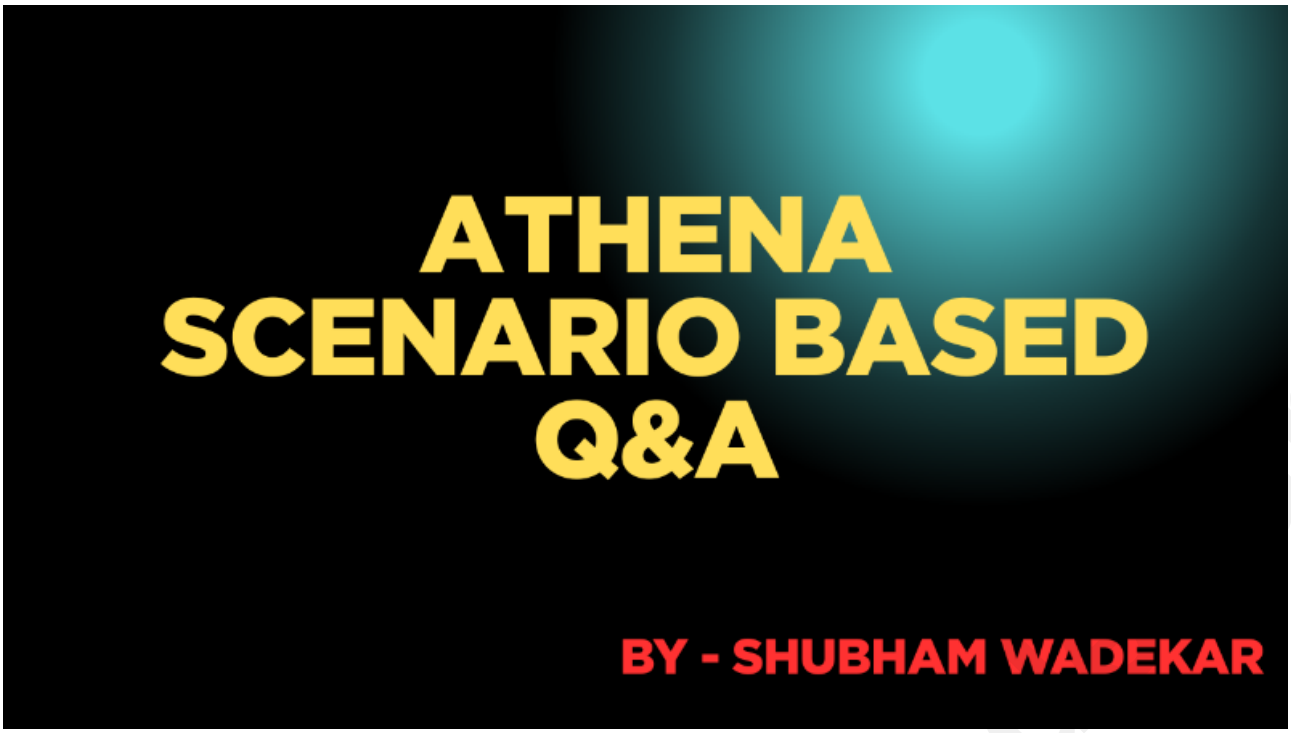

1. Your Athena queries on a 2TB dataset stored as CSV in S3 are slow and costly. What steps would you take to improve query performance while reducing cost?

I would change the storage format and layout so Athena scans far fewer bytes and does less work per query.

1. convert CSV to a columnar format
   Store data as Parquet (or ORC). Columnar formats let Athena read only the columns you SELECT and use min/max statistics to skip row groups. Do this with a CTAS query or a small Glue job that reads CSV and writes Parquet with compression (Snappy/ZSTD).

2. add sensible partitions that match filters
   Organize S3 as dt=YYYY-MM-DD (and optionally region or customer_tier). Athena will prune entire folders when you filter by those fields, so it won't scan irrelevant data.

3. avoid tiny files; target big files
   Aim for 128–512 MB per Parquet file. Use CTAS/UNLOAD or a compaction job to rewrite small CSVs into fewer large Parquet files. Fewer, larger files reduce S3 request overhead and speed up queries.

4. register a clean schema once
   Create a Glue Catalog table over the curated Parquet path with the correct data types (no VARCHARS for everything). Correct types avoid runtime casts and speed up filters/joins.

5. enable partition projection (optional but helpful)
   If you add a partition per day/hour, turn on partition projection so you don't need a crawler to add partitions. Queries are immediately aware of new dates and still prune.

6. prune columns in SQL
   Always SELECT only the columns you need. Columnar + column pruning drastically cut scanned bytes.

7. use compressed, splittable files
   Keep Parquet with Snappy/ZSTD. Don't gzip CSV for analytics; gzip CSV is not splittable and forces full-file reads.

8. create workload-optimized views
   Expose analysts to views that already filter on dt and cast types correctly. This reduces accidental full-table scans.

9. housekeeping
   Put old raw CSV in a /raw/ prefix and keep analytics against /curated/ Parquet. Set S3 lifecycle rules to move cold data to cheaper tiers. Monitor "data scanned" in Athena workgroup metrics and keep an eye on average file size.

Result: moving from CSV to partitioned Parquet with large files and proper typing typically cuts scanned data by 10–100×, making queries both faster and much cheaper.

2. Your team is running daily partitioned queries on an S3 dataset, but query scans are still large due to too many small files. How would you restructure the data to fix this?

I would compact small files into fewer large Parquet files per partition and adjust how we write data so new files are the right size from the start.

1. compact existing partitions
   For each dt=YYYY-MM-DD partition, read all small files and rewrite as 128–512 MB Parquet files. You can do this with Athena CTAS (writing to a new location then swapping), or a Glue/EMR job that coalesces and writes larger outputs. After validation, repoint the table or update the partition location.

2. fix the way new data lands
   Increase buffering at the writer (Firehose/Glue/Spark) so it writes larger files. Avoid per-record PUTs. If you're using Spark/Glue, set the number of output files per partition (coalesce/repartition) to hit the size target.

3. keep partitioning coarse enough
   Daily (dt) is usually sufficient. Avoid minute-level partitions unless you truly need them; too many tiny partitions cause both file and metadata overhead.

4. consider a table format that maintains layout
   If you often upsert or append small batches, store the table as Iceberg/Hudi/Delta and schedule regular compaction/clustering jobs. The streaming job can write small files, and the maintenance job rewrites them into big ones.

5. enforce file size checks
   Add a small monitoring job that reports average file size and file count per partition. Alert if a partition's average file size drops below, say, 64–128 MB so you can compact quickly.

6. keep Glue Catalog simple
   One authoritative table over the curated path. If you switched locations during compaction, update the partition location; or recreate the table with a CTAS so metadata matches the new layout.

Result: with compaction and better write buffering, each partition has a handful of large Parquet files. Athena opens fewer files, prunes row groups efficiently, and your daily queries scan far fewer bytes and finish much faster.

3. A new data source is added to your S3 data lake with schema changes every month. Queries in Athena start failing due to schema mismatch. How would you handle schema evolution using Glue?

I make schema evolution explicit and safe. I stop relying on Athena to "guess" the schema and instead control it with Glue, columnar formats, and a few guardrails so new optional fields never break existing queries.

1. land data in a format that tolerates evolution
   Convert new drops to Parquet (or Iceberg/Hudi/Delta). Columnar formats allow adding nullable columns without breaking readers. If the source is CSV/JSON, I run a small Glue job to normalize types and write Parquet to a curated prefix.

2. add columns, never drop/rename in place
   In Glue, I let evolution be additive. New source fields are added as nullable. I avoid destructive changes. If a breaking change is required, I publish a v2 table and deprecate v1 on a schedule.

3. control how Glue updates schemas
   If I use a crawler, I scope it to the dataset prefix and set the schema change policy to "update in database" and "add new columns only." I avoid letting the crawler rename or delete columns. For predictable drops, I update schemas programmatically with Glue APIs at the end of the ETL so changes are deterministic.

4. shield analysts with stable views
   I expose an Athena view that selects the stable columns used by dashboards. When a new column appears, the base table gets it as nullable, but the view stays unchanged, so queries don't fail. Analysts can opt in to new fields when ready.

5. validate schema before publish
   At ingest, I run a lightweight Glue job or Lambda that validates the input against an expected schema (Glue Schema Registry or JSON schema). If it detects incompatible changes (type flip, required field missing), it quarantines the batch and alerts, instead of letting bad files hit curated.

6. partition and table design that doesn't fight evolution
   I partition by dt=YYYY-MM-DD (and optionally region). I keep one table per dataset version. If the monthly changes keep piling up and get messy, I cut a new table version yearly and backfill a small window so each table stays clean.

7. handle type drift explicitly
   If a field flips type (string↔int), I normalize in ETL and keep a single canonical type in the curated table (for example, cast to string). I store the raw value in a separate "raw_json" or "_raw" column only if needed for audits.

8. test and automate
   I treat schema as code: unit tests for mappings, CI that runs a sample CTAS against a staging bucket, and a controlled Glue update (or ALTER TABLE ADD COLUMNS) during deployment.

Result: new monthly fields land as nullable in curated Parquet, Glue registers them safely, views keep existing queries stable, and only intentional schema changes reach Athena.

4. Your analysts complain that Athena queries are returning incorrect results due to outdated metadata. How would you ensure table definitions in Glue remain accurate?

I keep Glue in sync with what's in S3 and remove the need for frequent manual metadata updates. I standardize how partitions and schemas are updated and add checks so stale metadata can't sneak through.

1. eliminate partition drift with projection
   If partitions are predictable (date, hour, region), I enable partition projection in the Glue table. Athena computes partitions from patterns, so queries "see" new dates immediately without waiting for a crawler or ALTER PARTITION.

2. schedule or trigger metadata refresh where needed
   For datasets that cannot use projection, I run a scoped crawler after each successful load, or I call Glue APIs (CreatePartition/BatchCreatePartition) from the ETL at the end of the job. I never rely on ad-hoc manual runs.

3. pin schemas for curated data
   For curated Parquet/Iceberg tables, I manage schemas programmatically in the ETL (ALTER TABLE ADD COLUMNS or Iceberg ALTER TABLE). I avoid crawlers changing curated schemas unexpectedly. Crawlers are fine for raw discovery; code owns curated.

4. fix file layout and SerDe mismatches
   I make sure the table's SerDe and types match the actual files. I prefer Parquet with correct column types. For CSV, I standardize delimiters, headers, and quote rules, or I convert to Parquet to remove order/typing ambiguity that causes "incorrect results."

5. detect and block stale metadata
   I add a quick data contract check in the pipeline: compare a sample of new files to the Glue schema. If a required column is missing or a type flipped, the pipeline fails before publishing. I also run a daily metadata audit that reads a small sample from S3 with Athena's DESCRIBE/SELECT and compares to the Glue table schema; if mismatched, it opens a ticket automatically.

6. atomic publish pattern
   Writers land to a staging prefix, validate schema and partition entries, then atomically update partition locations (or Iceberg snapshot). Analysts only query the "promoted" path, so they never see half-published data or mismatched metadata.

7. documentation and governance
   Each table has an owner and a short README in table properties (owner, refresh cadence, partition keys). Analysts know which table is authoritative, and we archive deprecated ones to avoid forks.

8. monitoring
   I watch Athena workgroup metrics for sudden jumps in bytes scanned or error rates, which often indicate metadata drift. I alert on crawler failures and on Glue partition counts that stop increasing when new data should appear.

Result: partition projection or automated partition updates keep partitions fresh, schema changes are applied intentionally by ETL, and small validation checks catch drift early—so Athena reads match the actual data and analysts stop seeing wrong results.

