AWS EMR

11 July 2025 19:00

- **EC2 is the car.** It's the engine, the chassis, the wheels – the raw power and physical (virtual) structure that performs the actual movement. It's the underlying hardware that does the work.
- **EMR is the driver.** The driver doesn't *build* the car or *do the driving* in the sense of physically moving the pistons. Instead, the driver's role is to:
    - **Know how to operate the car:** EMR knows how to install, configure, and optimize big data frameworks (Spark, Hadoop, etc.) on the EC2 instances.
    - **Navigate and plan the route:** EMR orchestrates your big data jobs, distributing tasks across the EC2 nodes, managing resources (like YARN), and ensuring the job gets done efficiently.
    - **Handle maintenance:** EMR monitors the EC2 instances and the applications running on them, replacing failed "parts" (EC2 instances) if needed to keep the "journey" (data processing) going.
    - **Scale the journey:** The driver can decide to bring in more cars (add more EC2 instances) for a bigger convoy, or reduce them, based on the workload.
    - **Focus on the destination:** As the user, you tell EMR what big data job you want to achieve (the "destination"), and EMR (the driver) takes care of all the complex driving mechanics and route planning using the EC2 "cars."

  In AWS terms: You configure your EMR cluster: You choose the EC2 **instance types** (e.g., m5.xlarge for general-purpose, r5.2xlarge for memory-intensive) and specify the **number of nodes** you need for your master, core, and task groups. You also tell EMR which big data applications (Spark, Hadoop, Hive) you want pre-installed.
  In AWS terms: EMR sends requests to the EC2 service to launch the specified number and type of EC2 instances. These instances are *brand new, virtual servers* provisioned just for your cluster.
  In AWS terms: On the **Master Node EC2 instance**, EMR installs the master components of your chosen big data applications (e.g., Spark Master, Hadoop NameNode, YARN ResourceManager) and other cluster management software.
  In AWS terms: On the **Core Node EC2 instances**, EMR installs data storage components (e.g., HDFS DataNodes) and task-execution components (e.g., Spark Executors, YARN NodeManagers).
  In AWS terms: On the **Task Node EC2 instances**, EMR installs only task-execution components. These are often used with EC2 Spot Instances for cost savings.
  In AWS terms: EMR configures the networking and security groups so all the EC2 instances can communicate with each other, forming a secure and functional cluster. It also sets up distributed file systems like HDFS if used.

Intro - My Speech

Thursday, July 31, 2025 5:26 PM

Good afternoon! My name is [Your Name], and I'm excited to speak with you today.

My **background** is in building and optimizing large-scale data and machine learning platforms on AWS, with a **strong focus on delivering tangible business value**. Across my recent projects, I've operated as a **highly independent contributor**, taking ownership from **design to deployment**.

For instance, at a leading US fast-food company, I spearheaded the migration of their entire data ecosystem to an AWS Enterprise Data Warehouse. This involved designing robust data pipelines using PySpark, EMR, Redshift, and AWS orchestration services like Step Functions and Lambda, ultimately providing a single source of truth and streamlining their KPIs.

Following that, at a prominent US hospitality company, I led the migration of over 70 machine learning models from Hortonworks to AWS SageMaker. Here, I established end-to-end MLOps pipelines using SageMaker Pipelines and a robust CI/CD system with Terraform, ensuring automated, continuous, and reliable model deployments.

In both roles, my focus has been on architecting scalable, cost-efficient, and secure solutions, driving projects autonomously, and collaborating closely with business stakeholders to translate complex requirements into effective technical solutions.

I'm keen to learn more about the challenges you're facing and how my experience can contribute to your team's success. Thank you."

1st Project

Thursday, July 31, 2025 5:27 PM

The pipeline I built is an automated, end-to-end data flow using a serverless-first approach on AWS.

