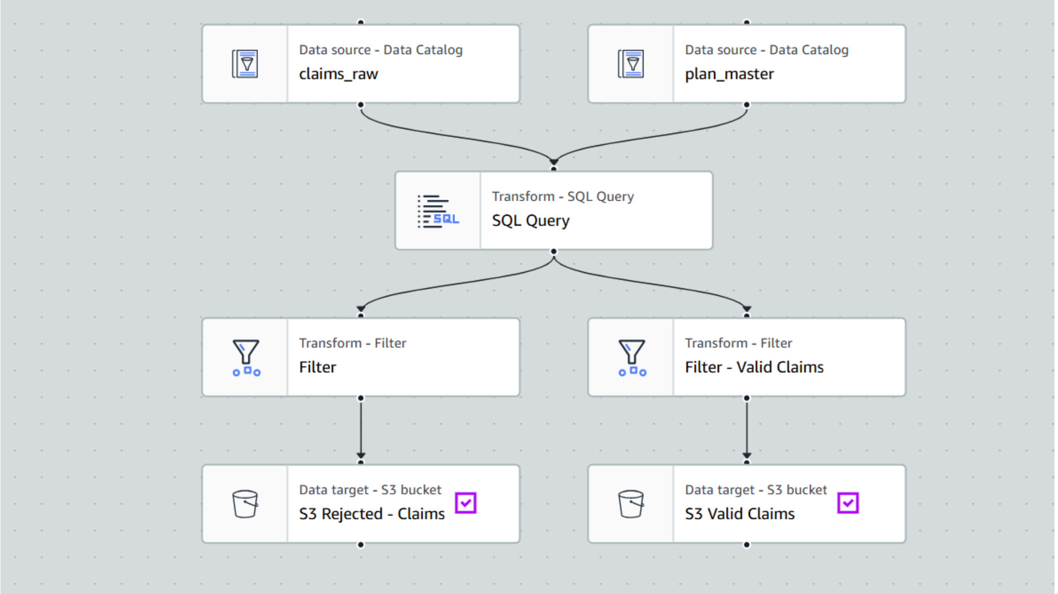
Create a glue job:

Why
This is your main transformation: join raw claims with the reference table, apply rules (SSN/DOB/Plan), and split into good vs bad—**without coding**.

How

- Go to **AWS Glue → ETL jobs → Visual → Create job**.
- **Sources:** add two Catalog sources:
  - hc_claims_db.claims_raw
  - hc_claims_db.plan_master
- **IAM role:** choose **role_s3_access** (the Glue-trusted role you created with S3 permissions).
- **Job name:** edi-validate-visual

Result
A new visual job with two source nodes ready to connect.



SQl Query:

```
SELECT
 claims_raw.source_bucket,
 claims_raw.source_key,
 claims_raw.claim_id,
 TO_DATE(claims_raw.svc_date, 'yyyy-MM-dd')         AS svc_date,
 claims_raw.plan_code,                          -- This is the claim's plan code

 plan_master.plan_name,                          -- This is the plan's name from the master table
 TO_DATE(plan_master.effective_from, 'yyyy-MM-dd')     AS effective_from,
 TO_DATE(plan_master.effective_to,   'yyyy-MM-dd')     AS effective_to,

 CASE
  WHEN plan_master.plan_code IS NOT NULL
  THEN TRUE ELSE FALSE
 END AS rule_plan_exists,
 CASE
  WHEN claims_raw.svc_date IS NOT NULL
   AND TO_DATE(claims_raw.svc_date, 'yyyy-MM-dd')
       BETWEEN TO_DATE(plan_master.effective_from, 'yyyy-MM-dd')
           AND TO_DATE(plan_master.effective_to,   'yyyy-MM-dd')
  THEN TRUE ELSE FALSE
 END AS rule_plan_active,

 CASE
  WHEN plan_master.plan_code IS NOT NULL
   AND TO_DATE(claims_raw.svc_date, 'yyyy-MM-dd')
       BETWEEN TO_DATE(plan_master.effective_from, 'yyyy-MM-dd')
           AND TO_DATE(plan_master.effective_to,   'yyyy-MM-dd')
  THEN 1 ELSE 0
 END AS good_flag,

 CASE
  WHEN plan_master.plan_code IS NOT NULL
   AND TO_DATE(claims_raw.svc_date, 'yyyy-MM-dd')
       BETWEEN TO_DATE(plan_master.effective_from, 'yyyy-MM-dd')
           AND TO_DATE(plan_master.effective_to,   'yyyy-MM-dd')
  THEN 'VALID' ELSE 'REJECTED'
 END AS status,
 CASE
  WHEN plan_master.plan_code IS NULL THEN 'PLAN_NOT_FOUND'
  WHEN NOT (
   TO_DATE(claims_raw.svc_date, 'yyyy-MM-dd')
   BETWEEN TO_DATE(plan_master.effective_from, 'yyyy-MM-dd')
     AND TO_DATE(plan_master.effective_to,   'yyyy-MM-dd')
  ) THEN 'PLAN_NOT_ACTIVE_ON_SVC_DATE'
  ELSE NULL
 END AS failure_reason,

 claims_raw.dt,
 claims_raw.type
FROM claims_raw LEFT JOIN plan_master ON claims_raw.plan_code = plan_master.plan_code;
```

Write Parquet outputs (partitioned)

Why

Parquet + partitions = tiny Athena cost, fast reads, production-style layout.