5. You receive deeply nested JSON events in S3 and need to flatten arrays for reporting. How would you design Athena queries to extract this data efficiently?

I avoid scanning raw JSON for every dashboard. I first map the JSON to proper array/struct types, use UNNEST to explode arrays, and then materialize a flattened Parquet table that reports can query fast.

1. Register or read JSON with types, not just strings
   If the bucket has JSON Lines, I create a table that maps big objects to ROW/ARRAY types (or keep a simple table with one string column json_line and cast paths when selecting). Typed columns let Athena push down projections and avoid heavy json_extract on every column.

2. Extract and flatten with UNNEST (explode)
   For an array of objects like items, I use CROSS JOIN UNNEST to get one row per array element. If my base column is raw JSON, I cast the path into an array of rows first. Example pattern:

- Select minimal parent columns
- CROSS JOIN UNNEST(parent.items) AS t(item)
- Select item.field1, item.field2, etc.
  If I need element position (for ordering), I add WITH ORDINALITY.

3. Only read what I need
   I project only required columns and filter early (date/region) to prune partitions and reduce scanned bytes.

4. Materialize as Parquet for speed
   After validating the flattening logic, I run a CTAS that writes the flattened result into a curated location in Parquet with Snappy/ZSTD, partitioned by dt=YYYY-MM-DD (and region/tenant if useful). Reports query this Parquet table or an Athena view over it, not the raw JSON. This cuts cost and latency by 10–100×.

5. Handle nested-of-nested safely
   If arrays are nested (e.g., orders.items.discounts), I UNNEST step by step: first UNNEST items, then UNNEST discounts from that derived alias. Keep names distinct to avoid confusion.

6. Keep file sizes healthy
   The CTAS (or a small Glue job) should target 128–512 MB Parquet files per partition. Fewer, larger files make Athena faster.

7. Validate and iterate
   I sample a few raw rows and compare counts after UNNEST to ensure I'm not accidentally multiplying rows via unintended joins. I also add simple DISTINCT checks if the source may duplicate events.

Result: the pipeline flattens arrays once into partitioned Parquet using UNNEST, and day-to-day reports read a small, columnar surface instead of re-parsing deep JSON every time.

6. Some records in your JSON contain optional fields, causing NULL results in Athena queries. How would you handle this schema-on-read challenge?

I design for safe, backward-compatible reads: make new fields nullable, guard casts with try/COALESCE, and, for recurring analytics, convert to Parquet so the schema is explicit and stable.

1. Treat optional fields as nullable by design
   In Glue, I model new or optional fields as nullable types. If I'm staying on raw JSON, I use functions that return NULL when keys are missing (json_extract/json_extract_scalar) and avoid hard casts that throw errors.
2. Guard every cast
   When pulling typed values from JSON, I wrap with try_cast (or try) and default with COALESCE, for example:

- COALESCE(try_cast(json_extract_scalar(json_col, '$.amount') AS double), 0.0)
- COALESCE(try_cast(json_extract_scalar(json_col, '$.flag') AS boolean), false)
  This stops queries from failing when a field is absent or malformed.

3. Use element_at / maps for flexible objects
   If a field lives in a dynamic attributes object, I parse it to a MAP and read keys with element_at(map_col, 'key') which returns NULL if not present. I still wrap with COALESCE for safe defaults.
4. Normalize once into curated Parquet
   For repeated use, I run a CTAS (or Glue job) that parses JSON into a curated Parquet table with explicit nullable columns and correct types. Optional fields become NULLs in Parquet; queries don't need json_extract. This is faster and avoids surprises.
5. Stabilize consumer SQL with views
   I expose analysts to views that select common columns and provide defaults (COALESCE) so dashboards don't break when fields are missing. When a new column appears, I add it to the view after testing.
6. Handle arrays with optional subfields
   When UNNESTing arrays of structs, I still use COALESCE on child fields (e.g., COALESCE(item.optional_attr, 'unknown')). If some array elements lack a child field, this keeps rows from vanishing or producing confusing NULLs.
7. Catch bad or changing data early
   I add a lightweight validation step (Glue job or Athena precheck) that samples new partitions and compares them against expected types. If a field's type flips (string→number), I normalize to a single canonical type (often string) in curated data and maintain a separate raw_json column for audits.
8. Keep partition pruning effective
   Optional fields shouldn't live in partition columns. Partition by stable keys like dt and region so queries can prune even when many optionals are NULL.

Result: optional fields stop breaking queries. On raw JSON, I read them defensively with try/COALESCE; for recurring analytics, I promote them to explicit nullable columns in Parquet. Analysts get stable, fast queries even as the source evolves.

7. Your organization needs to restrict analysts from querying sensitive columns (like salary, SSN) while allowing access to the rest of the dataset. How would you enforce this in Athena?

I put the tables under Lake Formation governance and use column-level permissions (and optional row filters/masking). Analysts only get access to allowed columns, and they can't bypass it by reading the S3 bucket directly.

Step by step

1. Register data locations in Lake Formation
   I register the S3 paths that back the Glue tables and turn on Lake Formation permissions for those databases/tables. I grant the Lake Formation service role data location access and remove direct S3 access from analyst roles.

2. Tag sensitive columns and grant by tag
   I create LF-Tags like pii=yes/no, sensitive=yes/no. I assign pii=yes to columns such as ssn, salary, dob, email. Then I grant analysts SELECT on the table with a tag-based policy that allows only columns where pii=no. Engineers/data stewards get broader access.

3. Optional: add row-level filters and masking
   If we must hide rows (for example, analysts only see their region) I create LF data filters with a WHERE predicate (region = ${caller_region}). For masking (for example, show only last 4 of SSN), I either:
   • use Lake Formation column-level masking (if available in the account/region), or
   • publish an Athena view that applies masking functions (regexp_replace, substr) and grant analysts SELECT only on the view, not on the base table.

4. Deny direct S3 reads so policies are enforceable
   I attach a deny in the analyst role/bucket policy that blocks s3:GetObject on the data paths unless the access is via Lake Formation/Athena. This prevents users from downloading raw files and side-stepping column controls.

5. Keep metadata controlled
   I restrict Glue Create/AlterTable to data engineering so no shadow tables expose sensitive columns. Table comments and a data catalog page document which columns are sensitive.

6. Test and monitor
   I validate with an analyst role that SELECT * returns only non-sensitive columns, and that attempts to reference salary or ssn fail. I log Lake Formation/Athena access in CloudTrail and review routinely.

Result: analysts can query the datasets in Athena but only see permitted columns. Sensitive fields are blocked or masked centrally, with no S3 backdoor.

8. A compliance audit requires all Athena queries and results to be encrypted at rest and in transit. How would you implement this using AWS services?

I enforce KMS encryption for data, results, and metadata, and require TLS for every network hop.

At rest

1. Encrypt source data in S3 with SSE-KMS
   I set a bucket policy that requires server-side encryption with our CMK (deny if x-amz-server-side-encryption isn't aws:kms or if the key isn't our approved key). All lake buckets (raw/curated) use KMS keys with tight key policies.

2. Encrypt Athena query results and spill
   I create an Athena workgroup with "Enforce query result location" pointing to an S3 results bucket that is SSE-KMS encrypted. I enable "Encrypt query results" with the same or a dedicated KMS key and make the workgroup mandatory for all users.

3. Encrypt the Glue Data Catalog and logs
   I enable Glue Data Catalog encryption with KMS and set Glue job bookmarks/checkpoints to S3 prefixes encrypted with KMS. CloudWatch Logs groups used by Athena/Glue are set to use KMS encryption.

4. Encrypt downstream targets
   If results are unloaded/CTAS to S3, those paths inherit SSE-KMS. If copies go to Redshift, the cluster uses KMS (or HSM) encryption and encrypted snapshots.

In transit
5) Require TLS to S3/Athena/Glue
I add bucket policies that deny requests when aws:SecureTransport is false so S3 only accepts HTTPS. Athena, Glue, and the console use TLS by default; for JDBC/ODBC clients I ensure drivers connect with TLS enabled.

6. Keep traffic inside the VPC
   I use VPC endpoints/PrivateLink for S3, Athena, and Glue where supported so traffic stays on the AWS network. Client applications in VPC subnets reach these services privately over TLS.

Governance and proof
7) Enforce via workgroups and SCPs
I force users into the secured Athena workgroup (no "primary" workgroup use) with an IAM policy/SCP. I deny creating or using workgroups that don't enforce KMS-encrypted result locations.

8. Audit trails
   I enable CloudTrail organization-wide and store logs in a KMS-encrypted S3 bucket. I also enable Athena workgroup metrics and access logs, plus S3 server access logs or CloudTrail data events on the data buckets.

9. Key management
   I rotate KMS keys per policy, use grants and key policies that limit decrypt to Athena/Glue roles and specific analyst roles, and monitor with CloudWatch/KMS key usage metrics.

Result: data files, query results, and metadata are KMS-encrypted at rest; all access uses TLS; and policies/workgroups prevent misconfiguration. Audit logs prove enforcement to compliance.

9. You are building a data lake on S3 where multiple teams will run queries using Athena. How would you organize partitions, buckets, and Glue integration to support analytics at scale?

I set up a simple, scalable layout that keeps raw and curated separate, uses predictable partitions, and lets Glue/Athena discover data without manual steps.

Bucket and folder layout

- I create separate buckets (or clearly separated prefixes) per zone: s3://org-raw, s3://org-curated, and s3://org-analytics. Raw holds source drops as-is; curated holds cleaned columnar data; analytics holds derived marts and shared views.
- Inside each bucket I use dataset-first paths, then partitions. Example: s3://org-curated/sales_orders/dt=YYYY-MM-DD/region=APAC/. Dataset-first keeps ACLs, lifecycle rules, and sharing simple.

Partition strategy

- I partition by the columns analysts always filter on, usually dt first, then a low-cardinality dimension (region, tenant, or source_system). Daily is the default; hourly only if queries need it. I avoid overly granular partitions (minute) that explode metadata.
- For very large tables, I also bucket or cluster the files by a frequently joined key (customer_id) at write time if I use Iceberg/Hudi/Delta; if I stay on plain Parquet, I at least sort within partitions to improve min/max pruning.

File format and size

- I convert everything to Parquet (Snappy or ZSTD) in curated/analytics. Target 256–512 MB files per partition to cut S3 request overhead and speed Athena scans. I run compaction jobs if writers produce many small files.

Glue Catalog integration

- I create one Glue database per domain and zone, for example retail_curated, finance_analytics. I register each curated dataset as one authoritative table with correct types.
- For predictable partitions (dt, region), I enable partition projection so Athena can query new dates immediately without crawling. If projection doesn't fit a dataset, I let the ETL add partitions programmatically at the end of the job.
- I keep crawlers only for raw discovery, scoped tightly to a prefix, with "add new columns only" and "new folders only" to avoid schema churn. Curated/analytics schemas are owned by code, not by crawlers.

Governance and access

- I put S3 locations and Glue tables under Lake Formation. Analysts get SELECT on curated/analytics; raw is restricted. If needed, I add column-level grants to hide PII. Bucket policies deny direct S3 reads for analysts to enforce LF policies.
- I tag tables and columns (owner, PII, SLA) in Glue for discovery.

Performance hygiene

- I publish SQL patterns that always filter on dt and region and select only needed columns.
- I monitor Athena workgroup "bytes scanned," average file size per partition, and partition counts; alerts fire when average file size drops (compaction needed) or tables get too many small files.

Result: clear zones, predictable partitions, columnar files, and Glue integration with projection give fast, cheap queries at scale while keeping metadata clean and governed.

