GLUE SCENARIO BASED Q&A

BY - SHUBHAM WADEKAR

1. How would you use Glue components (Jobs, Crawlers, Catalog, Triggers) to move raw data from S3 to Redshift automatically every hour?

If I had to build this pipeline, I'd set it up in a way where each Glue component plays a very specific role.

First, the raw data is coming into S3. I'd organize the bucket with partitions by date and hour, something like s3://bucket/raw/topic/dt=YYYY-MM-DD/hr=HH/. This makes it easier to process incrementally instead of scanning the entire bucket each time.

Next, I'd configure a Glue Crawler to scan that S3 path and create or update a table in the Glue Data Catalog. The crawler helps me avoid manually defining schemas, and I can choose to run it either every hour before the job or maybe once a day if the schema doesn't change often. The Catalog then becomes the central metadata store that the Glue job can reference.

Then comes the Glue Job. I'd create a PySpark-based Glue ETL job that reads from the Catalog table, applies any required transformations like casting datatypes, dropping duplicates, or filtering out bad records, and then writes the cleaned data into Amazon Redshift. I'd use a Glue Redshift connection with temporary S3 staging, so the job essentially does a COPY under the hood, which is faster than row-by-row inserts. I'd usually write into a staging table in Redshift first, and then merge it into the final reporting table to handle duplicates or late-arriving data.

To automate this, I'd use a Glue Trigger with a schedule that runs every hour. The trigger can also be chained — for example, first run the crawler, then run the ETL job, and finally run a post-load step if needed. With triggers, I don't have to rely on external schedulers.

On top of this, I'd enable Glue job bookmarks or pushdown predicates to make sure each hourly job only processes the new partition, not the full history. That keeps the job efficient and cost-effective.

From a practical side, I'd also make sure Redshift credentials are stored in AWS Secrets Manager, the Glue job runs inside the same VPC as Redshift for network access, and monitoring is set up with CloudWatch and SNS to alert if a job fails.

So overall, the flow would look like this:
S3 (raw files) → Glue Crawler updates Catalog → Glue Job transforms and loads into Redshift → Trigger runs the pipeline every hour.

2. Your S3 bucket has millions of partitioned files. How would you make Glue Crawlers faster and cheaper?

I'd treat the crawler as a metadata refresher, not a file scanner. The goal is to minimize what it touches and how often it runs.

1. Point the crawler at the smallest possible scope
   Keep the include path at the partition root you actually load (for example, s3://bucket/raw/events/dt=). Avoid pointing at the whole bucket. Add exclusion patterns for old years, backup prefixes, _temporary/, _SUCCESS, and any logs. If you have multiple datasets, use multiple small crawlers instead of one giant one.
2. Crawl new partitions only
   In the crawler settings, pick the recrawl policy "Crawl new folders only." This stops it from re-checking existing partitions every run. Also choose "Update all new and existing partitions only when the schema changes" to avoid churn on unchanged objects.
3. Reduce sampling and file-type confusion
   Use narrow, correct classifiers (e.g., JSON or CSV with the right delimiter/quote/escape) and remove generic ones you don't need. Keep sampling per partition tiny (a couple of files). The crawler only needs enough to infer schema—don't let it read tens of files just to be sure.
4. Partition deeper, but crawl shallower
   Keep a clean, predictable layout (like dt=YYYY-MM-DD/hr=HH). The crawler only needs to see the top-level partition folders to register them; don't let it descend into nested junk. If you have very deep hierarchies, split by dataset so each crawler's max depth is small.
5. Don't crawl what rarely changes
   If the schema is stable, don't run the crawler hourly. Run it daily or weekly for schema drift, and handle hourly arrivals without the crawler:
   • Read by path in your ETL (from_options) and write to your target.
   • Or programmatically add just the new partitions via BatchCreatePartition from a Glue job or a tiny Lambda. This event-driven registration (S3 PUT → Lambda → Glue BatchCreatePartition) is usually faster and cheaper than crawling.
6. Use time-boxed or sliding windows
   If you must use a crawler often, keep it on a sliding window prefix, like "current week" or "last N days." Move older data under an archived prefix the crawler never scans.
7. Keep small files under control
   Crawlers spend time listing and sampling. Compact tiny files upstream (or with a daily compaction job) so there are fewer objects to list and sample. Even if you don't change the total bytes, reducing object count speeds the listing phase.
8. Separate schema discovery from partition discovery
   Create the table once (manually or with a one-time crawl on a small sample). After that, stop full crawls. Only register new partitions programmatically. This avoids re-inference and keeps the table stable.
9. Limit IAM and network hops
   Give the crawler an IAM role scoped only to the paths it needs. Avoid cross-account or cross-region listings unless required; use resource links or data sharing patterns instead of crawling remote buckets.

10. Operational guardrails
    Add tight schedules and timeouts so a crawler can’t run for hours. Alert on long runtimes or excessive “objects scanned.” Keep separate crawlers per dataset so one problematic path doesn’t slow everything.

If I summarize the practical path I’d actually take: create the table once on a small sample, set a narrow crawler (or none), switch to event-driven BatchCreatePartition for hourly partitions, and reserve the crawler for occasional schema changes. That combination has given me the biggest speed and cost win on multi-million-file lakes.

3. Custom Classifier If the default classifier fails on complex nested logs, how would you build a custom Glue classifier?

I first pin down the log structure (JSON, semi-structured text, XML). Then I create a custom classifier that tells the crawler exactly how to parse it, and I give that classifier higher priority than the defaults so it's always picked.

What I do step-by-step:

1. Capture a small, representative sample of the logs in S3, including edge cases and bad rows.
2. Decide the classifier type:
   • JSON: when it's valid JSON but nested or irregular.
   • Grok: when it's text with repeated patterns (e.g., Apache, app logs).
   • CSV: when it's delimited but not standard (e.g., custom separator, escapes).
   • XML: if it's XML with namespaces.
3. Build the classifier:
   • JSON classifier: define a JSONPath that points to the array or object that represents "one record." Example: if each file is { "records": [{...}, {...}] }, I set JSONPath to $.records[*].
   • Grok classifier: write a grok pattern that captures the fields. Example pattern for a timestamped log line with level and JSON payload:

   ```
   %{TIMESTAMP_ISO8601:ts} %{LOGLEVEL:level} \[%{DATA:service}\]
   %{GREEDYDATA:msg} payload=%{GREEDYDATA:payload}
   ```

Later, in the ETL job, I parse payload as JSON to explode nested fields.
• CSV classifier: specify delimiter, quote, escape, header presence, and custom date formats.
• XML classifier: set an XPath to the repeating element, e.g., //event.

4. Assign the classifier to the crawler and move it to the top of the classifier priority list. I also add an include path that is as tight as possible and exclude everything that doesn't match (e.g., *.gz only if that's true).
5. Run the crawler on the sample first and validate the Data Catalog table: check column types, arrays/structs, and that partition keys are correctly inferred (I keep partitions like dt, hr in the S3 path, not in the payload).
6. Stabilize the schema: if nested fields are volatile, I map them to a stable shape in ETL (use defaults for missing fields, cast types). The classifier should only discover structure, not enforce business schema.
7. Guardrails: set crawler recrawl policy to "new folders only" and sampling to a few files per partition. Add CloudWatch alarm if the crawler starts touching too many objects (signals drift).

Minimal code I keep handy for ETL after a grok classifier extracted payload as text:

```
from pyspark.sql.functions import from_json, col

schema = "user_id string, event_name string, meta struct<os:string, app_ver:string>"

df2 = df.withColumn("payload_json", from_json(col("payload"), schema)) \

    .select("*", "payload_json.*").drop("payload","payload_json")
```

This keeps the classifier simple and pushes deep parsing to the job where I can control types cleanly.

4. Duplicate Tables - A crawler created separate tables for CSV and Parquet files of the same dataset. How would you keep only one logical table?

I pick one canonical format and one canonical prefix, and I make the crawler only see that. Everything else becomes staging or is excluded.

What I do step-by-step:

1. Choose the “source of truth.” I prefer Parquet as the single table because it’s columnar, typed, and cheaper to query.
2. Separate storage paths:
   • Move or write CSV to a staging prefix like s3://bucket/dataset_staging/ (short-lived).
   • Keep Parquet in s3://bucket/dataset/ with partitions like dt=YYYY-MM-DD/hr=HH/.
3. Make the crawler deterministic:
   • Point the crawler only to the Parquet prefix.
   • Add exclude patterns for *.csv and for the staging prefix.
   • Set recrawl policy to “new folders only.”
4. Clean up the Catalog:
   • Tag the unwanted CSV table as deprecated, confirm no queries/jobs rely on it, then drop it.
   • If the Parquet table has the right name, keep it; otherwise, rename it to the logical dataset name and update downstream jobs to the new table.
5. Enforce the pipeline:
   • Ingestion writes CSV to staging.
   • A Glue job converts CSV → Parquet and writes into the canonical prefix.
   • The crawler (or better, programmatic partition registration) updates the single Parquet table only.
6. Prevent re-appearance of duplicates:
   • Separate crawlers per dataset with tight include paths.
   • Strict exclusion patterns (*.csv, /staging/*, _temporary/, _SUCCESS).
   • If multiple teams land files, publish a storage contract: “only Parquet in the curated path.”
7. Optional modernization:
   • If the team wants ACID and schema evolution, create a single Iceberg table in the Catalog and always write to that table. Then deprecate both CSV/Parquet ad-hoc tables. This gives you one logical table regardless of how data arrived upstream.

In short: make Parquet the only curated location, point the crawler only there, drop the duplicate CSV table, and keep CSV strictly as a transient staging step.

5. Your lake has Parquet (historical) + JSON (incremental). How would you design crawlers/classifiers so Athena can query both together?

I keep discovery per format but make the schemas line up, then expose a single logical view for consumers.

1. create two Glue tables with the same business schema
   • One crawler for Parquet pointing at the curated historical prefix.
   • One crawler for JSON pointing at the incremental prefix. Attach a JSON classifier with a JSONPath that targets the record array or object per line.
   • In both places, enforce identical column names, types, and partitions (for example, dt and hr in the S3 path). If JSON is loose, map missing fields to null later in ETL.

2. control crawler behavior to avoid drift
   • Recrawl "new folders only."
   • Tight include paths; exclude tmp files, _SUCCESS, and unrelated datasets.
   • Keep the JSON classifier at higher priority than default so the crawler never misclassifies.

3. give Athena one thing to query
   Option A (fastest to roll out): create an Athena view that unions the two tables and standardizes types.
   Option B (preferred long term): add a small Glue job that converts the JSON incrementals to Parquet on arrival and writes into the same Parquet table/partitions as historical. Then the crawler (or programmatic partition adds) touches only the Parquet table, and analysts query one table.
   Option C (modern pattern): use an Apache Iceberg table as the single logical table. Land JSON to a staging area, run a lightweight Glue job to upsert/append into the Iceberg table. The crawler registers only the Iceberg table; Athena queries it directly.

4. practical guardrails
   • If you must union in Athena, cast JSON columns to the Parquet schema explicitly and align timestamps/timezones.
   • Keep partition keys in the path, not the payload.
   • Validate schema parity with a daily check: list columns and types from both tables and alert on mismatches.

A minimal Athena view when keeping two physical tables:

CREATE OR REPLACE VIEW analytics.events_all AS

SELECT cast_cols(*) FROM raw.events_parquet

UNION ALL

SELECT cast_cols(*) FROM raw.events_json;

Here cast_cols is just shorthand for explicit casts so both sides match perfectly.

6. How would you design a Glue ETL job to keep only the latest session record per user in clickstream data? Would you use DynamicFrames or DataFrames?

I'd use Spark DataFrames for the core logic because window functions and performance tuning are cleaner there. I read via the Catalog as a DynamicFrame, convert to a DataFrame, do the dedupe with a window, and write back.

1. inputs and keys
   • Source has user_id, session_id, session_start, session_end, updated_at (or event_time).
   • "Latest" is defined clearly up front: usually the max(updated_at) for each user_id, or max(session_end) if that's your truth.
2. incremental read
   • Enable Glue job bookmarks or filter by dt/hr to only process new partitions.
   • If late data is common, reprocess a small sliding window (for example, last 6 hours) and dedupe downstream.
3. dedupe with a window
   • Partition by user_id, order by updated_at desc (or session_end desc), pick row_number = 1.
   • If there can be ties, add a deterministic tiebreaker (for example, session_id, ingestion_ts).

Minimal PySpark skeleton:

```
from pyspark.sql import functions as F, Window

from awsglue.context import GlueContext

from awsglue.dynamicframe import DynamicFrame


# read (from Catalog)

src = glueContext.create_dynamic_frame.from_catalog(

  database="raw_db",

  table_name="clickstream_sessions",

  push_down_predicate=predicate_for_window  # e.g., dt >= '2025-08-22'

)

df = src.toDF()


# choose "latest" per user

w = Window.partitionBy("user_id").orderBy(F.col("updated_at").desc(),
F.col("session_id").desc())

df_latest = df.withColumn("rn", F.row_number().over(w)).where("rn = 1").drop("rn")
```

```
# write to curated (e.g., Parquet or Iceberg)
glueContext.write_dynamic_frame.from_options(
  frame=DynamicFrame.fromDF(df_latest, glueContext, "df_latest"),
  connection_type="s3",
  connection_options={"path": "s3://lake/curated/sessions_latest/", "partitionKeys": ["dt"]},
  format="parquet"
)
```

4. performance and correctness
   • Repartition by user_id before the window if data is skewed; use salting if a few users are extremely heavy.
   • Select only required columns before the window to cut shuffle size.
   • If the target is a slowly changing “latest” table, write to a staging location and then atomically swap, or use Iceberg and do an overwrite by filter.
   • Validate with row counts: total distinct user_id in the output equals distinct user_id in the input window.