Part 1: The Foundation - Data Ingestion
To handle continuous data ingestion from the source SQL Server, I implemented AWS Database Migration Service (DMS). This was a critical design decision because DMS's Change Data Capture (CDC) functionality allowed us to ingest only the changes in the source database—inserts, updates, and deletes—rather than performing costly full loads. I configured this to stream directly to an S3 raw layer in Apache Parquet format. To keep costs low, I automated the DMS task using EventBridge and Lambda based on our specific latency requirements.

Part 2: The Core - Data Transformation
For the data transformation, my goal was to process the raw data efficiently. I designed a state machine in AWS Step Functions to orchestrate this entire phase. I chose EMR Serverless as the compute engine, which was a key driver for cost savings, as it automatically scales up and down with the workload. The transformation logic was written in PySpark and Python scripts, which I developed and deployed to run on this serverless environment.

Crucially, I integrated Delta Lake at this stage. This allowed me to use the powerful MERGE operation, which intelligently applied the CDC changes from the raw data to the processed data layer, ensuring data consistency. The use of Delta Lake also provided essential features like schema enforcement, which improved data quality, and time travel, which was invaluable for auditing and debugging.

Part 3: The Destination - Data Warehouse
After the data was refined and stored in the S3 processed layer, the final step was loading it into Amazon Redshift. I created a Lambda function, triggered by Step Functions, to handle this. This function used the Redshift Data API to dynamically manage our tables and leverage the efficient COPY command to bulk load the data. This made the data readily available for downstream business intelligence and reporting.

Leadership and Operational Excellence
A major part of my contribution was building in operational excellence. I designed the Step Functions workflow to have robust error handling, triggering an SNS notification with detailed logs whenever a failure occurred. A similar notification was configured for a successful run. This proactive monitoring and notification system, along with the cost-optimization from EMR Serverless, ensured the pipeline was not just technically sound but also reliable and financially responsible.

Thank you for your time.

2nd project

Thursday, July 31, 2025 5:27 PM

"Certainly, let's dive into the second project. This was a critical **MLOps re-platforming initiative** for a **leading US Hospitality Company**. The core objective was to modernize their machine learning operations by migrating over 70 existing ML models from a legacy **Hortonworks environment (where operationalization often involved EMR-like custom processes)** to a fully automated AWS SageMaker ecosystem.

Project Context & Challenges:

The client's 70+ ML models were operational on an on-premises Hortonworks cluster. While functional, their existing setup lacked a unified, automated MLOps framework. This led to significant inefficiencies:

- **Fragmented & Inconsistent Operationalization:** Model deployments, though not entirely manual, were inconsistent, relying on disparate custom scripts and processes on their EMR-like environment, causing bottlenecks and errors.
- **Absence of Continuous Automation:** There was no integrated, automated pipeline for model training, evaluation, or continuous deployment, making model updates slow, risky, and resource-intensive.
- **Scalability & Monitoring Limitations:** The on-premises infrastructure struggled to scale for rapid model growth and lacked robust, integrated monitoring for performance and health.
- **Weak Governance:** Centralized version control and metadata management for models and artifacts were challenging.

The objective was clear: **standardize, automate, and scale their entire ML lifecycle on AWS SageMaker**, thereby ensuring continuous, reliable model updates and robust governance.

My Role and Approach:

As a highly independent contributor, I took complete ownership of designing, building, and implementing this new, comprehensive MLOps framework on AWS. My strategy was to establish a scalable, automated, and governed MLOps pipeline using AWS SageMaker Pipelines, deeply integrated with a robust CI/CD system, to completely streamline and **replatform** their model lifecycle from the ground up.