10. A client asks you to implement a data lakehouse pattern using Athena, Glue, and Lake Formation. What would the architecture look like, and what problems does it solve compared to a traditional data warehouse?

I build a governed lake on S3 as the storage layer, add an ACID table format for reliability, use Glue as the catalog, Lake Formation for security, and Athena as the SQL engine. It gives warehouse-like reliability and governance with lake flexibility and cost.

Architecture

- Storage on S3 with three zones: raw (immutable drops), curated (cleaned, conformed), and gold/analytics (business-ready marts). Everything is columnar Parquet, optionally Iceberg/Hudi/Delta tables for ACID.
- Glue Data Catalog as the single metadata store. Each table has strict types, ownership tags, and is versioned. ETL jobs update schemas via code.
- Lake Formation governs access. I register S3 locations, apply table/column-level permissions and optional row filters, and deny direct S3 access for analyst roles so all reads go through LF.
- Compute with Athena for serverless SQL over both Parquet paths and lakehouse tables. CTAS/UNLOAD build derived marts. For heavy transforms I use Glue ETL or EMR, writing back to Iceberg/Hudi/Delta so upserts, schema evolution, and time travel are handled.
- Data ingestion via Kinesis/Firehose or batch jobs lands raw JSON to raw/, transforms to curated Parquet with compaction and partitioning. Late-arriving corrections are applied with MERGE into Iceberg/Hudi/Delta.
- Partition projection and table maintenance (compaction, clustering, vacuum) keep metadata small and files right-sized.
- Observability: workgroups with enforced result encryption, CloudWatch metrics on bytes scanned, lineage from curated to raw via manifests, and data quality checks that gate promotion from curated to gold.

What problems it solves vs a traditional warehouse

- Schema evolution and mixed sources: the lakehouse accepts semi-structured data, evolving schemas, and very large volumes; ACID tables make changes safe without full reloads.
- Cost and elasticity: storage is cheap S3, compute is serverless Athena or on-demand EMR/Glue; you pay per scan or per job, not for an always-on cluster.

- Governance at the lake: Lake Formation enforces fine-grained access (columns/rows) across all engines that use the Catalog, something harder to do consistently with ad-hoc files or multiple warehouses.
- Reliability and correctness: ACID table formats bring transactions, snapshot isolation, upserts/deletes, and time travel—addressing the classic "mutable data on S3" problem that a plain data lake struggles with.
- One copy, many engines: the same Iceberg/Hudi/Delta tables can be queried by Athena, Spark on EMR, or Glue without copies, reducing duplication and drift.
- Faster delivery: new datasets can be landed and exposed quickly without modeling everything into rigid warehouse schemas up front; gold tables still give curated, business-friendly views.

Result: the client gets warehouse-like ACID guarantees, governance, and SQL access on open S3 storage, with lower cost and greater flexibility than a traditional warehouse-only approach.

11. You need to join a 1TB fact table with a 50MB dimension table in Athena, but queries are slow. What join strategy would you use, and why?

I make Athena do a broadcast (map-side) join so it sends the tiny dimension to each worker and streams through the big fact without shuffling the 1TB across the network.

How I do it, step by step

- Keep the small table truly small and columnar. Store the 50MB dimension as Parquet (Snappy/ZSTD), project only needed columns, and filter it first in the query (for example, WHERE active = true) so it's as small as possible.
- Put the small table on the right side of the join. Athena/Presto will usually broadcast the right-hand table when it's small. Example: FROM fact f JOIN dim d ON f.dim_id = d.id.
- Push filters into the fact before the join. Always filter fact by partitions (for example, WHERE f.dt BETWEEN … AND … AND f.region IN (…)) so Athena scans only relevant files.
- Make the fact table partitioned and compact. Partition the fact by date (and region/tenant if common filters). Keep Parquet file sizes ~256–512 MB in each partition so scanning is fast.
- Avoid data type mismatches. Ensure f.dim_id and d.id have the same type (both BIGINT or both VARCHAR). A cast on the big side kills performance.
- Join only once and select only what you need. Don't fan out with many joins; do one clean join and pick the needed columns to minimize data moved/written.
- If the dimension is reused a lot, precompute a "filtered dim" CTE or small CTAS table first, then join to that. Smaller broadcast → faster join.

Why this works
Broadcasting a ~50MB Parquet table is cheap; shuffling a 1TB fact table is not. With broadcast, each worker gets the small hash table in memory and streams its share of the fact, which is the fastest pattern for big fact + tiny dim.

12. Your marketing team runs frequent queries joining user activity logs with reference data. The queries are slow. How would you use CTAS to improve performance?

I materialize a denormalized, analytics-ready table with CTAS so day-to-day queries hit a small, partitioned Parquet dataset instead of rejoining raw logs every time.

What I build

- A CTAS job that runs hourly/daily:

```
CREATE TABLE analytics.user_activity_enriched
WITH (
 format = 'PARQUET',
 write_compression = 'SNAPPY',
 external_location = 's3://org-analytics/user_activity_enriched/',
 partitioned_by = ARRAY['dt','region']
) AS
SELECT
 l.user_id,
 l.event_time,
 date(l.event_time) AS dt,
 l.region,
 r.segment,
 r.campaign_id,
 -- only the columns marketing needs
 l.event_type,
 l.item_id
FROM logs l
JOIN ref_users r ON l.user_id = r.user_id
WHERE l.dt BETWEEN date_add('day', -7, current_date) AND current_date;
```

- The CTAS output is Parquet, partitioned by dt and region, and contains only the columns marketing uses. I keep files large (128–512 MB) by controlling upstream file sizes or running periodic compaction if needed.
- I schedule the CTAS (or a small Glue job that runs the same SQL) so the enriched table stays fresh. Marketing queries then do:
  SELECT … FROM analytics.user_activity_enriched WHERE dt BETWEEN … AND … AND region IN (…)
  No heavy join in their reports—Athena prunes partitions and reads only a few columns.

Extra improvements

- If reference data changes slowly, I refresh only the latest partitions (yesterday/today) instead of rebuilding history.
- I add useful derived fields (for example, is_new_user, days_since_signup) inside the CTAS so every dashboard doesn't recompute them.
- I publish a stable view over the CTAS table to hide paths and enforce filters.
- I verify data types align and that user_id is consistent across sources to avoid runtime casts.

Why this helps
CTAS turns repeated ad-hoc joins into one scheduled join that writes compact, partitioned Parquet. The result is fewer bytes scanned, less CPU, and much faster queries for the marketing team.

13. Your company wants to run analytics combining transactional data from RDS and historical data in S3. How would you design a federated query solution in Athena?

I would let Athena join the two worlds using a JDBC federated connector for RDS and standard Glue tables for S3, but I would also materialize the hot RDS slices into S3 for repeatable performance.

Design at a glance

- Keep S3 as the lake for historical data in Parquet, partitioned by date (and region/tenant). Registered in Glue.
- Expose RDS (MySQL/Postgres/Aurora) to Athena using the Athena Federated Query JDBC connector deployed as a Lambda in the account. Create an Athena data source that points to RDS through a secure VPC connection.
- Create external schemas: S3 tables in Glue DBs, and a separate federated catalog for RDS tables. Analysts can now SELECT ... FROM glue_db.table t JOIN "rds-catalog".db.table r ON ....

Security and networking

- Put RDS in private subnets. The Athena connector Lambda runs in the same VPC/subnets with security groups allowing DB port access. Use TLS to RDS and Secrets Manager for credentials.
- Enforce Lake Formation permissions on S3-backed tables and IAM policies for using the RDS connector. Encrypt query results with KMS.

Performance practices

- Always push filters to each side first. For example, filter S3 by dt range and project only needed columns; filter RDS by time or status via a subquery so the connector pulls back small result sets.
- Join keys must have matching types; avoid casting big S3 sides at runtime.
- Limit federated joins to small/medium result sets. For repeated analytics, schedule a job that pulls the needed RDS slice to S3 as Parquet (CTAS or Glue job), then join S3-to-S3.
- For ongoing freshness, use CDC from RDS to S3 (DMS → S3 Parquet) to keep a “current” incremental table in the lake and avoid live federated joins for heavy dashboards.

When I would materialize

- If the same reports run daily, create a CTAS that snapshots the RDS subset into S3 and then joins with the historical Parquet. This is faster, cheaper, and avoids load on the OLTP.

Result
Analysts can do ad-hoc joins through Athena federation, but production dashboards rely on S3-parquet materializations or CDC so performance is predictable and the OLTP is protected.

14. Analysts are complaining that federated queries against DynamoDB and S3 are very slow. How would you optimize performance or redesign the pipeline?

Federated connectors are great for exploration but slow for heavy analytics. I would first make the queries more selective, then materialize recurring joins into S3 so Athena scans Parquet instead of live DynamoDB.

Quick wins on federation

- Push down predicates. Ensure the query filters by key on the DynamoDB side (partition/sort keys). Avoid scans; fetch only selective ranges.
- Project only the columns you need. The connector moves fewer bytes.
- Reduce result set size. If you only need IDs from DynamoDB, pull those first and join to S3 in a second step.
- Watch connector limits. Large joins time out in Lambda; increase memory/timeout only helps a little.

Better long-term pattern (recommended)

- Materialize DynamoDB to S3 on a schedule. Use DynamoDB Streams → Firehose/Glue Streaming or Export-to-S3 to land change logs or snapshots as Parquet, partitioned by date/tenant.
- Build a curated S3 table for the DynamoDB data with proper types and 128–512 MB files.
- Join S3-to-S3 in Athena. This removes Lambda connectors from the hot path and is orders of magnitude faster and cheaper.

If I must stay live for freshness

- Cache hot DynamoDB subsets to S3 frequently (hourly CTAS or micro-batch) and point dashboards to the cached table.
- Or use Redshift with Spectrum: copy the selective DynamoDB subset into Redshift temp/staging tables, then join to Spectrum S3 tables; Redshift handles joins more efficiently for repeated workloads.

Operational and governance pieces

- Put a workgroup budget and track bytes scanned. Alert when a federated query scans too much or runs too long.
- Document best practices: always filter on DynamoDB keys, avoid wild-card scans, and prefer materialized S3 tables for BI.
- Secure both paths with Lake Formation and KMS encryption, and keep credential rotation for the connectors in Secrets Manager.

Result
Ad-hoc exploration can still use federation, but regular analytics run on S3 Parquet materializations—queries become fast, cheap, and reliable while the operational stores (like DynamoDB) aren’t burdened.

15. A critical Athena query is failing with HIVE_BAD_DATA errors when reading CSV files from S3. How would you troubleshoot and resolve this?

I assume the error means Athena can't parse some rows using the table's SerDe and column types. I narrow down which files/rows break, fix the schema/SerDe to match the actual CSV, cleanse data if needed, and then move the workload to Parquet so it's stable.

How I troubleshoot

1. Reproduce on a tiny slice. I run the same SELECT with a strict WHERE on one date/partition and LIMIT 1000 to see if it still fails. I also try listing S3 keys and querying a single file using the path in a LOCATION override (or an external table pointing only to that key).
2. Inspect raw lines. I download a failing object and check: delimiter, quotes, escapes, header row, embedded newlines, column counts, stray commas, and non-UTF8/BOM.
3. Check table DDL vs reality. I compare the SerDe and column types to the file. Typical mismatches:
   • Wrong delimiter (comma vs pipe/tab)
   • Missing quote/escape config (values like "New York, NY")
   • Header rows not skipped
   • Extra columns or fewer columns on some lines
   • Type cast issues (string in an INT column)
   • Windows line endings \r\n or stray \r
   • Files with different structures in the same prefix
4. Run a defensive probe. I select all columns as strings using the raw SerDe, or use regexp_extract_all on the raw line (if I temporarily register a one-column "line" table) to see per-field content.