5. why DataFrames over DynamicFrames
   • DataFrames give direct access to Spark SQL window functions, better Catalyst optimizations, and a larger ecosystem of functions.
   • I still use DynamicFrames for easy Catalog IO and then convert back when writing if needed. This keeps the code concise without giving up Spark capabilities.

7. How would you join a small customer dataset with a huge transaction dataset in Glue efficiently?

I keep the big scan on the transactions side and broadcast the small customer table so the join happens map-side without shuffling the big data.

• Read transactions (large) partition-pruned, read customers (small) with only needed columns.
• Convert to Spark DataFrames, cache the small one, and broadcast it.

```
tx = glueContext.create_dynamic_frame.from_catalog(
  database="raw_db", table_name="transactions",
  push_down_predicate="dt='2025-08-22'"
).toDF().select("txn_id","user_id","amount","dt")


cust = glueContext.create_dynamic_frame.from_catalog(
  database="ref_db", table_name="customers"
).toDF().select("user_id","segment","country")


from pyspark.sql.functions import broadcast
joined = tx.join(broadcast(cust), "user_id", "left")
```

• Make sure the small table actually fits in memory on each executor (tens/hundreds of MB is fine). If it's larger, reduce it (select only needed columns, dedupe, filter by country/active flags).
• Keep the large side partitioned by the join key if possible, and push down date/hour filters so you don't scan history.
• Handle skew: if a few user_ids dominate, enable adaptive execution and skew join, or salt the hot keys.
• File formats: Parquet/Iceberg for both sides; avoid CSV.
• Resource tuning: set adequate DPUs, and limit shuffle by broadcasting (no massive repartitions).
• If customers live in Redshift or a DB, extract them first to S3 Parquet as a small dim before the job; don't inner-join over JDBC.
• Validate: count distinct user_id before/after, sample the join, and log join stats.

This pattern (broadcast the small, filter the big) gives the biggest win with the least tuning.

8. Your Glue job keeps reprocessing the same files into Redshift. How would you use Job Bookmarks to fix this?

I enable bookmarks, make sure the source node is bookmark-aware, and keep the job identity stable so Glue can remember what it has already read.

• Turn on bookmarks on the job:

- Job parameter: --job-bookmark-option=job-bookmark-enable (use pause if you need a temporary backfill, disable to turn it off).
  • Read from S3 using a bookmark-aware source and give it a stable transformation_ctx:

```
src = glueContext.create_dynamic_frame.from_catalog(

  database="raw_db",

  table_name="transactions_raw",

  transformation_ctx="src_tx"   # important for bookmarks

)
```

Glue will track S3 object path + last modified so unchanged files aren't re-read.

• Keep the same job name between runs. If you rename or recreate the job, you lose the bookmark state and it will re-ingest.
• Don't point the reader at giant, changing roots. Keep a stable prefix (e.g., partitioned by date/hour). If you rewrite old files, Glue may see them as "new" due to modified timestamps—avoid file overwrites in processed partitions.
• For JDBC sources, set bookmark keys (e.g., updated_at) so only new rows are read; for S3, no keys are needed—bookmarks work off the file metadata.
• Idempotent load into Redshift:

- Land into a staging table, then MERGE into the target on a business key to prevent duplicates if a reprocess ever happens.
- Optionally store a high-watermark (max event_time) in a control table for extra safety.

• Operations you'll actually use:

- Reset bookmarks after a one-time backfill: --job-bookmark-option=job-bookmark-reset.
- Pause bookmarks to reprocess a window: --job-bookmark-option=job-bookmark-pause.
- Monitor bookmark metrics and log how many files/bytes were considered "new" each run.

With bookmarks enabled, a stable source context, and an idempotent Redshift load pattern, the job stops reprocessing the same S3 files.

9. If your S3 dataset has mismatched field names/types compared to Redshift, how would you transform it in Glue before loading?

I fix names and types in Glue so Redshift sees clean, compatible columns. I keep it simple: read via Catalog, standardize, cast, then load.

What I'd do:

- Start with a clearly defined Redshift target schema (names, types, nullability, timezones).
- Read S3 data through the Glue Catalog to get basic structure, then convert to a Spark DataFrame for precise control.
- Rename columns to match Redshift exactly (snake_case, avoid reserved words like "user", "timestamp"), and trim weird characters or spaces.
- Cast types to Redshift-friendly ones: use boolean/int/bigint/decimal(precision,scale)/timestamp without timezone. Parse strings to timestamps and numbers; normalize enums/text values.
- Normalize time to UTC before casting to timestamp. Parse ISO-8601 or epoch safely.
- Handle nulls/defaults and widen/narrow types thoughtfully (e.g., string → decimal(18,2), string → bigint). If downcasting, guard against overflow with filters or safe casts.
- Validate with quick checks (row counts, null ratios, bad-cast counts) before writing.
- Load to a staging table in Redshift, then merge into the final table.

Minimal PySpark sketch:

```
from pyspark.sql import functions as F, types as T

src = glueContext.create_dynamic_frame.from_catalog(

  database="raw_db", table_name="orders_raw", transformation_ctx="src"

).toDF()


df = (src

  .withColumnRenamed("OrderID", "order_id")

  .withColumnRenamed("TotalPrice", "total_amount")

  .withColumn("order_id", F.col("order_id").cast("bigint"))

  .withColumn("total_amount", F.col("total_amount").cast(T.DecimalType(18,2)))

  .withColumn("created_at_utc",

    F.to_utc_timestamp(F.to_timestamp("created_at_str", "yyyy-MM-dd'T'HH:mm:ssX"),
"UTC"))

  .drop("created_at_str")

)
```

```
# Optional: handle bad rows
df_ok = df.filter(F.col("order_id").isNotNull() & F.col("total_amount").isNotNull())


glueContext.write_dynamic_frame.from_jdbc_conf(
  frame=DynamicFrame.fromDF(df_ok, glueContext, "df_ok"),
  catalog_connection="redshift_conn",
  connection_options={"dbtable": "stg_orders", "database": "analytics"},
  redshift_tmp_dir="s3://company-temp/glue-redshift/"
)
```

Extra things I watch:

- Decimal precision/scale must fit Redshift target.
- Strings that should be booleans (“true/false”, “Y/N”) get mapped explicitly.
- If the source schema drifts, I put a mapping layer (ApplyMapping or a select with explicit casts) that fails fast on unexpected fields and alerts.
- I avoid reprocessing by enabling job bookmarks and by always landing in a staging table followed by a MERGE into the final table keyed on a business key (e.g., order_id).

10. You have deeply nested JSON with arrays/structs. How would you flatten it in Glue, and when would you prefer DynamicFrames over DataFrames?

If I need a single flat table, I explode arrays and pull struct fields up. If the data is truly hierarchical, I create multiple related outputs (fact + child tables). I choose the tool based on control vs convenience.

How I flatten with DataFrames (most control, best performance for complex logic):

- Read via Catalog → convert to DataFrame.
- Use explode_outer for arrays to avoid losing rows when arrays are empty.
- Pull struct fields with dot notation and alias them.
- Repeat explode for nested arrays; if there are multiple arrays, flatten in stages to avoid a blow-up in row counts.
- Add surrogate keys (e.g., event_id) before exploding so I can join child tables back if needed.

Sketch:

```
from pyspark.sql import functions as F


df = glueContext.create_dynamic_frame.from_catalog(
   database="raw_db", table_name="events_json"
).toDF()


# Example: events has user struct and items array<struct<sku,qty,price>>
df1 = df.withColumn("item", F.explode_outer("items")) \
      .select(
         "event_id",
         F.col("user.id").alias("user_id"),
         F.col("user.country").alias("user_country"),
         F.col("ts").alias("event_ts"),
         F.col("item.sku").alias("item_sku"),
         F.col("item.qty").alias("item_qty"),
         F.col("item.price").alias("item_price")
      )
```

If I need multiple outputs (star schema):

- Create events_head (one row per event).
- Create events_items (one row per event-item).
- Optionally, events_attributes using map_from_entries and explode if there's a dynamic attributes map.

How I flatten with DynamicFrames (quickest to get many child tables from nested JSON):

- Use the built-in Relationalize transform, which automatically generates a set of flat tables (DynamicFrames) with keys that preserve relationships.
- Good when the JSON structure is variable and I need to land several related tables without hand-writing explodes.

Sketch:

```
from awsglue.transforms import Relationalize

root = glueContext.create_dynamic_frame.from_catalog(database="raw_db",
table_name="events_json")

rel = Relationalize.apply(frame=root, name="events")

# rel is a dict-like; write each table

for name in rel.keys():

  glueContext.write_dynamic_frame.from_options(

    frame=rel[name],

    connection_type="s3",

    connection_options={"path": f"s3://lake/curated/events_rel/{name}/"},

    format="parquet"

  )
```

When I prefer DataFrames:

- I need strong performance tuning (pruning columns early, controlling shuffles).
- I need complex business rules, custom dedupe, conditional transforms, or carefully staged explodes.
- I want to integrate window functions and advanced Spark APIs.

When I prefer DynamicFrames (Relationalize):

- I want a fast path to break a nested payload into multiple flat outputs with referential keys.
- The schema changes often and I'm okay with the auto-generated table names/keys.
- I'm prioritizing speed-to-production over hand-optimized transforms.

Operational notes I always apply:

- Limit the scan with partition predicates; select only columns I actually need before exploding.
- Beware of multiple large arrays causing Cartesian growth; sometimes it’s better to keep separate child tables than to fully flatten into one giant table.
- After flattening, cast to Redshift-compatible types and write Parquet to S3 for a fast COPY/MERGE.
- If analysts query in Athena, I may keep nested Parquet for some use cases and only flatten fields required for Redshift.

11) How would you orchestrate three Glue jobs (Clean → Aggregate → Load) so they run in order automatically?

I would use a Glue Workflow with conditional triggers so the jobs run in sequence and only proceed on success.

Step-by-step:

- Create one Glue Workflow to represent the full pipeline.
- Add three Glue jobs: Clean, Aggregate, and Load.
- Add a Start trigger that launches the Clean job when the workflow starts.
- Add a conditional trigger for Aggregate with "on success of Clean".
- Add a conditional trigger for Load with "on success of Aggregate".
- Pass runtime parameters through the workflow (for example, processing date, S3 input/output paths, and run ID). I set these as workflow run properties and map them to each job's arguments.
- Set maximum concurrency to 1 within the workflow path so only the intended next step runs.
- Make each job idempotent:
    - Clean writes to a dated/atomic output path (e.g., s3://.../clean/dt=YYYY-MM-DD/), then does a success marker file only after write completes.
    - Aggregate reads only the success-marked partition from Clean and writes its own success marker.
    - Load reads only success-marked Aggregate output and writes in an idempotent way to the target (e.g., MERGE into Redshift/Iceberg table or overwrite partition).
- Add monitoring and alerts:
    - Enable CloudWatch metrics and logs for each job.
    - Add EventBridge rules or CloudWatch Alarms on job failure to notify the on-call channel.
- Optional hardening:
    - If I need time-based automation, I create an EventBridge schedule to start the workflow daily with the right date parameter.
    - Use Glue Catalog partitions and data quality checks (e.g., row count thresholds) between steps; fail fast if thresholds are not met, so downstream steps don't run with bad data.

This gives a clean, deterministic chain: Clean → Aggregate → Load, with clear success conditions and parameters flowing end to end.

12) If Job B fails in a workflow, how would you design retries so the pipeline doesn't restart from scratch?

I make retries localized to the failing job and ensure each job is restart-safe. Concretely:

- Configure retries at the job level:
    - Set Job B's "number of retries" to a sensible value (for example, 2–3) with exponential backoff. This lets transient issues clear without touching Job A or C.
- Use conditional triggers and failure branches:
    - Keep the success chain A → B → C.
    - Do not auto-restart the workflow; let the workflow pause at B on failure.
    - After fixing the issue (or automatically on next scheduled run), re-run only B using the same workflow run properties (date/run_id) so it processes the same input.
- Make B idempotent and checkpointed:
    - Inputs: B should only read success-marked outputs from A (so reruns see the same stable inputs).
    - Outputs: write to a temp/staging location, then atomically move/rename to final (or write to a partition with a success marker). On retry, B detects partial old attempts and cleans/overwrites that partition before writing.
    - If B writes to a warehouse, use MERGE/UPSERT semantics keyed by a deterministic batch_id or natural keys, so repeated attempts don't duplicate rows.
- Use a progress manifest:
    - Keep a small DynamoDB table (or S3 manifest) keyed by run_id + task shard. Mark completed shards for B. On retry, B skips shards already marked done and only processes the missing ones.
- Isolate state via parameters:
    - Pass run_id and processing_date from the workflow to all jobs. B uses these to find the exact input/output paths. This guarantees that re-running B doesn't accidentally target a new day's data.
- Add guardrails and observability:
    - CloudWatch alarms on B's error rate.
    - Structured error logs so I can quickly see which shard/partition failed.
    - Optional dead-letter pattern: if B still fails after N retries, emit an EventBridge event to open a ticket/notification, but keep A's success intact and do not roll back the entire workflow.
- Resume C only after B success:
    - The trigger for C requires B's success condition. Once B finally succeeds (after retries), C runs automatically with the same parameters—no need to re-run A.

This approach keeps retries local to Job B, avoids re-running completed steps, and ensures the pipeline can safely resume from where it failed without duplicating data.

13) How would you trigger a Glue Workflow automatically when a new file arrives in S3?

I use S3 → EventBridge to fire the workflow, with light filtering and a small guard to avoid duplicate/partial files.