How

- For the good branch, add a S3 target:
  - Format: Parquet
  - S3 path: s3://hc-curated-bucket/claims_validated/
  - Partition keys: dt, type
- For the bad branch, add another S3 target:

- Format: Parquet
- S3 path: s3://hc-curated-bucket/claims_rejected/
- Partition keys: dt, type

Result

Two curated datasets appear in S3, partitioned by dt and type.

8. Automate the ETL job with the help of Event Bridge

Create new policy for event bridge:

glue-start-edi-validate-visual-policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "glue:StartJobRun",
      "Resource": "arn:aws:glue:us-east-2:517542309978:job/DemoTesting"
    }
  ]
}
```

Now Create a role specifically for the ETL Automation.

EventBridge-start-Glue-Demo

Assign that create policy here.

Now Create Event Bridge policy.

Create policy

Name it

Select Event Source ()

Custom pattern event Json Editor

```
{
 "source": ["aws.s3"],
 "detail-type": ["Object Created"],
 "detail": {
  "bucket": {
   "name": ["hc-curated-bucket"]
  },
  "object": {
   "key": [{
    "prefix": "claims_raw/"
   }]
  }
 }
}
```

Select target Source : Glue Job

Select job

Create

8. Automation Update the Catalog after the job.

Glue Trigger:

Name: trg-crawl-curated-on-success

Event Source Job: DemoTesting – Succeeded – Action : Start crawler : cr-curated-outputs

Result : As soon as the job is successful, the partitions and the database is up to date for the athenea query.

9. Alerts:

SNS Topic: data-pipeline-alerts

Subscription : Email

Event Bridge Rule: evb-notify-edi-job-status

```
{
 "source": ["aws.glue"],
 "detail-type": ["Glue Job State Change"],
 "detail": {
  "jobName": ["edi-validate-visual"],
  "state": ["FAILED","SUCCEEDED","TIMEOUT"]
 }
}
```

Targe SNS Topic: data-pipeline-alerts

Result: Email on every success/failure; alarms if DLQ grows or parser fails.

10. Query Layer, Athena

Why: Interactive validation checks for updated results.

Tables: Claims_raw, Claims_validated, Claims-Rejected, Plan_master

DB: HC_Claims_db

11. Final resource count

- **S3:** 2 buckets, ~9 prefixes
- **SQS:** 2 queues (main + DLQ)
- **Lambda:** 1 function (edi-parser-lambda)
- **Glue database:** 1 (hc_claims_db)
- **Glue tables:** 4 (plan_master, claims_raw, claims_validated, claims_rejected)
- **Glue crawlers:** 2–3 (plan_master, claims_raw, curated_outputs) depending on job catalog settings
- **Glue jobs:** 1 (edi-validate-visual)
- **EventBridge rules:** 1–2 (start job, notify)
- **Glue trigger or EB rule:** 1 (kick crawler on success)
- **SNS topic:** 1 (data-pipeline-alerts)
- **IAM roles:** 3–4 (lambda, glue job, crawler, eventbridge optional)

12. End-to-end runtime flow

1. **Producer** drops file.edi → s3://hc-raw-bucket/edi/
2. **S3 event → SQS** (edi-intake-queue)
3. **Lambda (edi-parser-lambda)** consumes queue, parses .edi, writes JSONL → s3://hc-curated-bucket/claims_raw/dt=YYYY-MM-DD/type=837/<guid>.jsonl
4. **Automation**
   - *Path A:* EventBridge → Lambda → glue.start_job_run('edi-validate-visual')
   - *Path B:* EventBridge → Glue Workflow → job
5. **Glue Job (edi-validate-visual)** reads claims_raw (+ plan_master), validates, writes:
   - claims_validated/dt=…/type=837/*.parquet
   - claims_rejected/dt=…/type=837/*.parquet
   - (and updates tables if you enabled "create/update table")
6. **Crawler/Trigger** updates claims_validated / claims_rejected partitions (if the job didn't)
7. **Alerts** — EventBridge sends **SNS email** on job SUCCEEDED/FAILED
8. **Athena** — analysts query hc_claims_db tables for counts, trends, reasons