Fixes I apply

1. Correct the SerDe and table properties. For CSV I use OpenCSVSerDe or LazySimpleSerDe with explicit properties:
   • field delimiter (e.g., ',' or '\t')
   • quoteChar and escapeChar for quoted fields
   • skip.header.line.count='1' if there's a header row
   • set all string columns first to confirm parsing, then tighten types gradually
2. Align data types. I make the table columns match the file, or cast in SELECT: try_cast(col as bigint) to avoid hard failures. After a successful CTAS to Parquet I can enforce strict types.
3. Handle embedded newlines. If values contain newlines, I must rely on proper quoting in the source and the OpenCSVSerDe with quote/escape set. If the source is malformed, I cleanse upstream or run a one-time Glue job to repair.
4. Isolate bad files. If only a few files are corrupt, I move them to a quarantine prefix so queries stop touching them, then backfill fixed versions.
5. Unify layout per prefix. I do not mix different CSV shapes in the same table LOCATION. If there are multiple layouts, each gets its own table/prefix.

Make it robust long-term
• Convert to Parquet via CTAS or a Glue job (cast and clean in one place). Parquet avoids CSV quirks, stores types, and will cut query cost/latency.
• Add a lightweight ingest validation that checks column counts, encodings, and a sample cast to catch bad drops before they hit "curated".
• For evolving schemas, add new columns as nullable and keep older queries using views so they don't break.

Result: I fix the immediate SerDe/type mismatch, quarantine or repair bad files, and then move the workload to partitioned Parquet so HIVE_BAD_DATA stops happening and performance improves.

16. Your leadership wants visibility into query costs and usage by team. How would you set up monitoring and logging for Athena queries to achieve this?

I organize teams into Athena workgroups, tag them, and collect usage and cost from CloudWatch, CloudTrail, and the Cost and Usage Report. Then I publish simple dashboards and alerts.

How I set it up

1. Workgroups per team. I create one Athena workgroup per team (for example, marketing_wg, finance_wg) and make those mandatory in IAM policies. Each workgroup has:
   • its own KMS-encrypted results bucket/prefix
   • data usage controls enabled (optional scan limits)
   • a cost-allocation tag like Team=Marketing
2. Enforce usage. I attach IAM policies so users can only StartQueryExecution in their team's workgroup. This guarantees attribution.
3. Collect metrics. I enable workgroup metrics in CloudWatch (QueryCount, DataScannedInBytes, EngineExecutionTime). I build dashboards and alarms (for example, bytes scanned spikes).
4. Audit queries. I enable CloudTrail data events for Athena so every StartQueryExecution and GetQueryResults is logged with user, workgroup, query ID, and time. I also enable S3 server access logs or CloudTrail data events on the data buckets to see which datasets are most used.
5. Cost attribution. I activate cost allocation tags (Team) and make sure the results and Lambda/Glue resources are tagged similarly. I ingest the AWS Cost and Usage Report (CUR) into Athena and join CUR line items for Athena with CloudTrail/CloudWatch data to get per-workgroup and per-user costs.
6. Show true cost per query. Athena charges roughly by data scanned, so I compute cost = bytes_scanned / 1 TB × price per TB. I cross-check with CUR for accuracy and include S3 request costs if relevant at high volumes.
7. Dashboards and reports. I create an Athena/QuickSight dashboard by team and by top users: queries run, bytes scanned, estimated cost, top tables scanned, and failure rate. I send weekly email digests and monthly rollups.
8. Guardrails to reduce spend. I add views that pre-filter by date/region, publish "best practice" query snippets, and set optional per-workgroup scan quotas. If someone exceeds a threshold, the workgroup can throttle or reject queries with a friendly message.

Result: every query is tied to a team via workgroups and tags, leadership gets per-team usage and cost with simple dashboards, and we have guardrails to keep spend under control while keeping analysts productive.

17. Your data engineering team wants to run daily transformations in Athena and store pre-aggregated results in S3 for BI tools. How would you automate this pipeline?

I set up a simple, reliable daily job that runs Athena SQL, writes Parquet outputs to S3 (partitioned), and makes BI read only those pre-aggregates. I automate with EventBridge + Step Functions so runs are parameterized, idempotent, and observable.

How I design it

- Storage layout: I keep a curated zone in S3 (Parquet) and an analytics zone for pre-aggregates: s3://lake/analytics/<dataset>/dt=YYYY-MM-DD/region=.... Files target 256–512 MB for fast Athena reads.
- Tables: I register curated and analytics tables in Glue. I prefer an Iceberg table for analytics so I can do incremental INSERT INTO, MERGE, and time-travel; otherwise a standard external Parquet table works fine.
- SQL pattern:
    - If creating a fresh snapshot each day → CTAS into a temporary staging location, then atomically swap the partition location (or replace the Iceberg snapshot).
    - If appending to an existing table → INSERT INTO analytics.table PARTITION (dt='${run_date}', region) selecting from curated with all filters and casts in SQL.
    - I always filter source by partition (e.g., WHERE dt = '${run_date}') and select only needed columns.

Automation

- EventBridge schedule (e.g., 02:00 UTC) → Step Functions workflow.
- Steps:
    1. "Data ready" check (S3 prefix exists, expected file count/size).
    2. Run Athena query via StartQueryExecution in a dedicated workgroup (KMS-encrypted result bucket). Pass ${run_date} as a parameter.
    3. Optional validation query (row counts, min/max dates).
    4. Publish a success marker (e.g., _SUCCESS) or update a manifest table.
- Idempotency: write to a staging prefix like .../staging/dt=... first; on success, move/rename to .../dt=.... If rerun, staging is overwritten, so no duplicates. Iceberg makes this easier by committing a new snapshot only on success.
- Cost/performance: use CTAS/INSERT to Parquet with write_compression='ZSTD', partition by date and a low-cardinality dimension (region/tenant), and keep file sizes large. Enable partition projection on the analytics table so BI can query immediately.
- Governance: Lake Formation grants analysts SELECT on analytics tables only (not raw). Deny direct S3 reads for analyst roles.
- Observability: CloudWatch logs for Athena queries, Step Functions state transitions, and alerts on failures. Track Athena workgroup metrics (bytes scanned, execution time).

Outcome
BI tools query the small, partitioned pre-aggregate table instead of raw data. The daily job is serverless, cheap, idempotent, and easy to monitor/rollback.

18. A scheduled Athena report fails intermittently due to schema mismatches in source data. How would you build resilience into the automation process?

I add a staging/validation step before publishing, make schema evolution additive and explicit, and harden the SQL so minor drift doesn't break the run. If source changes are incompatible, the job quarantines data and alerts instead of producing bad reports.

What I change

- Two-phase publish: write into a staging table/prefix first, validate, then promote. If validation fails, production partitions are untouched. With Iceberg, this is a new snapshot committed only after checks pass.
- Schema governance: treat curated schemas as code. New fields are added as nullable (no drops/renames in place). I update Glue with ALTER TABLE ADD COLUMNS from the pipeline (not via an open-ended crawler).
- Preflight schema checks: before the report query, run a small Athena probe that samples latest partitions and compares column existence/types to expectations (or use Glue Schema Registry/JSON Schema in a Lambda step). If a field flipped type, fail fast with a clear message.
- Defensive SQL: use try_cast and COALESCE for optional fields, e.g., COALESCE(try_cast(amount as double), 0.0) so occasional bad rows don't crash the job. Keep strict casts inside the CTAS to Parquet so downstream is clean.
- Quarantine path: if incoming data is malformed (wrong delimiter, type flip), move those objects to .../quarantine/dt=... and alert. The report runs on the last good partition or skips the bad region/date.
- Rolling views: expose analysts to a stable view that selects only known-good columns. When schema evolves, I update the view in lockstep after validation.
- Retry/backoff: Step Functions retries the Athena task with jitter for transient issues (service blips). Persistent schema errors break the workflow and notify Slack/Email with the failing column and file sample.
- Compaction and typing: as part of staging, normalize types and write Parquet with consistent dtypes. This removes CSV/JSON quirks that often trigger HIVE_BAD_DATA and mismatches.
- Monitoring: add canary queries post-publish (row counts vs yesterday, null-rate thresholds, min/max bounds). If anomalies exceed thresholds, automatically roll back to previous snapshot (easy with Iceberg) or unpublish the just-written partition.

Outcome
The pipeline either produces a correct report or fails safely with quarantined inputs and actionable alerts. Additive schema evolution, defensive casts, and a two-phase publish mean intermittent source drifts no longer take the report down.

19. Your Athena query is scanning 100 GB of data daily, but only 5 GB is relevant. How would you optimize partitioning to reduce scan size?

I redesign the S3 layout so Athena can skip whole folders that aren't needed, and I make sure users always filter on those partition columns.

- Pick partition keys that match how people filter. If queries always filter by date and region, I use dt first, then region: s3://lake/curated/table/dt=YYYY-MM-DD/region=APAC/. If most queries also filter by tenant or source, add that as the next level. Keep keys low-cardinality; avoid minute-level partitions.
- Put the most selective column first. Date is usually first because every query has a time window. If region massively cuts data, make it the second level so Athena prunes early.
- Convert to Parquet and keep big files. Columnar + 256–512 MB per file per partition means fewer opens and better pruning. Compaction fixes legacy small files.
- Use partition projection. Define ranges for dt and an enum for region in the table properties so new partitions are queryable immediately and Athena doesn't need a crawler run. This prevents "full scan because metadata is missing."
- Enforce filters with views or sample SQL. Publish a view that requires dt and region, and distribute query templates that always include WHERE dt BETWEEN … AND … AND region IN (…).
- Sort within partitions on a frequent filter key. When writing Parquet, sort by common predicates (like tenant_id) so Athena can skip Parquet row groups using min/max stats.
- Keep hot and cold separate. Recent days in one table or prefix, deep history in another, so day-to-day queries never even see the cold path unless requested.
- Monitor and iterate. Track Athena bytes scanned per table and average object size. If scans are still high, re-check how analysts filter and add or adjust a partition level accordingly.

Result: the query touches only the dt and region folders it needs, scans around 5 GB instead of 100 GB, and runs much faster and cheaper.

20. Your dataset is stored in JSON. Queries are slow and costly. How would you redesign the storage layer to improve Athena performance?

I move from raw JSON to a curated, columnar layout that Athena can read efficiently, while keeping raw JSON for audits.

- Land a curated Parquet copy. Use CTAS or a Glue job to read JSON, cast types, and write Parquet (Snappy or ZSTD) to s3://lake/curated/table/. Parquet lets Athena read only needed columns and skip row groups by min/max stats.
- Partition to match filters. Organize as dt=YYYY-MM-DD and one more low-cardinality key like region or tenant. This enables partition pruning so Athena skips entire folders.
- Keep healthy file sizes. Target 256–512 MB Parquet files per partition. Raise writer buffering or run a compaction job so you don't end up with thousands of tiny files.
- Normalize schema and handle optionals. Make new fields nullable in Parquet and standardize data types (no mixing string and int for the same column). This removes the json_extract overhead in every query.
- Optional lakehouse table. If you expect late data or upserts/deletes, write to Iceberg/Hudi/Delta and use MERGE. Athena then queries the latest snapshot while maintenance jobs handle compaction and clustering.
- Preserve raw JSON separately. Keep gzip JSON in s3://lake/raw/table/ with lifecycle to cheaper tiers. Analysts query the curated Parquet; raw is only for audits or reprocessing.
- Register clean metadata. Create a Glue table over the curated path with correct types and partition projection. Avoid generic crawlers changing schemas unexpectedly.
- Publish analyst views. Provide a view with the most used columns and sensible defaults to simplify queries and reduce accidental wide scans.