Practical steps:

- Turn on S3 event delivery to EventBridge for the bucket.
- Create an EventBridge rule:
  - source = aws.s3
  - detail-type = Object Created
  - filter by bucket name
  - filter by object key prefix/suffix (for example, prefix = raw/events/, suffix = .parquet)
  - ignore folder markers or temp paths (exclude keys containing tmp/, _temporary/, or =_SUCCESS).
- Set the rule's target to "AWS SDK" → Glue → StartWorkflowRun. In input transformer, pass dynamic params like:
  - bucket, key (from the event)
  - processing_date derived from the key (if you partition by dt=YYYY-MM-DD)
  - a run_id (EventBridge event id)
- Make the workflow idempotent:
  - Clean job reads the exact S3 key from parameters, or the exact partition.
  - Write outputs to a deterministic path (for example, s3://.../clean/dt=YYYY-MM-DD/) and create a _SUCCESS marker only after a complete write.
  - Aggregate/Load only consume success-marked data, so a re-fire won't corrupt results.
- Add a small safety buffer if files are uploaded via multipart:
  - In the Clean job, re-check size > 0 and use a short object-exists + ETag-stability check (or a 1–2 minute delay in EventBridge via a second rule) to avoid reading a file that's still being uploaded.
- For burst control:
  - If you expect many small files, route S3 events → SQS → Lambda, batch keys, and then StartWorkflowRun once per batch. This keeps Glue from over-scheduling.
- Monitoring and alerts:
  - CloudWatch metric filters on workflow run failures.
  - EventBridge rule for Glue Workflow state change → SNS/Slack.

This gives a simple, no-Lambda (or minimal-Lambda) path: new file → EventBridge rule → StartWorkflowRun with the right parameters, plus filtering so only the right files trigger runs.

14) How would you combine Glue Workflows with Step Functions to get centralized failure monitoring?

I keep Glue Workflows for data-step orchestration inside a domain, and use Step Functions as the top-level controller for runs, retries, and alerts—all visible in one place.

How I set it up:

- One Step Functions state machine per pipeline family.
- First task: “Start Glue Workflow Run” using the AWS SDK integration (glue.startWorkflowRun) with parameters like processing_date and run_id.
- Poll for completion:
  - Add a loop: Wait (for example, 60–120 seconds) → GetWorkflowRun (glue.getWorkflowRun) → check status.
  - Exit on Succeeded; go to Catch on Failed/Stopped/TimedOut.
- Centralized retries:
  - In the Step Functions task that polls, set a Retry policy (max attempts, exponential backoff) for transient API errors.
  - Do not restart the entire pipeline on business failures; let the Glue Workflow itself keep job-level retries local to the failing job.
- Centralized failure handling:
  - Use a Catch block that publishes to SNS/Slack, creates a ticket, or writes an incident record to DynamoDB.
  - Include the workflowRunId, failing node, and processing_date in the alert context so the on-call can re-run the exact slice.
- Timeouts and SLAs:
  - Put an overall timeout in Step Functions (for example, 3 hours). If exceeded, mark the run as failed centrally even if Glue is still busy, and alert.
- Drill-down visibility:
  - Step Functions gives an execution history and one “red/green” status per run.
  - From the alert or dashboard, link to the Glue Workflow run page and CloudWatch logs of the failing job for details.
- Optional: multiple workflows:
  - If the pipeline has stages owned by different teams, orchestrate each stage as its own Glue Workflow. Step Functions runs them in sequence or in parallel, with per-stage Catch/Retry, keeping one pane of glass for status and alerts.

This structure lets Glue Workflows do what they’re best at (job chaining and data-aware steps), while Step Functions provides a single control point for monitoring, retries, timeouts, and notifications across the whole pipeline.

15) How would you configure a job to run only after multiple upstream jobs complete successfully?

I use a Glue Workflow with a conditional trigger that waits for all required upstream jobs to succeed.

Practical setup:

- Create one Glue Workflow that contains all jobs.
- Add the upstream jobs (for example: Ingest, Clean, Validate). Set them to run in parallel if they do not depend on each other.
- Create a conditional trigger for the downstream job (for example: Aggregate). In the trigger, select "on success" of Ingest, Clean, and Validate. The trigger condition uses an AND rule, so Aggregate starts only when all three report Succeeded.
- Pass a single set of workflow-run parameters (for example, processing_date and run_id) to every job so they all operate on the same slice of data.
- Make upstream jobs write a success marker after finishing, and write outputs to deterministic, date-partitioned paths. The downstream job reads only success-marked partitions. This avoids starting on partial data if an upstream job left temporary files.
- Keep retries local to each upstream job. If one fails, fix or retry only that job; the downstream trigger will not fire until all are green.
- Add CloudWatch alarms on each job's failure metrics and a workflow-level failure event to notify the team.

This gives a clean fan-in pattern: multiple upstreams can run in parallel, and the dependent job starts automatically only when all inputs are ready and successful.

16) How would you design a Glue Streaming Job that reads from Kinesis and writes to S3 in Parquet with low latency?

I build a Spark Structured Streaming job on Glue Streaming that reads Kinesis Data Streams, does light transforms, and writes Parquet to S3 with a small micro-batch interval and proper checkpointing.

Key design points:

1. Source and schema

- Use the Kinesis Data Streams source with starting position set to LATEST for new pipelines, or TRIM_HORIZON if I need historical replays.
- Define the schema up front for speed and stability. If messages are JSON, I parse with from_json using a fixed StructType. If the producer uses Glue Schema Registry (Avro/Protobuf), I deserialize with the registry library so schema evolution is managed.

2. Exactly-once and checkpoints

- Configure a unique checkpointLocation in S3. This stores Kinesis offsets and batch state, so the job can restart without duplicating data.
- Use deterministic output paths and rely on Structured Streaming's batch-idempotency so each micro-batch writes once even after retries.

3. Low latency tuning

- Set a short trigger interval (for example, processingTime = 30 seconds). I go lower (5–15 seconds) only if the business really needs it.
- Right-size workers to shard count. As a rule of thumb, at least one executor per 1–2 Kinesis shards. Choose WorkerType G.1X or G.2X and set numberOfWorkers based on throughput tests.
- Tune Kinesis read options to balance latency and throughput (for example, limit per-shard fetch to avoid large spikes).
- Keep transformations lightweight. Avoid wide shuffles inside the streaming job. If I must aggregate, use watermarks and time windows to bound state, or move heavy aggregations to a separate batch job.

4. Durable, analytics-friendly S3 writes

- Write to S3 in Parquet with Snappy compression. Partition by high-cardinality-but-stable keys like dt=YYYY-MM-DD and optionally hour if needed for querying and lifecycle policies.
- Expect many small files due to low-latency batches. Schedule a separate compaction job (hourly or daily) that merges small Parquet files into larger ones (for example, 128–512 MB) for Athena/EMR performance.
- Maintain a Glue Catalog table over the curated path so Athena/Redshift Spectrum can query it. Update partitions automatically using a crawler or by programmatically adding partitions at the end of each batch.

5. Backpressure, errors, and DLQ

- Enable backpressure by keeping the trigger small and making sure executors have headroom. Monitor Kinesis iterator age; if it grows, add workers or reduce per-batch work.
- For bad records, use a try-parse pattern: separate good rows and bad rows; write bad rows to an S3 dead-letter path with the original payload and error for later reprocessing.

6. Reliability and operations

- Put the job in “continuous” mode with automatic restarts on failure. Keep checkpoints in a stable S3 location that is not rotated.
- Add CloudWatch dashboards for Glue job metrics, Kinesis iterator age, and S3 error rates. Set alarms on failures and rising iterator age.
- Use IAM least privilege: read from the Kinesis stream, write to target S3 prefixes, write to checkpoint and DLQ prefixes, and update the Glue Catalog as needed.

7. Minimal code structure (conceptual)

- Read stream from Kinesis with options for region, stream name, and starting position.
- Parse JSON to columns, add ingestion_time, and derive dt/hour.
- Write stream to S3 in Parquet with checkpointLocation set, trigger set to the chosen latency, and partitionBy(["dt","hour"]).
- Keep the job parameters for stream name, target paths, micro-batch interval, and checkpoint path, so I can promote across environments without code changes.

This setup gives near–real-time delivery (tens of seconds), safe restarts with no duplicates, Parquet files ready for analytics, and a path to scale by adding workers or compaction without redesigning the pipeline.

17. If Kafka events arrive late, how would you handle them in a Glue Streaming Job so reports stay accurate?

I treat the stream in event time, not processing time, and let late records correct earlier results without double counting.

1. Parse and standardize event time early
   I extract event_time from the payload, convert it to a proper timestamp, and standardize timezone. This becomes the single clock for windows and aggregations.
2. Use watermarks to accept "late but not too late" data
   I apply a watermark on event_time based on the business SLA (for example 1 hour, 6 hours, or 1 day depending on how late events typically arrive). The watermark bounds state so Spark keeps only the windows that can still change. Anything later than the watermark is dropped or diverted to a quarantine table for investigation.
3. Choose the right windowing and keys
   For rollups I use tumbling or sliding windows on event_time plus the business key (for example user_id, product_id). That ensures the same window is updated when a late event for that key shows up.
4. Make the sink upsert-friendly so late data can "fix" history
   I write to a table format that supports MERGE/updates such as Delta Lake, Apache Hudi, or Apache Iceberg on S3. I keep a stable composite key like (window_start, window_end, dimension_keys) for aggregates, or a unique event_id for fact tables. With foreachBatch I upsert so a late record updates the previous aggregate instead of creating duplicates.
5. Keep a deduplication guarantee
   If the producer includes a unique event_id, I dropDuplicates by event_id within a watermark. If not, I create a deterministic id (hash of business keys + event_time + payload). This protects reports from replays.
6. Serve reports in a way that tolerates late data
   For BI, I either query the upserted lakehouse table directly (Athena/Presto/EMR) or refresh downstream warehouses from that table. For dashboards, I mark the most recent, still-open windows as provisional until the watermark closes them.
7. Operability and guardrails
   I monitor state-store size and watermark delay, alert if the lateness distribution shifts, and periodically review the watermark value. I also keep a dead-letter path for extremely late or malformed events so I can backfill them with a controlled batch if needed.

Minimal code shape (only to show intent):

- withWatermark on event_time
- groupBy window(event_time, "15 minutes") and keys
- foreachBatch to MERGE into Delta/Hudi/Iceberg by window key

18. A Glue Streaming Job reprocesses Kafka offsets after restart, causing duplicates. How would you fix it with checkpointing?

I let Spark recover offsets from a stable checkpoint in S3 and make the sink idempotent so even if a micro-batch replays, results don't duplicate.

1. Use a durable checkpointLocation and never change it for this job
   I configure a single S3 path like s3://bucket/checkpoints/glue-stream-job-1/. I ensure lifecycle policies don't delete it. This path stores Kafka offsets and state so on restart Spark continues from the last committed offsets.

2. Do not set startingOffsets after the first successful run
   I might use startingOffsets=latest or earliest only for the initial bootstrap. After that, I remove startingOffsets and let Spark use the checkpoint. Forcing startingOffsets on restarts overrides checkpointed offsets and causes reprocessing.

3. Keep a consistent consumer group
   Spark's Kafka reader uses the checkpoint to manage group offsets. If I change the checkpoint path or the job identity, Kafka treats it as a new consumer group and replays. I keep the job name, checkpoint path, and configuration stable.

4. Make writes idempotent (exactly-once effect at sink)
   Best option is an upsert-capable table format. In foreachBatch, I MERGE by a unique event_id so replayed records update the same rows rather than insert duplicates. If writing files only, I deduplicate within a watermark using event_id before writing.

5. Handle transactional producers correctly
   If producers use Kafka transactions, I set read_committed so I don't read aborted messages. This avoids weird replays when a producer aborts and retries.

6. Operational hygiene
   I run only one active instance of the job per checkpoint path. I protect the checkpoint S3 prefix from accidental deletion. If I must reset the checkpoint (major schema change), I write to a new sink and backfill once, then switch readers to the new table to avoid mixing old/new offsets.

In short: stable S3 checkpoint for offset recovery, no startingOffsets after bootstrap, consistent consumer identity, and an idempotent upsert sink. This combination prevents duplicates after restarts.

19. How would you enrich Kinesis streaming data with a static S3 reference dataset in Glue?

I join the streaming events with a static lookup table that I load from S3 at job start (and refresh it on a schedule if it changes). The pattern is a stream–static join with a broadcast of the static data so the join stays fast.

1. Load and prepare the reference data once
   I store the reference table on S3 in a columnar format (Parquet/Delta) with clean keys. In the Glue job's initialization code (outside the stream), I read it as a static DataFrame, select only needed columns, and cache/broadcast it. If it changes daily, I reload it every N batches inside foreachBatch.
2. Read and parse the Kinesis stream
   I read the Kinesis stream, parse JSON, and extract the join key from the event. I also normalize timezones and types so keys match exactly.
3. Do a stream–static broadcast join
   Spark allows joining a streaming DataFrame with a static DataFrame. Because the reference is static and relatively small, I broadcast it to avoid shuffles. For large reference tables, I keep them in Delta with partitioning and pre-filter before broadcast.
4. Handle misses and data quality
   I mark unmatched keys, route them to a dead-letter path for later fixes, and keep the main pipeline flowing. If the reference table uses SCD Type 2, I either pre-resolve effective records into a point-in-time snapshot before the join or handle validity range inside foreachBatch using event_time.
5. Refresh strategy for the static table
   If the reference changes, I refresh it safely:

- If changes are infrequent: reload every X minutes or every M batches (e.g., when batchId % 60 == 0).
- If using Delta: read the latest version each refresh, so readers get an atomic snapshot.

6. Write enriched output to an upsert-friendly sink
   I write to Delta/Hudi/Iceberg, with a unique event_id for idempotency. If needed, I upsert aggregates in foreachBatch.

Minimal shape (Scala/PySpark-like, shortened just to show the idea):

- read Kinesis → parse → eventsDf(key, cols...)
- refDf = spark.read.parquet("s3://.../ref/").select(key, attrs...).cache()
- enriched = eventsDf.join(broadcast(refDf), Seq("key"), "left")
- foreachBatch to write and occasionally refresh refDf if needed

Key Glue considerations

- Keep reference data compact (only needed columns), use Parquet/Delta, and add Z-ORDER/partitioning on the join key if the table is large.
- Broadcast only when the reference fits in executor memory; otherwise pre-filter by a Bloom list or split the join by hash buckets.
- Monitor null join rates; high misses usually mean bad keys or delayed reference updates.