Detailed Workflow and Solution Delivered:

Let me walk you through the end-to-end MLOps workflow we established:

1. **Model & Data Migration (Hortonworks to AWS S3 & ECR):**
    - **Challenge:** Migrating 70+ diverse ML models (varying algorithms, frameworks, dependencies) and their associated data from a proprietary Hortonworks environment to AWS.
    - **Solution:**
        - We first migrated all model artifacts and training data to **AWS S3**, establishing it as the central, versioned, and secure data lake for ML assets.
        - For each model, we developed a standardized process to **containerize** its inference code and dependencies into **Docker images**, stored in **Amazon ECR**. This provided a portable and consistent deployment unit for SageMaker.
        - **Python scripts** were critical in automating the bulk extraction from Hortonworks and orchestrating the secure upload to S3 and ECR.
    - **Independent Contribution:** I spearheaded the design of the S3 data lake structure for ML artifacts, established ECR repositories, and developed the Python automation scripts for efficient and secure bulk migration.
2. **End-to-End MLOps Pipeline with AWS SageMaker Pipelines:**
    - **Challenge:** Automating the entire ML lifecycle—from data preparation to model registration—for over 70 models, each with potentially unique steps and dependencies.
    - **Solution:** We built a generic, parameterized **AWS SageMaker Pipeline** that served as a reusable template for each model's lifecycle:
        - **Data Preprocessing:** A **SageMaker Processing Job** (using PySpark/scikit-learn containers) would read raw data from S3, perform feature engineering, cleaning,

and create train/validation/test splits.
        - **Model Training:** A **SageMaker Training Job** then trained the specific ML model, outputting the trained artifact to S3.
        - **Model Evaluation:** Another **SageMaker Processing Job** evaluated the model against the test set, generating critical performance metrics.
        - **Conditional Deployment Gate:** A conditional step evaluated these metrics (e.g., accuracy thresholds) to determine if the model qualified for registration.
        - **Model Registration:** Approved models, along with their metrics and metadata (e.g., training data version, hyperparameters), were then registered in **AWS SageMaker Model Registry** for centralized versioning and governance. We utilized **SQL queries** for efficient metadata access.
    - **Independent Contribution:** I was responsible for architecting and implementing the SageMaker Pipeline templates in Python, creating reusable components, defining step dependencies, configuring compute resources, and integrating evaluation metrics to control the deployment flow.
3. **CI/CD for Continuous Model Deployment (GitLab, Stash, Terraform):**
    - **Challenge:** Ensuring automated, continuous, and reliable deployment of new or updated model versions from the Model Registry to production, with full traceability.
    - **Solution:** We established a robust CI/CD system:
        - **GitLab/Stash (Source Control):** All model code, SageMaker Pipeline definitions, and infrastructure code (Terraform) were meticulously version-controlled.
        - **GitLab CI/CD Pipelines:**
            - **CI (Continuous Integration):** Triggered by code commits, these pipelines ran unit tests, linting, security scans on Python code, and built/pushed Docker images to ECR.
            - **CD (Continuous Delivery/Deployment):** Once a model was successfully registered in SageMaker Model Registry, a separate CD pipeline was automatically triggered.
        - **Terraform (Infrastructure as Code):** Terraform was used to manage all underlying AWS infrastructure for model deployments. The CD pipeline would:
            - Read the latest approved model version from SageMaker Model Registry.
            - Use Terraform to provision or update **SageMaker Endpoints** (for real-time inference) or **Batch Transform Jobs** (for offline predictions), including managing configurations, instance types, and auto-scaling.
            - This ensured consistent and reproducible infrastructure provisioning.
    - **Independent Contribution:** I designed and implemented the entire CI/CD workflow, authoring the GitLab CI/CD pipelines and developing the Terraform modules for SageMaker endpoint deployments. I also established the branching and review strategies for code quality.
4. **Monitoring & Governance (CloudWatch, SageMaker Model Monitor):**
    - **Challenge:** Continuously monitoring the operational health and performance of 70+ deployed models to promptly detect degradation, data drift, or operational issues.
    - **Solution:**
        - **AWS CloudWatch:** Integrated for comprehensive monitoring of operational health (endpoint latency, error rates, resource utilization) and custom model performance metrics.
        - **AWS SageMaker Model Monitor:** Configured to continuously analyze incoming inference data for **data drift and model quality drift**, automatically triggering alerts upon anomaly detection.
        - **Alerting:** CloudWatch alarms were configured to send **SNS notifications** to relevant teams (data scientists, MLOps engineers) for any critical issues.
    - **Independent Contribution:** I designed and implemented the complete monitoring strategy, configuring CloudWatch dashboards, setting up SageMaker Model Monitor, defining custom metrics, and integrating SNS for real-time alerting. I collaborated with data scientists to establish performance thresholds.