Result: queries run against compact, partitioned Parquet instead of parsing JSON on the fly, cutting scanned data and runtime by an order of magnitude while still retaining raw JSON for compliance and reprocessing.

21. A Glue Crawler inferred incorrect column types (e.g., string instead of bigint), causing Athena queries to fail. How would you fix this without breaking existing reports?

I correct the schema in a controlled way and keep analysts insulated behind a stable view, so no one's report breaks while we fix data types.

1. stop the crawler from flipping types again
   I scope the crawler to the dataset prefix and change its schema change policy to "add new columns only." For curated tables, I prefer not to use a crawler at all; I manage schema with code (Glue API or CTAS).

2. create a clean, type-correct table (don't mutate prod first)
   I run a CTAS over the affected partitions that casts columns to the correct types and writes Parquet to a staging prefix, partitioned the same way. Example: select try_cast(col as bigint) as col, handle bad rows with defaults or send them to a quarantine table.

3. swap analysts to the fixed data via a view
   If analysts currently query a table name directly, I insert a stable view layer:

- Create a view name that analysts use (or repoint an existing view) to select from the new, type-correct table.
- Keep column names identical so dashboards don't change. If I must rename, I alias in the view.

4. backfill and promote safely
   After validating counts and spot-checking values, I either:

- ALTER the original table's partition locations to the new, fixed files, or
- retire the old table and keep the view pointing to the new table location.
  Lake Formation grants stay on the view, so permissions don't change.

5. prevent future mis-typing
   I manage schema as code:

- Glue: run ALTER TABLE ADD COLUMNS or UpdateTable via pipeline code, never by open-ended crawler.
- Ingest validation: sample new files; fail the run if a critical column type drifts. Keep optional fields nullable.
- Store data in Parquet with correct types so crawlers don't have to "guess."

Result: queries start reading a type-correct Parquet surface immediately (via the view), reports keep working, and future schema is controlled by code instead of guesswork.

22. Your team needs to track schema changes over time for compliance. How would you handle schema versioning with Glue + Athena?

I version schemas explicitly, store the history, and give auditors the ability to query "as of" a date without guessing what the schema was.

1. keep schema history in Glue
   Glue Data Catalog supports table versions. I update schemas via code (Glue API/CLI), not by ad-hoc crawlers, so every change creates a new TableVersion with who/when/what. I tag versions with semantic versioning in table parameters (for example, schema_version=1.4.0).

2. store the canonical schema file
   Alongside the table, I keep a JSON (or Avro) schema document in S3 under a versioned path (for example, s3://meta/schemas/dataset/v1.4.0.json) and record its URI in Glue table properties. This is the contract we validate against at ingest.

3. enforce compatible evolution
   All changes are additive/backward compatible (add nullable columns). Renames/drops create a new major version (v2) in a new table or Iceberg schema update with a documented migration window. Pipelines validate incoming data against the current schema (Glue Schema Registry or a small validation Lambda) and quarantine incompatible files.

4. use a table format that supports time travel (optional, very helpful)
   If I use Iceberg/Hudi/Delta for curated data:

- I can query historical snapshots with Athena (Iceberg): SELECT … FROM table FOR VERSION AS OF <ts/snapshot>.

- Schema evolution (add/rename) is tracked in the table's metadata, and auditors can see both the schema at a point in time and the data as it existed.

5. views for stable consumer contracts
   Analysts use a stable view (dataset_current) that points to the latest versioned table. If a breaking change happens, I keep dataset_v1 and dataset_v2 in parallel, and the view flips only when downstream teams are ready. Old reports can continue to reference dataset_v1.

6. audit trail and discovery
   I log schema change events in a small "schema_changes" table (who, when, reason, Git commit, Glue version id). CloudTrail captures Glue UpdateTable calls. I expose a simple Athena view listing all columns and their first_seen/last_seen versions.

7. reproducibility of reports
   For month-end audits, I pin reports to a snapshot:

- If Iceberg: run the report with FOR VERSION AS OF the month-end snapshot.

- If plain Parquet: CTAS a monthly snapshot with the then-current schema into a versioned prefix.

Result: every change is recorded, validated, and discoverable; analysts have a stable interface; auditors can query historical data with the exact schema that was in effect at that time.

23. You have an Avro dataset in S3, but analysts need it in a relational view. How would you expose this to them via Athena?

I would register the Avro files in the Glue Catalog, then give analysts a clean, columnar, SQL-friendly surface using a view or a CTAS table in Parquet.

1. register the Avro as a table

- Point a Glue Crawler (scoped to the Avro prefix) or use a CREATE TABLE with Avro SerDe so Athena knows the schema.
- Keep the Avro table in a "raw" database; don't let analysts query it directly if it contains complex types.

2. design a relational projection

- Write a SELECT that picks only the columns analysts need, casts types as needed, and flattens simple structs (for example, raw.customer.name AS customer_name).
- If there are enums or unions in Avro, normalize them to simple VARCHAR/DOUBLE using COALESCE/try_cast.

3. expose a stable view for quick use

- CREATE VIEW analytics.avro_relational AS SELECT ... FROM raw.avro_table;
- This gives instant access with familiar column names. Use this for exploration or low-volume workloads.

4. materialize to Parquet for speed

- Use CTAS to write the relational projection to S3 as Parquet with compression (Snappy/ZSTD), partitioned by dt and another low-cardinality key (like region).
- Register that Parquet table in Glue and point BI tools to it. Parquet is much faster and cheaper to scan than Avro.

5. handle evolution safely

- Add new fields as nullable in Avro; update the view/CTAS when needed.
- Keep the raw Avro table intact for audits; the curated Parquet table is the one analysts use daily.

Result: analysts see a clean, relational schema (view or Parquet table) while the raw Avro remains in S3 for traceability and reprocessing.

24. You receive log data where some rows have nested arrays of objects. How would you flatten these structures to make them queryable in Athena?

I would use UNNEST to explode arrays, select the child fields I need, and then materialize the flattened result as Parquet so reports don't explode arrays every time.

1. make sure Athena sees arrays/structs as types

- If logs are JSON, either define a typed schema (ROW/ARRAY) or read the raw string and cast paths to ARRAY<ROW<...>>.
- Example: CAST(json_extract(raw, '$.items') AS ARRAY(ROW(id VARCHAR, qty BIGINT, price DOUBLE))).

2. flatten with UNNEST

```
SELECT
 e.event_id,
 e.dt,
 i.id   AS item_id,
 i.qty  AS quantity,
 i.price
FROM raw.events e
CROSS JOIN UNNEST(e.items) AS t(i)
```

- If arrays are nested (items[].discounts[]), UNNEST step-by-step: first items, then discounts from the item alias. Use WITH ORDINALITY if you need element positions.

3. filter early and select few columns

- Always filter by partitions (dt, region) before UNNEST to keep scanned bytes low.
- Project only needed parent and child fields.

4. materialize a flattened table for BI

- Run a CTAS to write the flattened result to S3 as Parquet (Snappy/ZSTD), partitioned by dt (and optionally region). Target 256–512 MB files per partition to avoid small-file issues.
- Point dashboards to this Parquet table, not to the raw logs.

5. handle missing/optional fields cleanly

- Use COALESCE/try_cast on optional child fields so NULLs or malformed values don't break queries.
- If some rows have empty arrays, a LEFT JOIN UNNEST pattern can preserve the parent row when needed.

6. keep raw and curated separate

- Leave the raw log table intact for audits. The flattened Parquet table is the analytics surface. Update it on a schedule (hourly/daily) or via incremental CTAS for the latest partitions.

Result: nested arrays become a simple, fast table that Athena and BI tools can query efficiently, while the raw logs stay available for reprocessing and compliance.

25. You need to ensure that external contractors can only query anonymized data (e.g., masked PII). How would you enforce this with Athena + Lake Formation?

I would put all analyst access through Lake Formation so we can enforce fine-grained policies, then expose only masked/allowed columns to contractors. I also block any direct S3 access so policies can't be bypassed.

- Register data and hand control to Lake Formation
  I register the S3 locations for my curated tables in Lake Formation and turn on LF permissions for the Glue databases. From now on, users get data via LF grants, not raw S3.
- Separate what contractors see
  I create either:
  1. dynamic masking via Lake Formation policies (preferred), where columns like ssn, phone, email are masked on the fly for the contractor principals, or
  2. a masked surface (Athena view or physically anonymized table) that selects only non-PII columns and applies masking functions (for example, regexp_replace(email, '(^.).+(@.*$)', '1**\2'), substr(ssn,-4)).
     Contractors are granted SELECT only on this masked surface. Internal staff can be granted broader access.
- Use LF-Tags and column-level grants
  I tag sensitive columns (pii=yes). Then I grant contractors SELECT on the table with a column-set that excludes pii=yes, or I grant by LF-Tag so any new pii column stays hidden automatically.
- Optional row filters
  If contractors should only see their tenant/region, I attach an LF data filter with a predicate (tenant_id = 'X' or region IN (...)) so even masked data is limited to their slice.
- Deny S3 backdoors
  I remove direct S3 permissions from contractor roles and add a bucket policy deny unless the call is authorized via Lake Formation/Athena. That prevents downloading raw Parquet/JSON outside LF controls.
- Governance extras
  All queries go through an Athena workgroup dedicated to contractors (KMS-encrypted results to a separate bucket). I enable LF/Athena audit logging and CloudTrail so every access is recorded.

Result: contractors can run SQL in Athena but only against masked/approved columns and rows, enforced centrally by Lake Formation, with no way to bypass via S3.

26. A security team requires query-level auditing to track who accessed sensitive datasets. How would you implement this?

I would attribute every query to a team/workgroup, force all access through Lake Formation, and collect audit logs from Athena, Lake Formation, and S3 into a searchable trail.

- Enforce identity and routing
  I create Athena workgroups per team (Marketing_WG, Finance_WG, Contractors_WG) and require their use in IAM. Workgroups have KMS-encrypted result locations and cost/scan limits. Users assume named IAM roles (no shared creds) so audits show real principals.

- Centralize permissions in Lake Formation
  Sensitive tables are governed by LF with column/row-level policies. Only LF-authorized reads are allowed; S3 bucket policies deny direct access unless via LF. This ensures audits reflect true data access, not just object downloads.

- Turn on auditing sources

  1. CloudTrail management and data events: logs Athena API calls (StartQueryExecution, GetQueryResults) and S3 object access.
  2. Lake Formation data access audit: records granted/denied data access decisions for governed tables (who, table/columns, when).
  3. Athena workgroup metrics/CloudWatch: execution time, bytes scanned, failures, linked to query IDs and workgroups.

- Build an audit lake
  I deliver CloudTrail (and optional S3 server access logs) to a KMS-encrypted S3 bucket. I create Glue tables over:
  • cloudtrail_events (for Athena StartQueryExecution with user, role, workgroup, query string),
  • lakeformation_audit (LF data access decisions),
  • s3_data_events (object-level reads/writes on data buckets).
  I join these to map each query to the exact datasets/columns touched and to confirm it went through LF.

- Make it easy to use
  I publish Athena views/dashboards that answer: who queried table X yesterday; which columns (PII) were accessed; top users by bytes scanned; failed access attempts. For near-real-time, I add EventBridge rules on CloudTrail events to notify security when sensitive tables are queried outside approved workgroups.

- Guardrails and retention
  I set retention policies on logs (for example, 1–3 years), rotate KMS keys per policy, and restrict who can read audit buckets. I also enable anomaly or threshold alerts (for example, unexpected queries from a contractor role, or query bytes spikes on sensitive data).