20. Your Glue Streaming Job lags when processing 100k events/sec. What tuning steps would you take to scale and reduce latency?

I scale the source, the compute, and the sink together, then remove bottlenecks in shuffles, state, and writes. I also cap intake to what the system can stably process.

1. Ensure enough parallelism at the source
   For Kinesis, I provision enough shards. Each shard has throughput limits; to handle 100k events/sec, I increase shard count so I can read in parallel across many shards. I also set read options to fetch more per read (for example, increase records per shard per fetch) while watching latency.
2. Right-size Glue workers and concurrency
   I move to larger/faster workers (e.g., G.2X/G.4X) and increase number of workers so total cores match or exceed the parallelism from shards. I keep only one streaming job instance per checkpoint path. I give executors enough memory headroom for state and broadcast joins.
3. Tune micro-batch cadence and intake
   I pick a small but stable trigger (for example 5–10 seconds) so batches are frequent and short. I cap the intake per batch to control processing time:

- For Kafka: set maxOffsetsPerTrigger.
- For Kinesis: raise per-shard fetch limits only to the point where batch time stays below trigger interval; don't overload the batch.

4. Reduce shuffles and skew
   I avoid wide shuffles in the hot path. Where aggregations are required, I pre-aggregate early (map-side combine), and choose reasonable spark.sql.shuffle.partitions (near 2–3× total executor cores) instead of defaults. If keys are skewed, I add salting or partial aggregation to prevent single partitions from becoming hotspots.
5. Control state size with watermarks
   For any stateful operations (windows, dedup), I set watermarks that reflect realistic lateness so state doesn't grow without bound. Smaller state means shorter batch times and fewer GC pauses.
6. Make the sink fast and idempotent
   I write to Delta/Hudi/Iceberg with file sizing tuned for throughput (avoid too many tiny files). I use foreachBatch to batch upserts/writes and commit fewer, larger files. If compaction or clustering is heavy, I run it asynchronously in a separate job—don't block the streaming writer.
7. Use efficient compute settings
   I enable Kryo serialization, AQE (adaptive query execution), and keep executor heap/off-heap balanced to avoid GC storms. I minimize Python UDFs in the hot path; prefer Spark SQL or Scala/Java UDFs if needed. I cache only what truly benefits (for example, a small broadcast reference table).
8. Storage and checkpoint hygiene
   I place checkpointLocation on a fast, dedicated S3 prefix and ensure no lifecycle deletes it. I also avoid excessive small state files by keeping batch sizes stable. If I see long checkpoint commits, I reduce parallelism in the sink or increase batch interval slightly.

9. Measure, then iterate
   I watch these metrics: input rows per second, processed rows per second, batch duration vs trigger interval, state store memory, shuffle read/write times, and file commit times. I scale shards/workers until processedRowsPerSecond consistently meets or exceeds input.

In practice, the biggest wins usually come from more shards (true source parallelism), right-sizing workers, reducing shuffles, setting tight but safe watermarks, and making the sink write fewer, larger, idempotent commits.

21. How would you speed up a Glue job on billions of rows in S3 using partition pruning and pushdown filters?

I make the job read as little data as possible by organizing the S3 data correctly and pushing filters down to the file scan so Spark skips whole folders and whole row groups.

1. Store data in a partition-friendly layout
   I keep the big table in Parquet/ORC with Hive-style partitions such as dt=YYYY-MM-DD/region=IN/. I only partition by columns that are used very often in WHERE clauses (date, region, tenant). I avoid over-partitioning with high-cardinality fields like user_id.
2. Write queries that let Spark prune partitions
   Partition pruning only works if the filter uses the exact partition column name. So I always filter like where dt = '2025-08-01' and region = 'IN' before any transformations. I never wrap the partition column in functions (no to_date(dt)), and I push these filters as early as possible in the job.
3. Use Glue's pushdown predicate when reading from the Catalog
   If I use DynamicFrames, I pass a pushdown predicate so Glue only lists and reads needed partitions:

- from Catalog: create_dynamic_frame_from_catalog(..., push_down_predicate="dt >= '2025-08-01' and region in ('IN','US')")
- from S3 directly: create_dynamic_frame_from_options(..., pushDownPredicate="...")
  This prevents the driver from listing millions of partitions and speeds up planning.

4. Turn on file-level predicate pushdown and column pruning
   Parquet/ORC can skip whole row groups when filters are on regular columns too (not just partitions). I enable filter pushdown and select only required columns at the top of the script. In code, I first select() the exact columns I need, then filter() so Spark can push both projection and predicates to the file scan.
5. Reduce small-file overhead
   Billions of rows often means millions of small files. I compact data during writes (target 128–512 MB Parquet files) and run periodic compaction in a separate job. Fewer, larger files make pruning more effective and speed up reads.
6. Use partition indexes for very large catalogs
   On very heavily partitioned Glue tables, I add a partition index so GetPartitions calls return fast when I apply partition filters. This avoids slow plan times and occasional driver pressure.
7. Verify pruning actually happens
   I check the Spark plan (explain) or Spark UI to confirm the partition filter is applied and the input bytes are a small fraction of the table. If I still see huge input, I fix the filter expression or the table's partition keys.

Result: with correct partitions + pushdown predicates + column pruning, the job scans only the tiny slice of data for the requested dates/regions and finishes much faster.

22. If a Glue job fails with out-of-memory errors on joins, how would you choose the right worker type while balancing cost vs performance?

I first reduce the memory pressure with join strategy and data shaping. Then I choose a worker type that gives enough memory per executor for the unavoidable shuffle, without overspending.

1. Shrink the join before throwing hardware at it
   • Prefer broadcast hash join when one side is small enough. I broadcast() the small table so there's no massive shuffle.
   • If both sides are big, I pre-filter early, select only needed columns, and repartition both sides on the join key to get balanced partitions.
   • Fix skew: add salting for hot keys or use skew join hints so one key doesn't explode a single partition.
   • Use Bloom filters or semi-joins to reduce the large side before the main join.
   These steps often remove the OOM without changing workers.

2. Right-size partitions to the hardware
   Too few partitions → big partitions that don't fit in memory. Too many → overhead. I set spark.sql.shuffle.partitions to about 2–3× total executor cores, and I avoid creating huge rows (wide structs) by trimming columns early.

3. Pick a worker class based on memory need, not just CPU
   • Start with a mid tier (for example, G.2X) for sizable joins.
   • If OOM persists due to wide rows or very large shuffles, move to a higher-memory tier (G.4X or G.8X) so each executor has more heap for shuffle and join state.
   • Increase number of workers when I need more parallelism; move up a worker size when I need more memory per task. I do one change at a time and re-measure.

4. Balance cost using a simple decision rule
   • If the job is failing due to not enough memory per partition, upgrade worker size (G.2X → G.4X).
   • If it's simply slow but stable, add more workers of the same size to increase parallelism.
   • After the join, if the job becomes I/O bound, scale down to a cheaper tier and keep more workers. If it remains memory bound, keep the larger tier.

5. Operational safeguards
   • Avoid caching large DataFrames unless truly needed.
   • Prefer Spark SQL expressions over heavy UDFs in the hot path.
   • Monitor GC time, shuffle spill, and peak executor memory in the Spark UI. Spills are okay; repeated OOM means you still need either more memory per executor or smaller partitions.

6. Example upgrade path I use in practice
   • Fix join strategy and pruning → test.
   • If still OOM, increase shuffle partitions and repartition by join key → test.
   • If still OOM, move from a smaller to a larger worker type to increase memory per executor → test.
   • Only after stable, tune worker count to hit the desired runtime at the lowest total DPU-hours.

This approach keeps costs under control by first reducing the join's memory footprint, then matching the worker type to the true bottleneck (per-executor memory vs overall parallelism).

23. How would you stop a Glue job from reprocessing all history daily when only new S3 files need to go into Redshift?

I make the pipeline incremental end to end: only read new files from S3, load them into a Redshift staging table, and upsert into the target so duplicates never appear.

1. Read only new S3 objects
   • Enable Glue job bookmarks and keep a stable S3 path layout. Bookmarks track which objects were already processed, so each run picks only new ones.
   • Organize data in date-based partitions (for example, s3://bucket/table/dt=YYYY-MM-DD/). Put an early filter on the latest partitions so the job doesn't list old paths.
   • If producers sometimes rewrite files, I add a processed-keys ledger (for example, in DynamoDB) keyed by S3 object ETag + path; I skip keys already recorded to avoid re-reads when bookmarks can't help.

2. Validate idempotency before loading
   • Add a unique event_id or file_id + row_number to each record. Drop duplicates within the batch to protect Redshift from accidental replays.
   • Keep small batches: compact tiny source files upstream so the job handles fewer, larger files each day.

3. Stage, then merge in Redshift
   • COPY only the new batch into a narrow staging table with the same schema as the target.
   • Run a single MERGE statement from staging to the target using a business key or event_id. When matched, update; when not matched, insert. This gives upsert behavior and prevents duplicates if a file is retried.
   • After a successful MERGE, truncate the staging table and, if I keep a ledger, mark those S3 keys as processed.

4. Operational safeguards
   • Keep bookmarks enabled and never change the input path without intention.
   • Add a guardrail filter like dt >= today-1 to avoid accidental full scans.
   • Alert if the job reads more than an expected max number of files or bytes; that's usually a sign bookmarks or filters slipped.

Result: the job does not reread historical data, Redshift only sees the delta, and replays cannot create duplicates.

24. Your Glue costs doubled last month. How would you use Glue Flex jobs and tuning to cut costs while meeting SLAs?

I split workloads by urgency, move non-urgent ones to Flex capacity, and tune every job to reduce DPU-hours. The goal is fewer DPUs for less time, without missing deadlines.

1. Classify jobs by SLA
   • Critical (tight deadline, interactive): keep on standard Glue.
   • Non-urgent (backfills, daily rollups, file compaction, metadata maintenance, crawler-style discovery): move to Glue Flex where it's cheaper but may queue before starting. Schedule these earlier so they still finish before the SLA window.

2. Right-size each job
   • Pick the smallest worker type that keeps the job stable, then add workers for parallelism only if needed. If a job is memory bound, upgrade worker size; if it's just slow, add more of the same size.
   • Set a maximum concurrency so multiple heavy jobs don't spin up too many DPUs at once.
   • Set job timeout and sensible retry counts so failures don't burn hours.

3. Reduce scanned data and shuffles
   • Use partition pruning and pushdown predicates so each run reads only the required partitions and columns.
   • Compact small files upstream; target 128–512 MB Parquet files. Fewer files cut planning and listing time, and reduce DPU-hours.
   • Trim columns early, avoid wide rows, and eliminate unnecessary caches.
   • Repartition intelligently: set shuffle partitions near 2–3× total executor cores and fix key skew with salting or partial aggregates.

4. Make sinks efficient
   • When writing to S3, coalesce to reasonable file sizes to avoid tiny-file storms that trigger more work later.
   • For Delta/Hudi/Iceberg, run compaction/clustering as a low-priority Flex job on a schedule instead of during hot paths.

5. Use incremental patterns
   • Enable Glue bookmarks where applicable so batch jobs only touch new files.
   • For JDBC reads, use incremental predicates (for example, updated_at >= last_watermark) to avoid full pulls.

6. Optimize development and backfills
   • For dev/test, process a sampled subset and run on Flex.
   • For historical reprocessing, split the range and run multiple small Flex jobs in parallel windows rather than one oversized standard job.

7. Monitor and iterate
   • Track DPU-hours per job, input bytes, and runtime trends. Identify top cost drivers and fix those first.
   • Watch queue times for Flex jobs; if a Flex job starts threatening the SLA, promote just that job back to standard while keeping the rest on Flex.

With these steps—Flex for non-urgent work, smaller and smarter jobs, less data scanned, and leaner writes—costs drop materially while SLAs remain intact.

25. If Athena queries on Parquet output are still slow/expensive, what Glue optimizations (e.g., partitioning, bucketing, compression) would you apply?

I reduce how much data Athena has to scan and how many files it touches. I also write Parquet in a way that lets Athena skip more row groups.

1. Partition on the columns used most in WHERE clauses
   Typical choices are date (dt=YYYY-MM-DD) and one low-to-medium cardinality dimension like region or source. I avoid high-cardinality columns (like user_id) as partitions because they create too many small folders.
2. Push filters and select columns early in Glue
   I keep only the columns I need and write them back as Parquet. Athena then benefits from column pruning and predicate pushdown automatically.
3. Right-size files and row groups
   I target 256–512 MB per Parquet file and ~128 MB row groups. Fewer, larger files speed up listing and reduce per-file overhead. I run a small compaction job if I have many tiny files.
4. Use fast compression that keeps pages small
   For Parquet I prefer ZSTD (best scan speed vs size) or Snappy (very common and fast). I avoid GZIP for Parquet because it slows reads. I set the codec once in Glue so every new file is consistent.
5. Sort data by common filter columns before writing
   If I sort within each partition by a frequent filter (for example dt, then customer_id), Parquet min/max stats let Athena skip more row groups. This is a cheap win without changing the table layout.
6. Keep partition keys clean and simple
   Partition columns must be plain strings like dt='2025-08-01'. In queries I filter on the raw partition columns (no functions on them) so Athena prunes folders correctly.
7. Consider a second-level layout instead of bucketing
   Hive bucketing gives limited benefits in Athena and is easy to misconfigure. I prefer a two-level partition like dt/region and good sorting. If I absolutely need bucket-style joins, I ensure both tables use the same bucket key and count, but I only do this after measuring.
8. Avoid full table scans from Athena
   I add a table property for partition projection (when partitions are predictable like dates). Then Athena can discover partitions on the fly without crawling, and queries that filter dt won't waste time on the metastore. If I keep crawlers, I also add a Glue partition index for very large tables so partition lookups are fast.
9. Use CTAS/INSERT INTO to optimize after the fact
   If I inherit a messy table, I run an Athena CTAS (or a Glue job) to rewrite into an optimized layout: fewer columns, better partitions, sorted files, and ZSTD/Snappy compression. Then I point reports to the optimized table.