Outcome & Impact:

This project was truly transformative. We successfully **replatformed** all 70+ models, moving from

inconsistent, custom-scripted operationalization to a fully automated, scalable, and governed MLOps framework on AWS. This dramatically reduced deployment times from days to hours, significantly improved model reliability, enabled faster iteration on new models, and provided transparent visibility into model performance in production, directly supporting the hospitality company's data-driven decision-making and innovation."

Thursday, July 31, 2025 5:31 PM

Top 50 Question

General & Cross-Project (High-Level / Architectural)

1. **Why AWS?** Beyond the client's existing setup, why was AWS a strong choice for both an EDW and MLOps platform compared to other cloud providers or continued on-premise solutions?
2. **Architectural Evolution:** How did your architectural design for these platforms evolve over time based on new requirements or lessons learned?
3. **Scalability:** What were your primary considerations for ensuring both data ingestion and processing, and ML model inference, could scale effectively to handle future growth?
4. **Resilience/Fault Tolerance:** How did you design for resilience and fault tolerance across both projects to minimize downtime? Provide specific AWS service examples.
5. **Security Best Practices:** Beyond basic IAM, what advanced security measures did you implement across both projects to protect sensitive data and ML models?
6. **Cost Optimization Philosophy:** Describe your overall philosophy for cost optimization on AWS. How do you balance cost with performance and reliability?
7. **CI/CD Across Projects:** While explicitly mentioned in Project 2, how much of a CI/CD mindset did you bring to Project 1's data pipeline development, even if not fully automated?
8. **Monitoring Strategy:** Detail your end-to-end monitoring strategy for both data pipelines and ML models. What tools and metrics were most critical for you as an independent contributor?
9. **Your Contribution Style:** As an independent contributor, how do you ensure thoroughness, documentation, and knowledge transfer, even without a direct team to hand off to?
10. **Problem-Solving:** Describe a significant technical problem you encountered in either project that required an innovative solution. How did you approach it independently?

Project 1: Enterprise Data Warehouse (Fast-Food Company)

Service Choice & "Why"

11. **Redshift:** Why was Redshift chosen over other AWS data warehousing options like Aurora, Athena, or even Snowflake (if considered)? What specific features made it suitable?
12. **EMR:** Why EMR for PySpark over AWS Glue or a standalone EC2 cluster? What were the deciding factors?
13. **AWS DMS:** Why DMS for data migration and CDC instead of custom Python scripts or other ETL tools? What were its key advantages for your specific needs?
14. **Step Functions:** Why Step Functions for orchestration over Apache Airflow on EC2 or AWS Managed Workflows for Apache Airflow (MWAA)?

Architecture & Flow

15. **Data Lake Layers:** Describe the different layers (raw, processed, curated) you implemented in S3 for the EDW. How did data flow between them, and what was the purpose of each?
16. **Ingestion Flow:** Walk me through the detailed flow of data from source systems through DMS into S3 for both one-time and incremental loads.
17. **Transformation Flow:** Explain the data flow from S3 raw to S3 processed/curated using EMR/PySpark. How did you manage temporary data and intermediate outputs?
18. **Redshift Loading Flow:** Describe the exact steps and services involved in loading the final consumable data from S3 into Redshift.

Performance

19. **Redshift Query Performance:** What were your top 3 strategies to optimize Redshift query performance for BI dashboards (beyond just VACUUM/ANALYZE)?
20. **EMR Job Performance:** How did you identify and resolve performance bottlenecks in your PySpark jobs on EMR? Give an example.
21. **DMS Performance:** How did you optimize DMS replication task performance for large datasets or high CDC throughput?
22. **Pipeline Latency:** How did you minimize latency across the entire data pipeline from source to

Redshift for KPI reporting?