Result: every Athena access to sensitive datasets is attributed (who/when/what), governed by Lake Formation, captured in CloudTrail/S3 logs, and queryable in Athena—satisfying audit and compliance needs with clear, actionable records.

27. A business team runs frequent exploratory queries using SELECT * across TBs of data. How would you control costs while still giving them access?

I make it easy for them to do the right thing and hard to do the expensive thing. I use workgroups with limits, give them smaller "explore" tables/views, and keep the heavy raw tables for engineering-only.

- Put the team in its own Athena workgroup with guardrails. I set a data scan limit per query and per day, enforce a KMS-encrypted results bucket, and route all their sessions to this workgroup via IAM. They still have access, but runaway queries are stopped automatically.
- Give them curated, narrow views instead of raw tables. I publish views that select the most-used columns and require filters like dt and region. These views sit on top of Parquet tables and hide wide/expensive columns by default.
- Offer "sample" and "recent" tables for exploration. I maintain small CTAS tables (for example, last 7 days only, 1% sampled, or top N tenants). They can iterate quickly and cheaply there; once logic is ready, they run it on the full table.
- Enforce partition pruning through design. All curated tables are partitioned (dt first, then region/tenant). The views and example SQL include mandatory WHERE dt BETWEEN ... AND ... so scans stay small.
- Educate and nudge with defaults. I share a starter SQL notebook that uses SELECT column list, not SELECT *. I add QuickSight datasets or saved Athena queries that already filter and project columns.
- Convert everything they touch to Parquet with big files. Columnar + 256–512 MB files per partition cuts scanned bytes massively compared to CSV/JSON and avoids many small-file opens.
- Turn on result reuse and caching where appropriate. If they re-run the same query, Athena can reuse cached results instead of rescanning.
- Track and report usage. I tag their workgroup with Team=Business, ingest Athena workgroup metrics and CUR into a dashboard, and send weekly summaries: top queries, bytes scanned, suggestions to trim column lists or add filters.

Result: they can explore freely, but in practice they hit smaller, partitioned, columnar surfaces with limits and examples that keep costs predictable.

28. Your Athena spend increased because multiple teams are querying the same datasets. How would you optimize costs using materialized views or CTAS tables?

I precompute the shared heavy work once and let everyone query the cheap, prebuilt result. I use Athena materialized views for small/medium rollups and CTAS tables for larger denormalized outputs.

- Identify hot, repeated patterns. I find the top expensive queries (same joins/filters/aggregations) across teams from workgroup metrics and CloudTrail logs.
- Build Athena materialized views for common rollups. For example, daily user_activity_by_region with filters on dt and region. Teams then query SELECT … FROM mv_user_activity …. I schedule or event-trigger REFRESH after new partitions land so the MV stays current. Queries on the MV scan only the small, incremental delta.
- Use CTAS to create shared, denormalized Parquet tables. For bigger joins (fact + several dims) or wide flattening, I run a CTAS nightly/hourly that writes to s3://lake/analytics/… partitioned by dt (and region/tenant). Everyone queries this CTAS table instead of redoing the join. Files are 256–512 MB for fast scans.
- Incremental maintenance only. I rebuild just the newest partitions (yesterday/today) rather than the whole table. With Iceberg/Hudi/Delta, I INSERT INTO or MERGE the latest slice and let the table handle snapshots and compaction.
- Publish lightweight, team-friendly views over the MV/CTAS outputs. These views hide paths, enforce partition filters, and expose only the columns teams need—avoiding SELECT *.
- Decommission expensive entry points. I keep raw/curated base tables for engineering, but I remove broad SELECT permissions for general users or route them to the MV/CTAS views via Lake Formation grants. This stops duplicate heavy scans at the source.
- Monitor and tune. I watch bytes scanned on base vs MV/CTAS tables. If scans on base remain high, I create another MV/CTAS for that pattern. I also check MV refresh times and ensure refresh only touches new partitions.

Result: instead of ten teams each paying to re-scan and re-join TBs, we compute once (MV/CTAS) and everyone reads a compact, partitioned Parquet surface—costs drop and performance improves without limiting access.

29. Your company wants to migrate from a traditional on-prem data warehouse to an Athena + S3 lakehouse. What challenges would you anticipate, and how would you mitigate them?

I expect changes in tools, performance patterns, governance, and data modeling. I would adopt a lakehouse design on S3 (Iceberg/Hudi/Delta), add clear contracts and quality checks, and guide teams with standards so we don't recreate warehouse problems in the lake.

What typically breaks and how I'd handle it

- Different performance model: on-prem warehouses cache data and have fixed compute; Athena charges by data scanned and hates small files. I'd convert data to Parquet/ZSTD, partition by dt and one low-cardinality key, target 256–512 MB files, and schedule compaction.
- ACID/upserts/deletes: plain S3 is append-only. I'd store mutable tables in Iceberg/Hudi/Delta and do MERGE/DELETE safely; use time-travel for audits and rollback.
- Workload isolation: warehouses have queues; Athena is serverless. I'd use Athena workgroups per team with scan limits, separate result buckets, and optional Concurrency limits, plus EMR/Glue for heavy transforms off the analyst path.
- Schema evolution: warehouses enforce schemas; lakes can drift. I'd use a schema registry, make evolution additive (nullable new columns), block breaking changes at ingest, and version schemas in Glue.
- Data quality and SLAs: I'd add Great Expectations/Deequ checks in pipelines, publish a data contract per table (owner, SLA, fields), and fail fast to quarantine bad drops.
- Security/governance: I'd replace DB grants with Lake Formation table/column/row-level permissions, deny direct S3 reads for analysts, and encrypt with KMS end-to-end.
- Cost predictability: scanning TBs can explode bills. I'd publish curated, narrow CTAS tables and materialized views for common queries; enforce partition projection and sample/recent datasets for exploration; monitor bytes scanned per workgroup.
- Skills and tooling: teams used to SQL-on-warehouse need patterns for UNNEST, partition filters, and CTAS. I'd provide query templates, example views, and a semantic layer (views or BI datasets) so most users don't touch raw.
- CDC from OLTP: I'd use DMS/CDC to land change logs to S3 as Parquet, then MERGE into Iceberg tables to keep "current" snapshots.
- Orchestration and ops: I'd use EventBridge + Step Functions (or Airflow) to run CTAS/INSERT jobs, with two-phase publish (staging → validate → promote), checkpoints, and alerts.

Outcome: we keep warehouse guarantees (ACID, governance, curated marts) with lake flexibility and cost, and we guide teams so performance and quality are predictable.

30. A downstream BI tool queries S3 data via Athena, but schema changes break dashboards. How would you design a robust data lake architecture to avoid this issue?

I would add a stable "contract layer" that BI reads from, and keep schema changes additive and validated before publish. I'd also use a lakehouse format to manage evolution and versioning.

Design I'd implement

- Zone separation: raw (immutable), curated (typed Parquet), and gold (BI-ready marts). BI queries gold only.
- Lakehouse tables for gold: store gold in Iceberg/Hudi/Delta so I can evolve schemas, do MERGE/DELETE, and roll back via time-travel if a change breaks a dashboard.
- Stable views for dashboards: expose BI to views that select a fixed set of columns and apply defaults (COALESCE) and masks. When schema expands, I add columns behind the scenes and update views only after tests pass.
- Additive schema evolution: new fields are nullable; no in-place renames/drops. Breaking changes create a v2 table; I run v1 and v2 in parallel until dashboards are migrated.
- Two-phase publish: write new partitions or snapshots to a staging location/table, run data quality checks (row counts, null rates, type checks) and schema diff, then atomically promote (swap partition locations or commit Iceberg snapshot). If validation fails, dashboards keep using the last good snapshot.
- Partitioning and file hygiene: Parquet with dt first and one more low-cardinality key; large files (256–512 MB) and scheduled compaction so Athena scans are small and consistent.
- Metadata control: manage Glue schemas via code (no freeform crawlers on gold). Use partition projection so new dates are instantly queryable; keep table parameters documenting owner, SLA, and schema version.
- Governance: put tables under Lake Formation; grant BI read access to views, not to underlying raw/curated tables. Deny direct S3 reads for BI roles.
- Monitoring and rollback: capture Athena failures, bytes scanned spikes, and BI query errors. If a deployment causes issues, use lakehouse time-travel to revert to the previous snapshot while we fix.
- Change management: treat schemas as code in Git; require PRs with a schema diff, sample run, and downstream test checks. Communicate planned additions with a deprecation window for any breaking changes.

Result: dashboards read from a stable, governed contract that evolves safely. New columns appear without breaking existing visuals, and if something slips through, we can roll back instantly while keeping the raw data and curated pipelines intact.

31. You need to join multiple large fact tables (each ~500 GB). What techniques would you use in Athena to optimize the join strategy?

I try hard to avoid "fact-to-fact" full scans. I shrink each side first, line up partitions, and then join the smallest possible slices.

- Reduce each fact before the join
    - Filter by partitions early (for example, dt, region/tenant). Make those filters part of the query's first CTE so Athena prunes folders before reading.
    - Project only the columns needed for the join and final output.
    - Pre-aggregate each fact to the grain you actually need (for example, per user_id per day), then join the aggregates instead of raw events.
- Align partitioning so pruning works on both sides
    - Store both facts as Parquet, partitioned the same way (dt first, then region/tenant).
    - In the query, filter both tables to the exact same dt and region ranges. When both sides are pruned similarly, the scan size drops a lot before any joining happens.
- Make the join key efficient
    - Ensure join columns have the same data type (no runtime casts on a 500-GB side).
    - If keys are skewed (a few values very hot), pre-split those keys into separate CTAS tables or add a "salt" column in pre-processing, then join on (key, salt) to avoid a single huge hotspot.
- Stage intermediate "build" tables
    - From fact A (after filters), CTAS the distinct join keys or a light aggregate (much smaller than 500 GB).
    - Join fact B only to that staged key set (semi-join reduction). This lets Athena skip chunks of fact B that don't match.
- Consider materializing recurring heavy joins
    - If this join powers many dashboards, create a daily CTAS (or Iceberg table) that pre-joins the filtered facts for the day and is partitioned by dt/region. Everyone queries that result instead of redoing the big join.
- File/layout hygiene
    - Keep Parquet files large (256–512 MB) and sorted within partitions by the join key when writing; better min/max stats help Athena skip row groups.
- Query structure tips
    - Use CTEs to apply filters first, then join the filtered CTEs.
    - Avoid cross joins and chained joins that multiply data early. Join the smallest pair first, then add the next table.

Result: each fact is reduced to a small, partition-aligned slice before joining; hot keys are handled; and for repeated use you materialize the join so day-to-day queries are fast and cheap.

32. A query joining user activity logs with multiple reference datasets runs very slowly. How would you use bucketing or partition alignment to improve performance?

I focus on two things that actually help in Athena: make both sides prune the same folders (partition alignment) and lay out Parquet so row groups can be skipped (sorting/compaction). Classic Hive "bucketing" metadata alone doesn't speed Athena much, so I treat it as optional. Here's exactly what I'd do.

First, prune before you join
I make both the activity logs and each reference dataset Parquet, and partition them the same way: date first, then a low-cardinality key like region or tenant. For example: .../dt=YYYY-MM-DD/region=APAC/. In the SQL, my first CTE filters all tables to the exact same dt and region window. That way Athena skips whole folders on every table before it even starts the join. This is the biggest win.

Second, shrink each side early
I select only the columns I need and pre-filter the reference tables (for example, only active rows or only current versions). If one reference is still big, I pre-stage a small "key set" from the logs (distinct user_id for that date/region) and semi-join the references to that key set first. That stops Athena from reading reference rows that don't match.