Result: Athena scans fewer bytes, opens fewer files, and finishes faster, which directly reduces cost.

26. How would you set up Crawlers + Data Catalog so Athena can query new raw S3 datasets automatically?

I give each dataset a stable S3 prefix, a scoped crawler with clear include/exclude rules, correct classifiers, and a schedule or event trigger. Then I manage permissions so Athena can read immediately.

1. Organize S3 and naming
   I put each dataset under its own prefix like s3://lake/raw/app1/ and use folder-style partitions if they exist (for example dt=YYYY-MM-DD/). Clear paths make crawling predictable.

2. Create a dedicated IAM role for the crawler
   The role can list/read only the dataset prefixes and write to the Data Catalog. Least privilege keeps things safe.

3. Choose or add the right classifier
   For CSV/JSON, I use built-in classifiers and set options (header, delimiter, quote). For nested JSON or custom logs, I add a JSON/Grok classifier so schemas are inferred correctly.

4. Configure the crawler to target a single database and table strategy
   I point the crawler at the dataset prefix and set it to "create a single schema for each S3 path." I also set table name and a table prefix if needed. I use exclude patterns to skip _temporary/, _SUCCESS, and archives.

5. Enable partition inference and schema updates
   I let the crawler detect partitions from folder names (like dt=...). I allow schema evolution with caution: add new columns, but do not drop or rename automatically. If I expect frequent schema drift, I version tables (table_v1, table_v2) and switch consumers deliberately.

6. Schedule or event-trigger the crawler
   If the dataset lands daily/hourly, I schedule the crawler shortly after data arrival. For near-real-time discovery, I trigger the crawler using S3 event → EventBridge → start crawler. This keeps the Catalog fresh without manual steps.

7. Add a Glue partition index for very large partitioned tables
   If the dataset grows to millions of partitions, I add a partition index so Athena and Glue can look up partitions quickly when queries filter on them.

8. Set table properties for better querying
   I verify the SerDe, input formats, and compression settings match the files. If the layout is date-predictable, I consider enabling partition projection in the table properties and then reduce the crawler frequency (or stop crawling partitions entirely).

9. Grant access via Lake Formation (or direct IAM if not using LF)
   I grant SELECT on the database/table to analyst roles so Athena can query immediately. This avoids "Access Denied" surprises.

10. Monitor and alert
    I enable CloudWatch metrics and an SNS alarm on crawler failures or schema changes. If the crawler detects unexpected schema deltas, I review before promoting to production tables.

With this setup, as soon as new files land in the raw S3 prefix, the crawler updates the Glue Data Catalog, and Athena can query the fresh partitions automatically with the correct schema.

27. What's the most efficient way to load terabytes of transformed data from Glue to Redshift, and how would you handle the small file problem?

I avoid JDBC row-by-row inserts and use Redshift COPY from S3. Glue writes the transformed data to S3 in large, well-sized Parquet files, then I run a COPY into a staging table and MERGE into the target.

1. Write to S3 in a Redshift-friendly layout
   • Parquet with Snappy or ZSTD compression.
   • Repartition/coalesce so files are big (256–1024 MB each) and the count roughly matches 2–4× the number of Redshift slices. This gives high parallelism without tiny-file overhead.
   • Partition by common filters like dt=YYYY-MM-DD to make reloads and backfills targeted.

2. Load using COPY into a staging table
   • COPY from S3 with IAM role; Parquet lets Redshift read columns efficiently.
   • Disable unnecessary auto encoding during heavy loads if it slows things, then ANALYZE after load.
   • Use a dedicated WLM/queue for loads so user queries don't compete.

3. Upsert safely into the target
   • Use a staging table that mirrors the target schema.
   • Run a single MERGE from staging to target on business keys; then TRUNCATE staging. This keeps idempotency and lets me rerun a failed batch without duplicates.

4. Handle the small file problem at the source
   • In Glue, before writing: df.repartition(N, col("some_key")) or just coalesce(N) to hit target file sizes.
   • If the upstream creates many tiny files, I run a nightly compaction Glue job that rewrites to big Parquet files.
   • I avoid writing one file per partition per task; I tune spark.sql.shuffle.partitions to a number that yields the desired file count.
   • I keep COPY batches pointed at a manifest listing only the files in the batch, so late-arriving or extra files won't sneak in.

5. Table design and maintenance
   • Choose sort/dist keys to fit query patterns, or enable Automatic Table Optimization.
   • After large loads, run ANALYZE (and VACUUM only if heavy deletes/updates).
   • Monitor load times, rows/sec, and skew; if one sort key causes hotspots, adjust.

This approach maximizes throughput (COPY from S3, columnar Parquet) and eliminates tiny-file drag by coalescing/compacting in Glue before loading.

28. How would you integrate Glue with Lake Formation so analysts can query only specific columns in Athena?

I put Glue's Data Catalog under Lake Formation governance and grant column-level permissions to analyst roles. Athena then enforces those permissions automatically.

1. Put the lake under Lake Formation control
   • Register the S3 data locations in Lake Formation.
   • Turn on Lake Formation permissions for the Glue Data Catalog and set data lake admins.
2. Create databases/tables in the Glue Catalog (governed)
   • Use crawlers or Glue jobs to create/update tables, but the permissions are managed in Lake Formation, not directly in Glue.
   • Ensure the crawler/job IAM roles have Lake Formation permissions to write metadata.
3. Define access using LF grants or LF-Tags
   • Simple case: grant SELECT on specific columns to an analyst IAM role or group (for example, allow columns id, date, amount but deny email, phone).
   • Scalable case: set up LF-Tags like pii=yes/no, domain=sales, and attach tags to columns. Grant analysts access via tag expressions (for example, pii=no). This automatically applies as new columns/tables are added.
4. Optional row-level filters and masks
   • If needed, add row filters (for example, region = 'IN') and column-level masking for sensitive columns. Athena will enforce both.
5. Hook up Athena to Lake Formation
   • Use an Athena workgroup with Lake Formation integration.
   • Grant the analyst role Lake Formation permissions (not just S3/IAM). S3 bucket policies should allow access "via Lake Formation" so LF can authorize.
6. Separate roles for engineers vs analysts
   • Data engineers' role gets broader table/column access (and CREATE/ALTER) for ETL.
   • Analysts' role only gets SELECT on approved columns, possibly via LF-Tags.
7. Audit and lifecycle
   • Enable Lake Formation audit logs to see who queried which columns.
   • When schemas evolve, new sensitive columns inherit tags/policies, so analysts don't see them until explicitly permitted.

With this setup, Glue still manages the schemas, but Lake Formation is the policy brain. Analysts in Athena only see and query the columns they're allowed to, with no code changes on their side.

29. How would you design a pipeline where Glue writes streaming data to S3, but analysts query it almost in real time via Athena?

I keep the writer fast and append-only, and I use a table format that Athena can read while data is still arriving. The target is a 1–5 minute end-to-dashboard delay.

1. Ingest and write from Glue Streaming
   I read the stream (Kafka/Kinesis), parse and clean the events, and write to S3 in Parquet using a table format that supports atomic snapshots and concurrent reads, like Apache Hudi or Apache Iceberg. I enable upserts with a unique event_id so retries do not create duplicates.

2. Partitioning and small-file control
   I partition by a time column that matches query patterns, for example dt=YYYY-MM-DD/hour=HH. I keep micro-batches short (for example every 1–2 minutes) but coalesce files so each file is at least 128–256 MB. I run background compaction/clustering (Hudi/Iceberg) on a schedule so the table stays efficient without blocking the stream.

3. Make new data instantly queryable without crawlers
   I avoid waiting for crawlers by using a governed table (Hudi/Iceberg) registered in the Glue Data Catalog once. Athena reads the latest table snapshot automatically; no new partitions need to be “added” by a crawler each minute. If I use plain Hive partitions, I turn on partition projection in the table properties so Athena can resolve recent hour partitions on the fly.

4. Event-time handling and idempotency
   I set a watermark on event_time and deduplicate by event_id to keep windows accurate when late events arrive. Because the sink is upsert-capable, late data can correct recent partitions and Athena sees the fix on the next snapshot.

5. Access and reliability
   I govern access in Lake Formation so analysts can query safely. I keep checkpointing on S3 stable so restarts resume without reprocessing. I alert if batch duration grows beyond the trigger interval or if the table compaction falls behind.

Result: Glue streams to an ACID lakehouse table on S3; Athena queries that table with near-real-time freshness and without crawler lag, while file sizes and compactions keep query costs low.

30. How would you join Redshift customer master data with S3 clickstream data in Glue, and what schema/performance/cost challenges might arise?

I avoid pulling large data over JDBC during every run. Instead, I keep a fresh copy of customer master in S3 and join it with clickstream there, using Glue for the compute.

Design

1. Move customer master to S3 regularly
   I export customer_dim from Redshift to S3 in Parquet (UNLOAD) on a schedule, or replicate it continuously with DMS into an Iceberg/Hudi table. That gives me a small, clean, columnar dimension on S3.
2. Join pattern in Glue
   If customer_dim is small to medium, I broadcast it for a fast stream-static or batch-static join with the clickstream fact. If it is large, I repartition both datasets by the join key and use a standard shuffle join with trimmed columns.
3. Write optimized outputs
   I write results as Parquet to S3 in a query-friendly layout (date/hour partitions, 256–512 MB files). If the output feeds Athena or Redshift again, I keep a stable schema and reasonable file sizes to control costs.

Key challenges and how I handle them
• Schema mismatches
Redshift types (for example, VARCHAR length, DECIMAL precision, TIMESTAMP TZ) may not match Spark/Parquet. I normalize types in Glue: cast IDs to consistent string/bigint, standardize timestamps to UTC, and sanitize column names so they are Hive/Athena friendly.

• Slowly changing dimensions
If customer_dim is SCD2, I resolve a point-in-time view before the join (effective_from/effective_to) so each click maps to the correct customer attributes at that time. I can precompute the current snapshot daily if “as-of-now” is enough.

• Performance bottlenecks
Pulling from Redshift via JDBC is slow and memory-heavy; that’s why I use UNLOAD/DMS to S3. For the join itself, I reduce shuffle size by selecting only needed columns, fixing skewed keys (salting or partial aggregation), and setting shuffle partitions to about 2–3× total executor cores. I choose worker size for memory-heavy joins and scale worker count for throughput.

• Small files
Clickstreams often generate many tiny files. I coalesce/repartition to target big Parquet files, and I run periodic compaction if upstream continues to create small files.

• Cost control
I minimize DPU-hours by pruning data by date or customer segments, broadcasting when possible, and avoiding unnecessary caches/UDFs. I schedule heavy compaction/jobs on Glue Flex when SLAs allow.

• Governance and PII
I use Lake Formation to mask or restrict PII columns from the joined output. Analysts only see permitted columns in Athena, while engineers can access the full dataset if needed.

Result: Redshift stays focused on serving queries; S3 holds a clean, query-optimized customer dimension; and Glue performs the join efficiently and cheaply, producing governed, analytics-ready data.

31. How would you design IAM roles so a Glue job reading/writing customer PII in S3 has only minimum permissions?

I apply strict least-privilege, scoped to just the data paths, keys, and services this single job needs. I separate duties between the job runtime, catalog/governance, and administration.

1. Create a dedicated Glue job role
   • Trust policy allows only the Glue service to assume it.
   • No wildcard admin policies attached; start from empty and add only what's required.

2. Scope S3 access to exact prefixes (or an access point)
   • Allow GetObject/List on the specific input prefix like s3://company-lake/pii/sourceA/ and PutObject on the specific output prefix like s3://company-lake/pii/curatedA/.
   • In the bucket policy, additionally restrict by aws:PrincipalArn to this role and by s3:prefix so it can't read other folders.
   • Prefer an S3 Access Point (or VPC Access Point) limited to those prefixes and to the VPC used by the job.

3. Enforce TLS and private access
   • Bucket policy denies if aws:SecureTransport is false.
   • If the job runs in a VPC, use an S3 Gateway/Interface VPC endpoint and bucket policy with aws:SourceVpce so traffic never leaves AWS backbone.

4. Lock down KMS usage (SSE-KMS everywhere)
   • Give the role kms:Decrypt/kms:Encrypt only on the specific CMK used for these datasets.
   • In the CMK key policy, allow only this role (and key admins) and add a kms:ViaService condition for s3.<region>.amazonaws.com and logs.<region>.amazonaws.com.
   • Deny wildcards like kms:* on all keys.

5. Minimal Glue permissions
   • glue:GetTable/GetPartition on only the required databases/tables. No Create/Update unless the job truly needs to write metadata.
   • If Lake Formation is enabled, grant data permissions there (column/table/row) and keep IAM policies metadata-only.

6. JDBC/warehouse access kept separate
   • If the job touches Redshift/JDBC, use Secrets Manager for credentials. Grant the role only secretsmanager:GetSecretValue for that one secret and nothing else in Secrets Manager.
   • Redshift IAM role for COPY/UNLOAD is separate and scoped to the same S3 prefixes and KMS key.

7. Guardrails and denies
   • Service Control Policies or bucket "explicit deny" to prevent any principal except approved roles from touching pii= true resources (use resource tags + aws:ResourceTag conditions).
   • CloudWatch Logs permissions limited to one log group.
   • No PassRole permission to avoid privilege escalation.

8. Observability and review
   • Enable CloudTrail and S3 server access logs.
   • Use IAM Access Analyzer/last accessed data to prune unused permissions after a week of running.

This gives a single Glue job role with the smallest possible set of actions on just the right S3 prefixes, the one KMS key, the necessary Catalog entries, and nothing else.