Cost Optimization

23. **Redshift Cost:** Beyond instance type, how did you optimize the cost of your Redshift cluster (e.g., WLM, concurrency scaling, data distribution)?
24. **EMR Cost:** How did you manage EMR cluster costs effectively? (Think instance types, Spot Instances, cluster lifecycle).
25. **S3 Cost:** What S3 lifecycle policies did you implement for cost savings on the data lake?

Contribution & "How"

26. **KPI Consolidation:** "You collaborated with business teams to define and consolidate KPIs." How did you technically enable this consolidation? What was *your* specific contribution to ensuring these KPIs were accurately represented in the EDW?
27. **DMS Configuration:** "Extracted data using AWS DMS." Detail *how* you configured DMS replication instances, endpoints, and tasks, including any specific settings for performance or error handling.
28. **PySpark Scripting:** Describe a complex data transformation you designed and implemented using PySpark on EMR. How did you structure the code for reusability and testing?
29. **Step Function Design:** Walk through the design of one of your more complex Step Functions workflows. How did you handle branching, parallelism, and error states within it?
30. **Self-Service Reporting:** "Enabling self-service reporting." What were the specific technical steps *you* took in the EDW design and data presentation to enable this for Power BI users?

Project 2: ML Model Migration & MLOps (Hospitality Company)

Service Choice & "Why"

31. **SageMaker:** Why was SageMaker chosen as the primary platform for MLOps over building custom ML infrastructure on EC2 or Kubernetes?
32. **SageMaker Pipelines:** Why SageMaker Pipelines specifically for workflow orchestration over Step Functions or Airflow in this MLOps context? What advantages did it offer?
33. **Terraform:** Why Terraform for IaC over AWS CloudFormation or CDK in your CI/CD setup?
34. **S3 for ML Data:** What were the specific advantages of using S3 for secure and scalable *ML data storage* (training data, model artifacts, inference logs)?

Architecture & Flow

35. **MLOps End-to-End Flow:** Describe the complete automated flow of a single ML model from code commit/data update through training, evaluation, registration, and deployment.
36. **Model Registry Integration:** How did SageMaker Model Registry integrate into your CI/CD and deployment flow? What role did it play in model governance?
37. **CI/CD Stages:** Detail the different stages of your CI/CD pipeline (e.g., build, test, deploy). What specific actions happened at each stage relevant to ML models?
38. **Monitoring Feedback Loop:** Explain how CloudWatch and Model Monitor formed a feedback loop for continuous model improvement and operational health.

Performance

39. **SageMaker Training Performance:** How did you optimize the training performance of the 70+ models on SageMaker (e.g., instance types, distributed training, data access)?
40. **SageMaker Endpoint Latency:** What measures did you take to ensure low-latency inference for real-time SageMaker endpoints?
41. **CI/CD Speed:** How did you optimize the execution speed of your CI/CD pipelines for faster model iterations?
42. **Data Processing for ML:** How did you ensure efficient processing of large analytical datasets from Redshift and S3 for ML model training, particularly considering memory and core utilization?

Cost Optimization

43. **SageMaker Cost:** What specific strategies did you employ to control costs for SageMaker training jobs and inference endpoints (e.g., Spot instances, auto-scaling, instance types)?
44. **S3 Storage Cost for ML:** How did you manage S3 storage costs for large volumes of training data and model artifacts?
45. **CI/CD Tooling Cost:** How did you ensure your GitLab/Stash and related CI/CD tooling were cost-efficient on AWS?