Third, lay out Parquet for predicate and join efficiency
When I write Parquet, I sort within each partition by the join key (user_id). Sorted row groups produce tight min/max stats, so Athena can skip row groups that don't contain the user ids on the other side. I also target 256–512 MB files per partition to reduce open/seek overhead.

Fourth, join order and broadcast
I join the large fact (logs) to the smallest reference first so Athena can broadcast the small table. Then I add the next smallest. I make sure join key data types match (no runtime casts on the big side), and I avoid exploding the row count early.

About bucketing
If I already have a Spark/Glue pipeline, I can write both datasets with the same bucketed_by = user_id and the same bucket_count. In Athena this helps mostly by creating repeatable, evenly distributed files; the real speed still comes from partition pruning and sorted Parquet. I don't rely on bucketing metadata alone to make joins faster.

Hot-key handling
If a few users are extremely hot, I add a tiny "salt" during write (for example, user_id#b00..b07) to spread those users across files and then join on (user_id, salt). This removes reducer hotspots.

If this query is frequent
I materialize it. I run a scheduled CTAS (or Iceberg INSERT INTO) to build user_activity_enriched partitioned by dt and region. Teams query that small, denormalized table instead of rerunning multi-way joins.

Result: the query reads only the aligned partitions, scans fewer Parquet row groups thanks to sorting, broadcasts tiny references, avoids skew on hot users, and—if it's common—hits a prebuilt table rather than rejoining every time.

33. Your analysts want to run a single query joining customer profiles from DynamoDB with purchase history in S3. How would you enable and optimize this?

I can do it two ways. For ad-hoc exploration I enable Athena federation to DynamoDB and join live to S3. For repeatable analytics I materialize a DynamoDB slice to S3 and do an S3-to-S3 join, which is much faster and cheaper. I usually give them both: federation for quick checks, and a curated S3 table for dashboards.

Option A: single federated query (quick to enable)

1. I deploy the Athena DynamoDB connector (Lambda) and create a federated data source.
2. I make sure the join uses DynamoDB keys so the connector can push down filters. For example, filter profiles by tenant_id and a small set of customer_ids.
3. In SQL I filter S3 by dt/region first, select only needed columns, and then join to the federated table on a like-typed key (no casts).
4. I keep result sets small. If the profiles table is large, I first CTAS a temporary key set from S3 (distinct customer_id for the date range) and then join DynamoDB to that set, so the connector pulls only matching rows.
5. I set timeouts/memory on the connector Lambda high enough for these selective reads and tag the workgroup so costs are attributed.

Pros: zero extra pipelines, great for one-offs. Cons: slow/expensive for big joins; Lambda timeouts; DynamoDB RCUs consumed.

Option B: materialize DynamoDB to S3 (best for BI and heavy joins)

1. I create an S3 Parquet table for customer profiles using one of:
   • DynamoDB Export to S3 (point-in-time), or
   • DynamoDB Streams → Firehose/Glue Streaming for near-real-time CDC, or
   • A short Glue job that reads DynamoDB in parallel (segmented scans) and writes Parquet.
   I partition by tenant_id (and maybe dt if I snapshot), and I store only the fields analysts need.
2. Now I join S3 Parquet to S3 Parquet in Athena. I align partitions (same tenant_id; purchase history is also partitioned by dt and tenant_id), and I sort Parquet within partitions by customer_id for better pruning.
3. If analysts run this a lot, I build a daily/hourly CTAS purchases_enriched that pre-joins profiles to the day's purchases and is partitioned by dt/tenant_id. All dashboards query this table; it's fast and cheap.
4. I schedule small incremental updates: refresh only the newest partitions, not the whole history. If I use Iceberg, I can MERGE changes from the CDC feed and keep snapshots for rollback.

Security and governance
I put both tables under Lake Formation and grant analysts SELECT on the curated and enriched tables (not raw DynamoDB). I deny direct S3 reads for those roles so policies can't be bypassed. All results and data are KMS-encrypted.

Performance checklist I follow

- Make sure customer_id types match across sources.
- Always filter S3 by dt and tenant_id first.
- Project just the handful of columns analysts need.
- Keep Parquet files 256–512 MB and compact if writers produce small ones.
- If I must stay federated, I keep DynamoDB reads selective and consider caching a small "profiles_latest" S3 table refreshed hourly to avoid hitting DynamoDB for everything.

Result: analysts can run a single query today via federation for exploration, and their production dashboards run on a fast S3-to-S3 join (or a pre-joined CTAS table) that's cheap, stable, and easy to maintain.

34. You're asked to replace a costly ETL pipeline with Athena federated queries across RDS + S3. What are the trade-offs, and when would you advise against it?

I can make this work for ad-hoc or light analytics, but I'm careful. Federated queries save pipeline cost and engineering time, yet they push compute to a Lambda connector and live against the OLTP, so they can be slow, brittle, and risky for production dashboards.

What I'd gain

- Less plumbing to build and maintain. No nightly export job, fewer moving parts.
- One SQL query can join fresh rows in RDS with big history in S3. Great for quick investigations and one-off analysis.
- Lower upfront cost. You pay per query and per Lambda execution instead of running an always-on ETL/cluster.

Hidden costs and limits

- Performance is limited by the connector. Athena pulls data through a Lambda function; large joins can hit Lambda memory/time limits. Wide scans or big shuffles are slow and may fail.
- OLTP load risk. Queries can put extra read load on RDS and interfere with transactional traffic. Even read replicas have finite IOPS/CPU.
- Limited pushdown and type quirks. Not all filters/joins push down nicely; you might end up moving lots of data over the network. Type mismatches cause casts on large sides and slow everything down.
- Operational complexity moves, not disappears. You now manage connector Lambda packaging, VPC networking, Secrets Manager creds, timeouts, and retries.
- Cost unpredictability. A few poorly written federated queries can rack up Lambda + RDS + Athena costs quickly.

When I'd advise against replacing ETL with federation

- Regular dashboards with strict SLAs. I don't want every morning's KPI to depend on live reads from RDS via Lambda.
- Heavy joins/aggregations. If we're joining hundreds of GB in S3 to millions of RDS rows repeatedly, it's cheaper and faster to stage RDS data in S3 Parquet and do S3-to-S3.
- Sensitive or regulated workloads. I prefer controlled, audited pipelines that land immutable copies in S3 with KMS and Lake Formation, not ad-hoc live access to OLTP.
- Tight cost control is required. Materializing once (CDC to S3) is more predictable than paying for repeated federated scans.

What I'd propose instead for production

- Keep federation for one-off analysis and prototyping.
- For recurring queries, build a small CDC path from RDS → S3 Parquet (DMS to S3 or Debezium/Kinesis). Partition by date/tenant, store as Iceberg/Hudi/Delta, and query in Athena with S3-to-S3 joins.
- Optionally snapshot a small "current dimensions" table daily/hourly into S3 so freshness is good without hitting RDS.

- If near-real-time is required, materialize just the hot slice (last few hours) frequently, not the whole DB.

This gives the best of both: federation for speed of iteration, and a governed, fast, and cheaper lakehouse path for daily analytics.

35. Your Athena query performance degrades suddenly, even though the dataset hasn't grown. How would you troubleshoot the root cause?

I run a quick, structured checklist: verify what changed in the query and environment, confirm file/layout health in S3, check metadata and partitions, and isolate which join or stage got slower.

Start with the basics

- Compare the exact SQL and workgroup. Did anyone add SELECT *, remove date filters, or change the workgroup (and thus the result location or settings)?
- Re-run with EXPLAIN to see if join order or broadcast behavior changed (for example, the "small" table isn't small anymore due to filters not applied).
- Check Athena workgroup metrics for the query: data scanned, execution time, and if "bytes scanned" jumped without data growth.

Validate partition pruning and metadata

- Ensure the WHERE still filters by the partition columns (dt, region). If a view changed and now computes date(col) on the partition key, pruning can break.
- If using partition projection, confirm parameters (ranges/enums) still include requested values and didn't fall back to scanning all.
- If using crawlers, make sure partitions are current; missing partitions can force scans of unexpected locations.

Inspect S3 layout and files

- Look at the affected table's latest partitions: did a recent ingestion start producing thousands of tiny files instead of 256–512 MB files? Small files increase open/seek overhead.
- Check Parquet stats: were new files written unsorted, so min/max no longer prune row groups?
- Confirm compression/codecs: did someone write GZIP CSV into a curated location by mistake?

Schema and types

- Make sure join keys are still the same type on both sides. A recent change to VARCHAR on one table will force casts on the big side and kill performance.
- Check for new, very wide columns added to views or CTAS outputs that inflate scan size even if you don't select them.

Joins and “small table” size

- Confirm filters on the dimension tables still apply before the join (for example, active = true). If a filter moved after a join, the “small” table isn’t small anymore, so broadcast may stop and a big shuffle happens.
- If using SELECT DISTINCT to get keys for a semi-join reduction, validate cardinality didn’t explode accidentally.

Engine and environment

- Look for regional service issues or throttle events in CloudWatch for Athena.
- Check S3 performance anomalies: high 5xx, throttling, or cross-region access. Ensure data and Athena are in the same region.

Fixes I apply based on findings

- Restore partition pruning: rewrite predicates to reference raw partition columns (no functions on them), or adjust partition projection settings.
- Reduce scan set: add back date/region filters, explicitly list columns instead of *, and push filters into CTEs before joins.
- Repair layout: run a quick compaction job on the hot partitions to merge tiny files; re-sort by common predicate/join key when writing Parquet to restore min/max pruning.
- Correct schema drift: align join key types; update Glue schema and views; cast at write time in ETL rather than at query time.
- Rethink join plan: pre-filter dims, broadcast the truly small side, or stage a distinct key set from the fact and semi-join dims first.
- Roll back a bad change: if a new CTAS/view release caused the regression, revert to the previous Iceberg snapshot or table version while we fix.

Outcome
By checking pruning, file health, schema/join behavior, and recent changes, I can identify the exact reason scans or shuffles grew and apply a targeted fix—usually restoring partition filters, compacting small files, or correcting a schema or view change.

36. A query runs successfully in Athena Workbench but fails when triggered via Lambda automation. How would you debug this issue?

First, I would check what's different between running in Workbench and running through Lambda. In Workbench, queries usually succeed because my user has the right permissions and defaults set. In Lambda, the execution happens through an IAM role, and that role might be missing permissions or some configurations.

The first thing I check is the IAM role attached to Lambda. It must have permissions for Athena queries, Glue Catalog access, and also permissions to read and write in the S3 bucket where Athena stores query results. If encryption is used (like KMS), I also make sure that role has KMS permissions.

Second, I look at the Athena workgroup and output location. In Workbench, I may be using one workgroup, but in Lambda I might be unknowingly using another. The workgroup defines things like the result S3 location, encryption, and query limits. If those don't match, the query can fail.

Third, I check if I'm passing the database name correctly in Lambda. In Workbench, I might have set the default database once, but Lambda needs me to pass it explicitly. If I don't, the query may fail saying "table not found."

Another thing I check is Lambda timeout. Athena queries can take a few minutes. If the Lambda function has a short timeout, it can stop even if Athena is still running the query. In that case, I either increase the Lambda timeout or use Step Functions to manage the wait.

Finally, I check the CloudWatch logs. Lambda logs and Athena query execution logs usually give me the exact reason — whether it's a permissions issue, output bucket issue, or timeout.

So in short, the most common causes are: missing IAM permissions, wrong workgroup or S3 output location, missing database parameter, or Lambda timing out. I debug step by step by checking IAM role, S3 bucket access, database/workgroup configuration, and logs.