32. How would you ensure Glue pipelines encrypt data at rest and in transit (S3, Glue, Redshift) to meet compliance?

I enable encryption by default at every layer (S3, Glue, Redshift, logs, checkpoints) and force TLS on all network paths. I also prove it with policies that reject non-compliant writes.

1. S3 at rest
   • Turn on default bucket encryption with SSE-KMS using a customer-managed CMK.
   • Bucket policy requires x-amz-server-side-encryption = aws:kms and the specific KMS key ARN, and denies requests without it.
   • Enforce aws:SecureTransport true and optionally aws:SourceVpce to keep traffic on VPC endpoints.
   • Encrypt Glue checkpoints/temp paths with the same CMK.
2. Glue job/runtime
   • Create a Glue "security configuration" that enables: S3 encryption (SSE-KMS CMK), job bookmark encryption, CloudWatch Logs encryption, and Spark UI logs encryption. Attach this config to every job.
   • Run jobs in a private VPC with no public subnets; route S3 via VPC endpoints so traffic stays private.
   • For JDBC sources/targets, require SSL (set connection property useSSL/ssl=true and validate certs).
3. In transit everywhere
   • S3: HTTPS only via aws:SecureTransport in bucket policy.
   • Redshift: set require SSL in the parameter group and use jdbc:redshift:ssl=true; block non-SSL at the security group/NACL.
   • Kafka/Kinesis: use TLS endpoints; for Kafka, configure ssl.* properties.
   • Internal services (Athena/Glue Data Catalog): traffic is TLS by default; avoid custom endpoints that bypass TLS.
4. Redshift at rest
   • Enable cluster encryption with KMS (CMK).
   • Encrypt automated and manual snapshots and cross-region snapshot copies with KMS.
   • COPY/UNLOAD use IAM roles scoped to the encrypted S3 prefixes and CMK; require UNLOAD to specify ENCRYPTED if plain CSV is used (Parquet + SSE-KMS preferred).
5. Key management and access
   • CMK key policies restrict usage to specific roles; enable automatic key rotation.
   • Use kms:ViaService conditions to limit CMK use to S3, Logs, and Redshift where applicable.
   • No broad kms:* permissions; use grants for specific services if needed.
6. Governance and proof
   • Lake Formation governs data access; column masks and row filters apply after encryption so only authorized users see decrypted data through Athena/EMR.
   • CloudTrail logs are encrypted and retained; Config rules or Security Hub controls alert on buckets without SSE-KMS or policies missing SecureTransport.
   • Periodically run a "deny list" control: SCPs that deny s3:PutObject without KMS on accounts handling PII.
7. Edge cases
   • Temporary files: ensure Glue temp directories and Spark shuffle spill paths are in encrypted buckets.
   • Third-party sinks: use TLS endpoints and store creds in Secrets Manager, not in code; restrict the role to read only that secret.

With these settings, every byte written is SSE-KMS encrypted by default, every connection is over TLS, and guardrail policies enforce and audit the posture so the pipeline stays compliant.

33. How would you use Lake Formation to let analysts query Athena tables but hide sensitive PII columns?

I put the S3 data under Lake Formation governance and grant analysts only the columns they're allowed to see. Athena enforces these permissions automatically.

1. register the S3 locations in Lake Formation and turn on LF permissions for the Glue Data Catalog
2. create or crawl the tables in the Glue Catalog as governed tables
3. tag columns with lf-tags (for example, pii=yes/no, domain=sales) or select columns explicitly
4. grant analysts select only on approved columns (either via lf-tag expressions like pii=no, or by listing specific columns)
5. optionally add row filters and column masks if needed (for example, hash(email) or show last 4 of phone)
6. grant engineers broader access using separate roles; analysts' role gets only column-level select
7. ensure Athena workgroups use Lake Formation authorization and that S3 bucket policies allow access via Lake Formation
8. audit with Lake Formation access logs to confirm analysts never read masked/forbidden columns

Result: analysts can run the same SQL in Athena, but sensitive columns are hidden or masked by Lake Formation policies.

34. If a Glue job role has S3 permissions but still can't access a bucket, how would you troubleshoot IAM/trust issues?

I trace the request path end to end and look for explicit denies, missing KMS rights, or the wrong role being assumed.

1. confirm the runtime principal
   check the job details and CloudTrail to ensure the job is actually running as the intended iam role. print sts get-caller-identity from the job if needed. make sure only one job role is configured and that the trust policy allows glue.amazonaws.com to assume it.

2. look for explicit denies in bucket or org
   inspect the bucket policy for denies on aws:principalarn, aws:sourcevpce, aws:userid, aws:securetransport, or ip conditions. check organization scps and permission boundaries that may override allows.

3. verify kms permissions and key policy
   if the bucket enforces sse-kms with a specific cmk, the role must have kms:decrypt/encrypt and the key policy must allow the role (or a grant). also check kms:viaservice conditions match s3 in the right region.

4. check the exact s3 actions and resources
   ensure the role has s3:listbucket on the bucket arn (with appropriate s3:prefix conditions) and s3:getobject/s3:putobject on object arns. many failures are just missing listbucket on the bucket while getobject is granted on objects.

5. validate the path and access points
   confirm the code is hitting the correct bucket/prefix and, if using an s3 access point, that the access point policy allows this role and account. avoid typos and trailing slashes that break prefix conditions.

6. network and endpoint constraints
   if the bucket policy requires a specific vpc endpoint (aws:sourcevpce), make sure the glue job runs in that vpc and subnets, and dns to s3 is via the endpoint. enforce https by setting aws:securetransport true.

7. lake formation vs direct s3
   if reading through a governed table (athena/spark with lf), lack of lf permissions can look like s3 access denied. grant the table/column permissions in lake formation or bypass lf by reading s3 paths directly.

8. object ownership and acl quirks
   if objects are written by another account, enable bucket owner enforced object ownership or ensure the role has access via acls. mismatched ownership commonly blocks reads even when bucket policy looks fine.

9. reproduce and pinpoint
   use the iam policy simulator with the role, action, and resource. check cloudtrail events for the exact deny reason. try aws s3 ls s3://bucket/prefix from within the job to capture the error verbatim.

fixes usually involve correcting the assumed role or trust policy, adding listbucket, granting kms on the right key with a matching key policy, or removing an explicit deny in the bucket or scp.

35. How would you design Glue jobs to mask or delete personal data (PII) on request to meet GDPR requirements?

I separate two capabilities: ongoing masking for analytics, and subject-driven erasure (Right to be Forgotten). Both run as controlled, auditable Glue jobs with strict least-privilege roles and KMS encryption.

Ongoing masking for analytics
• Identify PII columns in the Catalog using tags (for example, pii=email, phone, address).
• In ingestion/curation Glue jobs, apply deterministic masking before data lands in the analytics zone:
– Tokenize IDs with a salted hash (for example, SHA-256(customer_id + org_salt)) so joins still work.
– Partially mask free-text (keep last 4 digits of phone, mask rest).
– Use format-preserving encryption for values that must look valid (for example, card PAN in PCI contexts).
– Drop sensitive columns entirely if not needed.
• Keep a separate, access-restricted "gold-pii" table only if a legitimate purpose exists; otherwise, store only masked columns.
• Enforce access with Lake Formation column masks so analysts never see raw PII.

Subject-driven erasure (DSAR)
• Maintain a subject ID map (for example, by customer_id or a stable subject_key) that lists all tables/partitions where the subject appears. Keep this map updated during ingestion.
• Build a parameterized "erasure" Glue job: input is the subject_key list from your DSAR system. The job:

1. Locates all occurrences across S3 lakehouse tables (Delta/Hudi/Iceberg) using partition filters to minimize scans.
2. For soft-delete, writes tombstones/upserts to set PII fields to null or masked equivalents immediately, so downstream queries stop seeing PII.
3. For hard-delete, issues deletes in the table format and triggers compaction/vacuum to physically remove data from files (required because S3 is immutable at file level).
   • For non-ACID raw zones (plain Parquet), rewrite affected partitions: read → filter out subject rows → overwrite partition atomically to a new prefix → swap pointer.
   • For Redshift or JDBC stores, run a transactional DELETE or UPDATE to null out PII, followed by VACUUM/ANALYZE if needed.

Operational controls and audit
• All jobs run with security configurations: SSE-KMS for S3, encrypted bookmarks/logs, TLS to sources/targets.
• Keep minimal audit logs showing request ID, tables touched, row counts removed or masked (never log raw PII).
• Implement retries with idempotency: if the job re-runs for the same subject, it produces the same state.
• Enforce SLA (for example, 30 days) by scheduling daily erasure windows and alerting on any missed subjects.
• Validate with data quality checks that no PII remains in analytics tables after the job.

Result: analytics never sees raw PII, and DSAR erasure reliably nulls or removes personal data across lake/warehouse with audit proof.

36. How would you use Glue DataBrew to clean raw customer data with nulls, bad dates, and extra spaces before storing curated data?

I create a repeatable DataBrew recipe that standardizes text, fixes dates, handles nulls, and outputs compact Parquet to a curated S3 prefix, then schedule it.

Setup
• Create a DataBrew dataset pointing to the raw S3 prefix (CSV/JSON/Parquet).
• Run a data profile to see null rates, distinct counts, outliers, and bad formats.
• Define a project with a recipe; lock column data types (strings, integers, timestamps in UTC).

Recipe steps (typical sequence)
• Trim and normalize text: Trim whitespace, collapse multiple spaces, standardize case (for example, names to proper case, emails to lowercase), remove control characters.
• Clean dates: Parse with explicit formats (for example, dd-MM-yyyy vs MM/dd/yyyy). Coerce invalid dates to null, enforce ranges (for example, birth_date between 1900-01-01 and today), and standardize to ISO 8601 UTC.
• Handle nulls:
– For required business keys, drop rows with null keys to a rejection file.
– For optional fields, impute with sensible defaults (median for numeric, mode for small categorical, or empty string) and add "_imputed" flags for transparency.
• Fix common anomalies: split combined fields (for example, "city, state"), trim leading zeros when appropriate, remove surrounding quotes, normalize phone numbers to E.164, and standardize country/region codes with lookup mappings.
• Deduplicate: define a key (for example, email_normalized + birth_date) and keep the most recent record by update_ts.
• Schema enforcement: cast columns to target types, rename to snake_case, and drop unused columns.
• Validations: add rules (for example, email must match regex, date not in future). Send failures to a quarantine S3 path with the reason.

Output and scheduling
• Set the job to write Parquet with Snappy or ZSTD, partitioned by dt=YYYY-MM-DD, and target 256–512 MB files (DataBrew job size settings influence file count).
• Store cleansed data to s3://lake/curated/customers/ and rejected records to s3://lake/quarantine/customers/.
• Create a schedule (hourly/daily) or trigger via EventBridge after raw drops complete.
• Optionally register the curated path in the Glue Data Catalog (crawler or manual schema) so Athena/Glue can query it.
• Monitor runs with CloudWatch metrics and review the profile reports over time to catch drift.

Result: a click-built, versioned recipe that consistently trims spaces, fixes dates, handles nulls, enforces schema, and lands compact, query-friendly Parquet ready for Athena or downstream Glue jobs.

37. How would you use Glue FindMatches to detect and merge duplicate customer records, and what parameters would you tune?

I treat this as a two-part problem: first, use Glue FindMatches to generate match groups of likely duplicates; second, apply clear survivorship rules to merge those groups into a single "golden" customer record.

Plan

1. Prepare input data
   Clean obvious issues first so the model focuses on real duplicates: trim spaces, standardize case, normalize phone to E.164, split full name into first/last, standardize addresses where possible, and keep a stable primary key column like source_record_id. Only keep columns that help match quality (name, email, phone, address, DOB, etc.).

2. Create a FindMatches ML Transform
   Point it to the prepared dataset in S3/Glue Catalog and set the primaryKeyColumnName to source_record_id. Start with a representative sample if the table is huge.

3. Generate and label a training set
   Use the "labeling set generation" to produce candidate pairs. Manually label a few hundred to a few thousand as "match" or "no match." Include tricky edge cases: same name different person, nicknames, typos, international formats.

4. Train and evaluate
   Train the transform and review precision, recall, and F1. If precision is too low (too many false matches), tune for stricter matching; if recall is too low (missed duplicates), tune for more permissive matching.

5. Tune key parameters
   • precisionRecallTradeoff: move toward 1.0 for higher precision (fewer false merges, may miss some dupes), toward 0.0 for higher recall (catch more dupes, risk more false merges).
   • accuracyCostTradeoff: increase if the business prefers fewer wrong merges even at the cost of missing some duplicates; decrease if catching all duplicates is more important.
   • enforceProvidedLabels: set true once you trust your labels so the model adheres to them strictly.
   • computeStatistics: enable to help interpret results during tuning.
   I iterate: retrain, inspect example clusters, adjust tradeoffs until the confusion matrix looks acceptable for the business risk tolerance.

6. Run the transform on full data
   The output adds a match_id (cluster/group id) and a confidence score per record. Now I have groups of records that likely represent the same customer.

7. Merge with survivorship rules in a Glue job
   For each match_id group:
   • Pick the survivorship "base" record (for example, the most recent by update_ts or the record from the system of record).
   • For each column, apply a rule: prefer non-null, prefer verified over unverified, pick longest string for address, pick most frequent value when consistent, or use a priority order of sources.
   • Keep a crosswalk table that maps all source_record_id values to the new golden_customer_id so downstream systems can trace lineage.
   • Write the golden table to S3 (Delta/Hudi/Iceberg) and optionally publish back keys to source systems if needed.

8. Operational guardrails
   Schedule periodic re-runs as new data lands, log merges for audit, and add a manual review queue for low-confidence groups before final merge. Keep PII locked down with Lake Formation and KMS.

Result: FindMatches gives me high-quality duplicate clusters with controllable precision/recall; simple, transparent survivorship rules create auditable, trusted golden customer records.

38. How would you apply Glue data quality checks to ensure fields like customer_id and order_date are always valid?

I define a Glue Data Quality ruleset, run it automatically in pipelines, fail fast on critical checks, and quarantine bad rows so the rest can proceed safely.

Design

1. Start with profiling and rule discovery
   Run a Glue Data Quality "recommendation" on recent partitions to auto-suggest checks. Then convert those into explicit rules and add a few business-specific ones.

2. Core rules for customer_id
   • Completeness: customer_id is not null.
   • Uniqueness: unique per table or per partition (decide the correct level).
   • Format: matches expected pattern (for example, prefix + digits) or length within a range.
   • Type: strictly string or bigint (no implicit casts).
   • Referential integrity: if there's a master dimension, customer_id must exist there.
   • Stability: sudden new distinct counts trigger a warning (guard against ID generator bugs).

3. Core rules for order_date
   • Completeness: not null.
   • Parseability: convertible to timestamp.
   • Range validity: not in the future beyond an allowed skew (for example, ≤ 24 hours ahead) and not older than a defined lower bound if that's a requirement.
   • Consistency: timezone normalized (UTC) and within business hours if relevant.
   • Freshness: max(order_date) must be within an expected freshness window for daily/hourly feeds.

4. Implement rules in Glue Data Quality
   Create a ruleset and attach it to the job step that validates each arriving partition before curation. Configure thresholds: critical rules fail the job; non-critical rules log warnings. Send metrics to CloudWatch and write a human-readable report to S3.

5. Handle failures without blocking everything
   Use two outputs:
   • Pass path → curated table.
   • Fail path → quarantine table with a reason column (for example, "order_date_in_future").
   Alert on any fail rate above a small percentage and open a ticket automatically if critical checks fail.

6. Keep checks fast and cheap
   Evaluate rules on the current partition or a recent sliding window, not the whole table. Push filters to scan only new data. Keep schemas tight so type checks are cheap.

7. Evolve the rules safely
   Version rulesets. When schema changes add columns, update the ruleset and run in “warn” mode for a short period before making them “fail” rules. Review DQ trends weekly to adjust limits (for example, acceptable null rates).

8. Downstream enforcement
   Block publishing to Redshift/BI unless the DQ step reports success for that batch. Stamp each curated file with dq_pass=true and a ruleset version so consumers can trust the data.

Result: every batch is automatically checked for non-null, unique, well-formed customer IDs and for sane, timely order dates; bad data is isolated, good data flows through, and dashboards remain trustworthy.

39. If FindMatches misses true duplicates or flags false ones, how would you improve its precision vs recall?

I iterate on three things: better training labels, cleaner matching features, and the model trade-off knobs. My goal is to align the model's mistakes with business risk (merging two different people is usually worse than leaving a duplicate).

1. Improve the labels
   I regenerate the labeling set and hand-label more pairs, especially edge cases (nicknames, typos, international formats). I balance "match" and "no match" examples and include hard negatives (same name, different person). More and better labels move both precision and recall up.

2. Engineer better matching signals
   Before training/inference, I normalize inputs so the model compares apples to apples: lowercase emails, E.164 phones, trimmed names, split full_name into first/last, standardized addresses, canonical country/state codes, and UTC dates. I add phonetic keys (Soundex/Metaphone) and nickname mappings so "Jon/John" match. Cleaner, richer features reduce both missed matches and false matches.

3. Tune FindMatches trade-offs
   • precisionRecallTradeoff: move toward 1.0 to be stricter (higher precision, lower recall) when false merges are costly; move toward 0.0 to catch more dupes when missing them is costly.
   • accuracyCostTradeoff: increase when you want fewer false positives even if recall drops; decrease when you can review low-confidence matches manually.
   • enforceProvidedLabels: set true once labels are trusted so the model honors them strongly.
   I retrain after each adjustment and check precision/recall/F1 and sample clusters.

4. Adjust the decision threshold and post-processing
   I set a higher minimum score for auto-merge and route mid-score pairs to a human review queue. I also choose cautious clustering logic (avoid chaining low-confidence edges) to prevent "match waterfalls."

5. Block to reduce bad comparisons
   I use blocking keys (for example, same email domain or same postal code) to avoid comparing records that cannot match. That reduces random pairings and improves precision.

6. Close the loop operationally
   I log false merges/splits reported by users, add them to the label set, and retrain periodically. Over time the model adapts to real data quirks.

Result: cleaner features + stronger labels + tuned trade-offs give a model that hits the precision/recall balance the business needs, with a manual review safety net for ambiguous cases.

40. How would you design a workflow so only datasets with >95% data quality score are published to the curated zone?

I gate publishing on a data quality step and branch the pipeline based on the computed score. Anything below 95% goes to quarantine with alerts; only passes reach curated.

1. Define rules and the score
   I create a Glue Data Quality ruleset for the dataset (completeness, uniqueness, valid formats, referential checks, date ranges). I enable the built-in score so each run outputs a numeric quality score and per-rule results.
2. Orchestrate with a gate
   I run the pipeline as: ingest → validate (DQ) → branch → curate/quarantine. This can be a single Glue job with logic, or a Glue Workflow/Step Functions state machine with a Choice state that reads the DQ result file.
3. Branching logic
   After the DQ run, the job reads the score (for example, from the Data Quality result JSON in S3).
   • If score ≥ 0.95: write to a temporary curated path, update the Glue Catalog atomically (or swap a table version/manifest), then mark the partition as published.
   • If score < 0.95: write the batch to a quarantine path with a “reason” column, emit an SNS/Slack alert, and open a ticket automatically.
4. Keep it idempotent and safe
   I always write to a staging location first and only “promote” to curated after the pass condition. That avoids partial publishes. I tag partitions with dq_pass=true and store the ruleset version alongside, so auditors and consumers know what standard was applied.
5. Fast and cheap checks
   I evaluate rules only on the new partition/window, push filters to avoid scanning history, and keep schemas strict so type checks are cheap. Critical rules are “fail” (block publish); informational rules are “warn” (publish but alert).
6. Governance and observability
   I publish the score and rule metrics to CloudWatch for dashboards and SLAs. I set alarms if scores trend down or if we see repeated quarantines. Lake Formation governs access so analysts never see quarantined data unless they have a special role.
7. Continuous improvement
   When schema evolves, I version the ruleset, run in warn-only for a few runs, then promote to fail once stable. I review top failure reasons monthly and fix upstream issues to keep pass rates high.

Result: only batches with a verified ≥95% quality score enter curated, promotion is atomic and auditable, and bad data is isolated with clear reasons and alerts.

41. If you have datasets in different formats (CSV, JSON, Parquet) across S3 buckets, how would you make them queryable in Athena while avoiding duplicate tables in Glue Catalog?

I set up a single, well-governed Glue Data Catalog with clear ownership and a "one dataset → one table" rule. Then I normalize formats for performance, and I use crawlers carefully so they update tables instead of creating new ones.

• Catalog design and ownership
Create one database per domain (for example, sales_raw, sales_curated). Define naming standards like app_dataset_zone (for example, web_clicks_raw, web_clicks_curated). Only a small set of roles can create tables; everyone else can only read. This prevents random duplicate tables.

• Raw zone vs curated zone
In raw, I register what actually lands (CSV/JSON/Parquet). In curated, I use Glue ETL/CTAS to convert everything to Parquet (or Iceberg/Hudi) with consistent schema and partitions so Athena is fast. Analysts primarily query curated.

• One crawler per dataset, not per person
Point the crawler to the exact S3 prefix of the dataset. Set "Create a single schema for each S3 path," provide a fixed table name, and use include/exclude patterns to avoid neighboring folders. Use Recrawl policy = Crawl new folders only. Set Update behavior = Update in database so it changes the existing table instead of making a new one with a suffix.

• Prevent duplicates by policy
Restrict CreateTable in Glue to a CI role or data engineering role. Everyone else submits a change request. This stops ad-hoc tables. For multi-account, use Resource Links to reference the same table instead of copying metadata.

• Handle different formats cleanly
If a dataset arrives in mixed formats, split raw tables by format (for example, web_events_json_raw, web_events_csv_raw) under the same database, then consolidate to a single curated Parquet table. Don't let the crawler infer two schemas into one table.

• Partition strategy and projection
Partition by dt=YYYY-MM-DD (and maybe region). For large, predictable partitions, enable Athena partition projection so you don't need the crawler to add daily partitions. That also avoids churn in the Catalog.

• Schema evolution discipline
Allow adds but block destructive changes. If a breaking change is needed, version the table (..._v2) instead of letting the crawler mutate columns in-place, which often triggers duplicates elsewhere.

Result: every dataset has exactly one authoritative table per zone, crawlers only update that table, and Athena users get stable, fast Parquet tables in curated.

42. Your Glue ETL job runs daily. How would you ensure only new S3 files are processed and avoid reprocessing old data?

I combine Glue job bookmarks with strict path conventions and idempotent writes. The job reads only files it hasn't seen, and even if a retry happens, the sink won't duplicate.

• Enable and respect Glue bookmarks
Turn on job bookmarks (job parameter job-bookmark-option=job-bookmark-enable). Read S3 using Glue's DynamicFrame APIs (from_catalog or from_options) so bookmarks track processed object keys. Keep the input prefix stable; moving files breaks bookmarks.

• Date-partitioned paths + early filters
Organize data as s3://bucket/dataset/dt=YYYY-MM-DD/ and filter on just today/yesterday before any joins. This reduces listing and makes bookmarks faster. Use pushDownPredicate (DynamicFrame) to limit to the intended partitions.

• Guardrail on first run vs subsequent runs
On day 1, let it read a bounded backfill range. After that, remove any "force" options and rely on bookmarks. Don't mix spark.read directly for sources you expect bookmarks to manage.

• Idempotent sink pattern
Write to a staging location, then MERGE into the target (Delta/Hudi/Iceberg on S3, or Redshift via COPY to staging + MERGE). Use a unique event_id or file_id + row_number so duplicates are naturally ignored on retries.

• Handle upstream rewrites and duplicates
If producers occasionally rewrite the same object, maintain a lightweight ledger (DynamoDB) keyed by S3 object key + ETag. Skip if already seen. This complements bookmarks for edge cases.

• Operational hygiene
Protect the bookmark/ checkpoint S3 prefix from lifecycle deletion. Alert if the job suddenly reads far more files than expected (a sign bookmarks or filters slipped). Keep one active job per dataset to avoid racing updates to the bookmark state.

Result: the daily run scans only the new partition's files, bookmarks ensure each object is processed once, and the sink's MERGE guarantees no duplicates even if a batch is retried.

43. You have a Glue workflow with three jobs (Clean → Aggregate → Load). How would you ensure the Load job doesn't run if the Aggregate job fails?

I tie job execution to upstream success and add a defensive data check so Load can't start unless Aggregate actually produced valid output.

• Workflow dependencies
In the Glue Workflow, I create two triggers:
– Trigger 1: starts Aggregate only when Clean has SUCCEEDED
– Trigger 2: starts Load only when Aggregate has SUCCEEDED
I do not use "Any state" triggers—only "Succeeded". I also set retries/timeouts on Clean/Aggregate so they either succeed within bounds or surface a failure that blocks Load.

• Defensive handoff (belt-and-suspenders)
Even with triggers, Load verifies that Aggregate produced what we expect:
– Check for a success marker (for example, _SUCCESS or a manifest file with row counts) in the Aggregate output prefix
– Optionally, validate record count > 0 and schema hash matches
If these checks fail, Load exits gracefully without writing, which protects downstream systems if someone runs Load manually by mistake.

• Orchestration alternatives for richer control
If I need branching/alerts, I wrap the three steps in Step Functions:
– Clean → Aggregate → Choice: if Aggregate status == SUCCEEDED → Load; else → notify/stop
This gives clear failure paths and notification hooks.

• Idempotency and recovery
Aggregate writes to a staging location and only promotes/renames on success. Load reads only the promoted location, so a partial Aggregate never triggers a bad Load even if someone forces it.

Result: Load can only start on an Aggregate SUCCEEDED event, and a quick runtime sanity check prevents accidental runs or empty loads.

44. If you are migrating an on-prem Hive-based Spark ETL pipeline to Glue, how would you map Hive Metastore metadata into Glue Data Catalog?

I inventory the Hive schema, decide which tables can be re-created by crawlers vs which need exact SerDe definitions, then programmatically create matching databases/tables in Glue so Spark/Athena see identical metadata.

• Inventory and classify
Export a list of Hive databases, tables, locations, input/output formats, SerDes, partition columns, and table properties. Classify:
– Standard formats (Parquet/ORC/CSV/JSON with common SerDes) → safe to recreate via crawler or scripted creates
– Custom/legacy SerDes or complex row formats → script exact definitions (don't rely on inference)

• Re-create databases and tables in Glue (programmatic path)
Use a small migration script (boto3 Glue client) that loops through the inventory and calls:
– create_database(name, description, locationUri)
– create_table(...) with StorageDescriptor (columns, SerDeInfo, InputFormat, OutputFormat, Location), PartitionKeys, TableType, Parameters
This preserves schema, SerDe, file formats, and table properties 1:1. For external tables, point to the same S3/HDFS locations (after data is moved to S3).

• Crawlers for simple cases
For plain Parquet/ORC datasets laid out as Hive-style partitions, point a crawler at each dataset prefix with:
– Include/exclude patterns so it doesn't create duplicate tables
– Recrawl = "new folders only"
– Update behavior = "update in database"
This is faster than scripting for straightforward tables.

• Partitions at scale
If tables have millions of partitions, I avoid importing every partition. Instead I enable partition projection in Glue/Athena (define ranges/patterns for dt, region, etc.). For moderate partition counts, I create partitions via batch CreatePartition calls.

• Normalize and fix incompatibilities
– Make column names Glue/Athena-friendly (no spaces, backticks, or uppercase-only semantics)
– Align types (for example, Hive TIMESTAMP with/without TZ vs Spark/Athena expectations)
– Ensure table locations use s3:// paths; convert HDFS paths during data migration

• Permissions and governance
Put the new Catalog under Lake Formation if used. Grant SELECT (and column-level restrictions if needed) to consumer principals. Limit Create/UpdateTable to data engineering roles to prevent drift.