37. You need to automate a pipeline where Athena queries raw S3 logs daily, applies transformations, and loads results into Redshift. How would you design this automation?

I would keep it simple and reliable.

First, I would make sure the raw logs in S3 are organized by date, like putting each day's logs into a separate folder. Then I would create a Glue Crawler or Glue Catalog table so Athena can read these logs easily.

Next, I would write an Athena query to transform the data. For example, I would clean fields, convert data types, and keep only the required columns. I would run this query daily for the new day's logs and store the results back in S3 in a clean format, like Parquet, which is faster and cheaper.

After that, I would load the transformed data into Redshift. The easiest way is to use Redshift COPY command from the Parquet files in S3. COPY is very fast and efficient. I would load it into a staging table first and then move the data into the final table so that I don't create duplicates.

For automation, I would use EventBridge or Glue triggers to run the pipeline every day. For example, EventBridge can trigger a Lambda function daily. The Lambda will:

1. Run the Athena query for yesterday's logs
2. Wait for query to finish
3. Run the Redshift COPY command to load data

For monitoring, I would set up CloudWatch alarms to notify me if the pipeline fails.

This way, the pipeline is simple: daily trigger → Athena query for transformation → store results in S3 → load into Redshift → monitor with CloudWatch.

38. A business unit wants self-service daily reports powered by Athena. How would you automate report generation while ensuring data freshness and cost control?

First, I would make sure the source data in S3 is always up to date and properly organized. For example, I would partition the data by date so Athena can quickly pick only the new data instead of scanning everything.

Next, I would use a Glue Crawler or define tables in the Glue Data Catalog so the business unit can easily query the data through Athena without worrying about file formats. If new data arrives daily, I would either schedule the crawler or use partition projection so that the schema is always fresh without needing too many crawler runs.

For automating the report, I would create a set of SQL queries in Athena that generate the required daily summaries. Then I would schedule these queries to run automatically every day. This can be done using EventBridge and a Lambda function. The Lambda can trigger Athena queries, wait until they finish, and then store the results in S3 in CSV or Parquet format.

Once the results are in S3, I would make them easily accessible. For example, I can connect Athena to QuickSight so the business unit can view dashboards without needing to run queries themselves. If they just need raw files, I can store them in a "reports" folder in S3, partitioned by date.

For data freshness, the pipeline will run daily at a fixed time, right after new data lands in S3. That way, the reports always reflect the latest data.

For cost control, I would:

- Partition the data properly (like by date or region) so queries scan less data.
- Convert data to Parquet format instead of CSV/JSON to reduce scan size and speed up queries.
- Create pre-aggregated daily tables so that business users don't repeatedly run heavy queries. They can just query the smaller report tables.
- Use Athena workgroups with query limits to control how much each business unit spends.

So in simple words, the solution is: organize and partition the data in S3 → keep Glue Catalog updated → schedule Athena queries daily with EventBridge + Lambda → store results in S3 or visualize in QuickSight → apply partitioning and Parquet for cost savings → use workgroups for budget control.

39. How would you optimize Athena queries to improve performance when queries on large datasets in S3 are taking too long and becoming costly?

The first thing I would do is reduce the amount of data Athena has to scan, because Athena pricing and speed depend directly on the amount of data read.

To achieve this, I would:

- Convert the data to columnar formats like Parquet or ORC instead of using raw CSV or JSON. These formats are compressed and allow Athena to only scan required columns.
- Partition the data in S3, usually by time (year, month, day) or another high-level filter that matches query patterns. This way, queries can skip unnecessary partitions.
- Use bucketing if there are common join keys to avoid scanning the full dataset during joins.
- Optimize file sizes. Very small files (like thousands of small JSON/CSV files) slow down queries, and very large single files can also cause imbalance. I would aim for files around 128MB–512MB in size.
- Select only the required columns instead of using SELECT *. This avoids scanning extra data.
- Use compression (like Snappy) which Athena can read natively.
- Make sure partitions are correctly registered in the Glue Catalog or use partition projection so Athena doesn't waste time scanning empty locations.
- Where possible, pre-aggregate or summarize data into smaller tables so that reporting queries don't always hit the full raw dataset.

In short, the key optimizations are: use Parquet/ORC, partition smartly, optimize file sizes, select only needed columns, and pre-aggregate when possible. This reduces both query time and cost.

40. You receive deeply nested JSON data in S3. How would you flatten and extract specific attributes using Athena?

Athena has good support for JSON, so I would start by creating an external table on top of the JSON files in S3. For nested JSON, I can either use a JSON SerDe in the Glue Catalog table definition or I can load the JSON as a single string column and then parse it using Athena functions.

If I just need specific attributes, I would use the built-in JSON functions like json_extract or json_extract_scalar to pull out values. For example:

```
SELECT
 json_extract_scalar(event, '$.user.id') AS user_id,
 json_extract_scalar(event, '$.user.country') AS country
FROM raw_json_table;
```

If the JSON has arrays or nested objects, I would use the UNNEST function in Athena. That lets me expand arrays into multiple rows and then pick the fields I need. For example, if each record has a list of items:

```
SELECT
 order_id,
 item.value:name AS item_name,
 item.value:price AS item_price
FROM orders
CROSS JOIN UNNEST(json_extract(order, '$.items')) AS t(item);
```

If the JSON structure is consistent, another option is to define the Glue Catalog table schema with the nested structure directly. Athena supports struct, array, and map data types, so I can query them with simple dot notation like user.id or items[1].name.

If the data is too complex or performance is an issue, I would run a one-time transformation job in Glue to flatten the JSON into a cleaner Parquet table. That makes daily queries much easier and faster.

So in short: if it's one-time or simple, use json_extract and UNNEST in Athena; if it's recurring and complex, flatten with Glue into Parquet and then query the flattened structure.

41. How would you enforce strict access controls and security for sensitive data queried through Athena?

For sensitive data, I would make sure access is restricted at multiple levels — storage, catalog, and query.

First, at the S3 layer, I would keep sensitive data in separate buckets or prefixes and enable encryption (SSE-KMS). I would write bucket policies so only specific IAM roles or users can access that location. This ensures no one can bypass Athena and directly read the files.

Second, at the Glue Data Catalog and Athena level, I would use AWS Lake Formation. Lake Formation allows me to define table-level, column-level, and even row-level permissions. For example, if some users should not see PII fields like email or phone, I can restrict access to those columns. Athena automatically enforces these policies.

Third, I would restrict Athena workgroups. Each team can have its own workgroup with limits, budgets, and separate output locations. This prevents data leaks in shared S3 output buckets. Output results also need to be encrypted, and the S3 result bucket should have access only for authorized users.

Fourth, for auditing and compliance, I would enable CloudTrail and CloudWatch logs to monitor who is querying what data. I would also enable query logging in Athena workgroups. If sensitive queries are run, I can track exactly who accessed what.

Fifth, I would make sure users access Athena through federated authentication (like SSO via IAM Identity Center) so that individual user identities are logged instead of everyone using one shared IAM role.

So, to summarize simply: enforce access on S3 with encryption, use Lake Formation for fine-grained permissions, secure output buckets, control workgroups, and enable full auditing and logging.

42. You need to join two large datasets in Athena (e.g., user profiles + transactions), but queries are very slow. How would you optimize the joins?

When two large datasets are joined in Athena, the main problem is that Athena has to scan both datasets fully, which becomes slow and costly. To optimize, I would try to reduce the amount of data scanned and make the join more efficient.

First, I would make sure both datasets are in columnar format like Parquet and properly compressed. This speeds up scans.

Second, I would partition both datasets by a common key if possible. For example, if both user profiles and transactions can be partitioned by region or date, Athena can prune partitions and scan less data during the join.

Third, I would check if one of the datasets is relatively small, like a few MBs or GBs. If that's the case, I can broadcast the smaller dataset by rewriting the query in a way that Athena loads it fully into memory before joining with the large dataset. In Spark this is called a broadcast join; in Athena, the same effect happens if the optimizer detects a small table.

Fourth, I would pre-aggregate or filter data before the join. For example, instead of joining the entire transactions table with all user profiles, I would filter only the date range or the user IDs I need first. That way, Athena scans less data.

Fifth, I could create a pre-joined or materialized table if the join is required frequently. For example, I can run a Glue ETL job once a day that joins profiles and transactions into a clean, optimized Parquet dataset. Then analysts query that directly instead of running heavy joins every time.

In summary: convert to Parquet, partition smartly, reduce dataset size with filters, use broadcast joins when possible, and precompute joined tables if the join is used often.

43. How would you automate daily Athena query execution and store results in S3 for reporting and downstream analysis?

I would use a simple automation flow. First, I would write the Athena SQL query that produces the required results. Then, I would use Amazon EventBridge to schedule this query execution daily. EventBridge can trigger a Lambda function at a fixed time.

The Lambda function would call Athena's StartQueryExecution API, pass the SQL query, and specify the output location in S3 where the results should be stored. Once the query finishes, Athena automatically writes the results (in CSV or Parquet) into the S3 bucket. If I want, I can also move or rename the file with Lambda to store it in a "reports/daily" folder organized by date.

For downstream analysis, other systems can directly consume these result files in S3. If needed, I can register this reports folder in the Glue Data Catalog so users can query the report table easily without re-running the heavy query.

For monitoring, I would check CloudWatch logs from Lambda and Athena query execution states. If a query fails, I can send an SNS notification to the team.

So in short: EventBridge schedules → Lambda triggers Athena query → results stored in S3 → optionally cataloged for reuse → monitoring with CloudWatch and SNS.

44. A new data source introduces additional fields to your dataset in S3, causing schema mismatches in Athena. How would you handle schema evolution?

Athena depends on the schema defined in the Glue Data Catalog. If the incoming data has new fields, I would handle it in one of two ways.

If the new fields are not immediately needed, I can just ignore them because Athena is schema-on-read. Athena will scan the file but only use the columns defined in the table.

If I do need the new fields, I would update the Glue Catalog table definition. This can be done by running an ALTER TABLE ADD COLUMNS statement in Athena, or by re-running a Glue Crawler so it detects the new fields automatically. The new columns will show up as nullable, and older data without those fields will just return null.

For frequent schema changes, I would prefer using a format like Parquet or ORC with schema evolution support, because they handle new columns more gracefully. I would also avoid hardcoding strict schemas in downstream systems, so they don't break when new fields arrive.

So in short: Athena is schema-on-read, so mismatches don't break everything. If new fields are needed, I alter the table or run a crawler to add them. For long-term stability, I use Parquet/ORC and plan for optional columns.

45. An Athena query fails with the error HIVE_BAD_DATA due to data parsing issues. What steps would you take to debug and resolve the failure?

This error usually means that the file contents don't match the table schema. To debug, I would first check which file caused the issue. Athena query execution logs usually show the file path.

Once I know the file, I would download a small sample and manually inspect it. I would check if the delimiters are correct (for CSV/TSV), if quotes are consistent, or if some rows have extra/missing columns. For JSON, I would check for malformed JSON or inconsistent nesting.

I would also verify that the table schema in the Glue Catalog matches the actual data format. For example, if a column is defined as bigint but the file has string values, Athena can throw this error.

If the dataset is large, I might run a query with LIMIT 10 or filter by date to narrow down which partitions are bad. Sometimes only one file is corrupted.

To fix the issue, I can:

- Correct the file format at the source if possible
- Adjust the Athena table schema to match the data correctly (e.g., change column type from int to string)
- Use functions like try_cast() to safely convert bad values instead of failing
- Move or delete corrupted files if they're unusable

So in short: I debug by checking the logs for the bad file, inspecting the data, comparing it to the table schema, and then either correcting the data, adjusting the schema, or using safe casting functions.