• Validate before cutover
For each migrated table:
– Run SELECT count(*), sample queries in Athena and a small Spark job using the Glue Catalog
– Compare schema and row counts to Hive source
– Only then switch producers/consumers to use the Glue Catalog name

• Spark/ETL pipeline switch
In Spark configs, set the Hive metastore to use the Glue Data Catalog. The ETL code continues to use the same table names; only the metastore backend changes.

• Rollback and documentation
Keep the mapping file (hive_db.table → glue_db.table) and a rollback plan. Document any tables re-modeled in curated form.

Result: the Glue Data Catalog mirrors the Hive Metastore accurately where needed, crawlers handle simple cases, partition projection prevents metadata blowups, and Spark/Athena can read the data immediately with minimal code change.

45. How would you let analysts query raw, curated, and aggregated S3 data in Athena without them manually creating schemas?

I centralize all schemas in Glue Data Catalog and automate their creation/updates, so analysts only need SELECT permissions in Athena.

Design

1. One Glue Data Catalog as the single source of truth. Create databases like raw, curated, and analytics. Enforce naming standards so datasets are easy to find.
2. Automate table creation:
   • Raw: use one crawler per dataset, scoped to its exact S3 prefix with include/exclude patterns. Set recrawl policy to "new folders only" and update behavior to "update in database" so it updates one table instead of creating duplicates.
   • Curated/Aggregated: have Glue ETL jobs write Parquet/Iceberg and register or update tables programmatically at the end of each job (Glue APIs or Spark/Iceberg integration auto-commits table metadata).
3. Avoid waiting on crawlers for predictable partitions by enabling partition projection. Analysts can query new date/hour partitions immediately without manual ALTER PARTITION or crawler runs.
4. Give analysts access via Lake Formation. Grant SELECT on the specific databases/tables (and column-level filters if needed). No one outside data engineering needs CreateTable privileges.
5. Publish a data dictionary: lightweight Glue table properties and a Confluence/README page per dataset (owner, refresh cadence, sample queries). Analysts discover and query instantly from Athena's table list.
6. Keep performance consistent: in curated/analytics, store columnar (Parquet or Iceberg), partition by dt and another low-cardinality dimension, and target 256–512 MB file sizes so Athena scans are fast and cheap.

Result: data engineers own schema automation; analysts open Athena, see governed tables across zones, and query without creating or maintaining schemas themselves.

46. If a source keeps adding new optional fields, how would you manage schema changes in Glue so Athena/Redshift queries don't break?

I design for backward compatibility: treat new fields as nullable, keep stable views for consumers, and automate safe metadata updates.

Athena/S3 side

1. Store data in Parquet or an ACID table format (Iceberg/Hudi/Delta) that supports add-column evolution. New optional columns are appended as nullable; old queries keep working.
2. Control schema updates:
   • If using crawlers, set update behavior to "add new columns only" so it never drops/renames.
   • Prefer programmatic schema updates (Glue API / Iceberg ALTER TABLE ADD COLUMN) at the end of the ETL that introduces the field, for deterministic changes.
3. Shield consumers with views. Maintain consumer-facing Athena views that select only the stable columns; update the view when stakeholders are ready to adopt new fields. This prevents BI breakage.
4. Validate and default. In Glue jobs, add withColumn for new fields and default to null or a safe value when absent. Add data quality rules to alert on unexpected type changes.

Redshift side
5) Land raw to S3 first. Keep an external table (Spectrum/Iceberg) over the lake to absorb schema drift without impacting Redshift immediately.
6) Stage before target. Load daily deltas into a Redshift staging table. When optional fields are adopted, run ALTER TABLE ADD COLUMN (nullable) on the target and update the MERGE to populate them. Existing queries keep working because added columns are nullable and not referenced.
7) Consider semi-structured for fast-moving schemas. If new fields appear frequently, land them in a SUPER column (JSON) in staging and gradually materialize important ones into typed columns. Downstream SQL can read SUPER with PartiQL during the transition.

Governance and safety
8) Version schemas. Track a schema_version in table properties and in the curated dataset. Communicate changes and keep a simple changelog.
9) Prevent destructive changes. Disallow drops/renames via policy; if a breaking change is required, publish a v2 table and deprecate v1 on a schedule.
10) Monitor. Use Glue data quality checks to detect unexpected type shifts or sudden null spikes on new fields, and alert before consumers feel it.

Result: new optional fields land safely as nullable, Catalog metadata updates automatically or via code, consumer views remain stable, and Redshift adoption happens on your terms without breaking existing queries.

47. Your Glue Data Catalog has thousands of tables, with duplicates from different teams. How would you organize it to maintain a single source of truth?

I apply a “data product” model with clear ownership, strict naming, and controlled creation so only one authoritative table exists per dataset and zone. Everything else either references it or gets archived.

1. Databases by zone and domain
   Create databases like sales_raw, sales_curated, sales_analytics (one domain per set). This prevents teams from spraying tables into random places.

2. One dataset → one table per zone (authoritative)
   Adopt a naming convention: domain_dataset_zone (for example, web_clicks_curated). Publish a short owner/SLAs in table properties (owner, contact, refresh_cadence, source_system). Declare this table the only source of truth for that dataset/zone.

3. Lock down who can create/alter tables
   Restrict Glue CreateTable/UpdateTable to a small “catalog-admin” role or CI pipelines. Everyone else has read-only. All table creation goes through code review (IaC: CloudFormation/Terraform or a small boto3 script). This alone stops most duplicates.

4. Use Resource Links and cross-account sharing
   When multiple accounts need the same table, share from a central producer account via Lake Formation and create Resource Links in consumer accounts instead of copying schemas. Consumers see the same table, not a fork.

5. Crawler discipline (or avoid crawlers)
   If you must use crawlers, register one crawler per dataset prefix with fixed table name, include/exclude patterns, Recrawl = “new folders only”, Update = “update in database”. For predictable partitions, prefer partition projection and disable frequent crawls. Never point multiple crawlers at the same prefix.

6. Catalog hygiene and de-dup process
   Run a weekly “catalog linter” job that flags duplicates by similar names/locations and empty/never-used tables (based on CloudTrail/Athena query logs). For duplicates: pick a canonical table, migrate consumers, then tag others as deprecated=true and set an auto-delete date.

7. Versioning policy for breaking changes
   When schema breaks, publish dataset_v2 and mark v1 deprecated with a sunset date. Don’t mutate the canonical table in-place in ways that break consumers.

8. Documentation and discovery
   Maintain a simple data dictionary (could be a Glue table property + Confluence page). Publish sample queries and owners. Analysts discover the canonical table quickly and stop creating their own.

Result: a small set of governed, owner-backed canonical tables per dataset/zone; tight create/alter controls; duplicates detected and retired; consumers use shared links instead of copying schemas.

48. How would you use Glue + Lake Formation to give analysts access only to curated data, while engineers can access both raw and curated?

I put the Catalog under Lake Formation governance, separate users into analyst and engineer groups, and grant permissions at database/table/column level so analysts never touch raw.

1. Separate databases by zone
   Keep raw and curated in different Glue databases (for example, retail_raw, retail_curated). Register their S3 locations in Lake Formation.

2. Define groups and roles
   Create two IAM roles (or SSO groups): analysts and engineers. Engineers get broader rights; analysts are read-only on curated.

3. Apply Lake Formation permissions
   Grant analysts SELECT on the curated databases/tables (and only specific columns if needed using column-level grants or LF-Tags like pii=no). Do not grant them anything on raw. Grant engineers SELECT on both raw and curated, plus optional ALTER/CREATE if they manage pipelines.

4. Enforce S3 via LF
   Use bucket policies that allow access "via Lake Formation" and deny direct S3 reads to raw prefixes for analyst roles. This ensures all access checks go through LF policies.

5. Optional row filters and masks
   For curated, add row-level filters (for example, region-based) and column masks for semi-sensitive fields. Engineers can see full data; analysts see filtered/masked views.

6. Register curated tables once, avoid ad-hoc schemas
   Glue jobs that produce curated data register/update tables programmatically. Analysts never create tables; they just query. For partitioned tables, enable partition projection so analysts can query new partitions instantly without crawler runs.

7. Auditing and guardrails
   Enable Lake Formation and CloudTrail audit logs. Create a periodic report of who accessed what. Add an SCP that blocks CreateTable in Glue for analyst roles to prevent shadow tables.

8. Onboarding checklist
   When a new curated dataset is published: set owner/contact, tag columns with pii=yes/no, grant analysts via LF-Tag expression (pii=no), test access with an analyst role, and publish sample Athena queries.

Result: analysts can query only curated, non-sensitive columns through Athena; engineers can work across raw and curated. Policies are enforced centrally by Lake Formation, and direct S3 access to raw is blocked for analysts.

49. If you migrate from Hive Metastore (with custom SerDes and partitions) to Glue Catalog, how would you make it work smoothly with Athena and Redshift?

I start by inventorying every Hive table (formats, SerDes, partition keys, table properties), then recreate metadata in Glue in a way that Athena and Redshift Spectrum both understand. Where Athena can't use a custom SerDe, I convert the data to Parquet and register a new table so queries keep working.

Plan

1. Catalog inventory and classification
   Export Hive DDL for all databases. Classify tables:
   • Standard (Parquet/ORC/CSV/JSON with common SerDes) → safe to recreate directly.
   • Custom/legacy SerDes or odd storage layouts → plan to rewrite data to Parquet or keep Spark-only access.
2. Programmatic migration of metadata
   Write a small script (boto3/Glue API) that, for each table, calls create_database and create_table with the original columns, partition keys, StorageDescriptor (InputFormat/OutputFormat/SerDeInfo), table properties, and S3 locations. Keep names and database structure identical where possible to minimize code changes.
3. Handle custom SerDes explicitly
   Athena supports a limited set of SerDes. If a table uses a custom SerDe that Athena can't read, run a one-time (or rolling) Glue/Spark job to rewrite those datasets to Parquet in a curated prefix with the same partitioning. Register a new Athena-facing table over Parquet while optionally keeping a Spark-only table over the legacy layout for backfill.
4. Partitions at scale
   If there are many partitions, avoid importing each one. Enable partition projection on Athena/Glue for predictable keys (for example, dt, region). For moderate counts, batch-create partitions with CreatePartition. Add a Glue partition index if you keep millions of partitions.
5. Schema normalization for cross-engine compatibility
   Standardize types and names: lowercase column names, replace spaces/special chars with underscores, align DECIMAL precision/scale, timestamps in UTC. Add table properties for classification and owner/SLAs.
6. Permissions and governance
   Put the new Catalog under Lake Formation. Grant SELECT to consumers and manage column-level controls there. Update S3 bucket policies to allow access via LF, not direct IAM, to keep governance centralized.
7. Redshift integration
   Create an external schema in Redshift that points to the Glue Data Catalog. For heavy/critical datasets, consider CTAS into Redshift-managed tables or use Spectrum against Parquet/Iceberg directly. Ensure Parquet compression (Snappy/ZSTD) and sane file sizes so Spectrum scans are efficient.
8. Validation and cutover
   For each migrated table, run sample Athena queries (count, min/max) and a small Spark job reading via the Glue Catalog. Compare results to Hive. Cut over consumers in batches. Keep a rollback plan and tag old Hive entries as deprecated during transition.

Result: Glue mirrors Hive for supported formats, non-supported SerDes are converted to Parquet for Athena/Spectrum, partitions remain queryable at scale, and both Athena and Redshift read consistently through the Glue Catalog.

50. How would you configure a Glue Crawler to handle new fields in JSON data without breaking Athena queries?

I let the crawler add new columns safely, never drop/rename, and I shield consumers with stable views. When evolution is fast, I move the data to Parquet with explicit schemas.

Steps

1. Use a JSON-aware setup
   Create a dataset-specific crawler targeting only the dataset prefix. Use the built-in JSON classifier or a custom JSONPath classifier if the payload nests deeply. Keep include/exclude rules tight so it doesn't pick up unrelated files.
2. Safe schema change policy
   In the crawler's configuration:
   • Schema change policy → "Update in database" and "Add new columns" only.
   • Do not enable automatic delete/rename; set delete behavior to log only. This prevents the crawler from breaking existing schemas by dropping/renaming columns.
3. Recrawl discipline
   Set Recrawl policy = "Crawl new folders only." For hourly/daily partitions, this avoids re-inferring old data and keeps evolution incremental. Combine with partitioned paths (for example, dt=YYYY-MM-DD/hour=HH).
4. Keep queries stable with views
   Expose an Athena view that selects the stable set of columns used by dashboards. When the crawler adds a new optional field, the base table gains a nullable column, but the view shields consumers until they opt in to the new field.
5. Control types and defaults in ETL
   If types drift (for example, id flips between string and number), add a lightweight Glue job that normalizes JSON before landing to the raw/curated table (cast to string, coerce bad values to null). Write Parquet in curated to lock types.
6. Prefer Parquet in curated
   For fast-moving JSON, land raw as JSON, but run a Glue job to write curated Parquet with explicit schema (nullable new fields). Register the curated table once; Athena reads it without re-inference. This avoids surprises from schema-on-read.
7. Testing and promotion
   Use a dev crawler on a sample prefix first. If the new column set looks good, run the prod crawler. Alternatively, register the additional columns programmatically (Glue API) at the end of the ETL that first sees them, so metadata changes are deterministic.
8. Guardrails and monitoring
   Add Athena/Config checks that alert if the crawler attempts a destructive change or if a key column's type changes. Keep crawler IAM minimal, and schedule after data arrival to avoid half-written files.

Result: new JSON fields appear as nullable columns without breaking existing Athena queries, analysts continue to use stable views, and curated Parquet provides consistent types and better performance as the schema evolves.