Contribution & "How"

46. **Migration Strategy:** "Successfully migrated 70+ existing ML models." Detail *your* strategy for

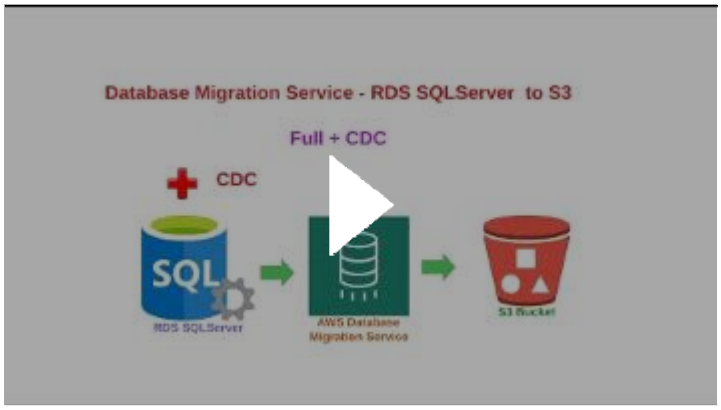
migrating such a large number of models. How did you standardize the migration process for diverse models?

47. **SageMaker Pipeline Design:** Describe *how* you designed a reusable SageMaker Pipeline template that could be applied across multiple models. What parameters did you expose?
48. **CI/CD Implementation:** "Established a robust CI/CD system leveraging GitLab and Stash for source control and Terraform for infrastructure as code." Walk through *your* specific steps in implementing this CI/CD system. What were the biggest challenges?
49. **Model Monitoring Implementation:** "Integrated AWS CloudWatch for continuous monitoring of model performance." Detail *how* you implemented this. What custom metrics did *you* define and capture for ML model performance?
50. **Terraform for MLOps:** Give a specific example of an AWS resource (e.g., SageMaker Endpoint, Model Package Group) that *you* provisioned and managed using Terraform within this MLOps context. Explain *how* your Terraform code was structured.

Thursday, July 31, 2025 5:32 PM

AWS SQL SERVER to DMS to S3 to Redshift using AWS EMR + Step Function + Lambda + Event bridge

AWS Database Migration Service (DMS) SQLServer to S3 bucket - Full load + CDC



1. Data Ingestion (SQL Server to S3 Raw):

- Source: The pipeline starts with an on-premises Microsoft SQL Server database.
- AWS DMS CDC: AWS Database Migration Service (DMS) ingests data. It connects to the SQL Server and uses its transaction logs to capture all DML operations: inserts ('I'), updates ('U'), and deletes ('D'). AWS DMS names CDC files using timestamps, for example `20141029-1134010000.csv`.
- S3 Landing (Raw Layer): The captured changes, along with an operation indicator (e.g., 'I', 'U', 'D'), stream to a dedicated S3 "raw" bucket, typically in Apache Parquet format for efficient storage and query performance.
- Scheduling: The DMS task can be scheduled to run periodically or continuously using EventBridge and Lambda functions, optimizing based on latency requirements and cost.

2. Data Processing and Refinement (S3 Raw to S3 Processed):

- Orchestration with Step Functions: AWS Step Functions orchestrates this stage, triggering the necessary actions sequentially.
- EMR Serverless: A Step Functions task invokes EMR Serverless, a cost-optimized option that automatically scales compute resources based on workload needs. This eliminates the need to manage clusters manually.
- Script Execution: PySpark and Python scripts, stored in S3, are submitted as jobs to the EMR Serverless application. These scripts are executed within the EMR Serverless application environment.
- Delta Lake & `MERGE` Operation: The Spark applications running on EMR Serverless read the raw Parquet files from the S3 "raw" bucket. They then utilize Delta Lake's

features:

- `MERGE` Operation: The `MERGE` statement matches the incoming CDC records with the target Delta table and applies inserts, updates, or deletes.
- Schema Enforcement and Evolution: Ensures data quality by validating the schema of incoming data and automatically handling schema changes to maintain data consistency.
- Time Travel: The Delta Lake format maintains a transaction log, enabling "time travel" to query previous versions of the data. This is invaluable for auditing, debugging, and historical analysis.
- S3 Processed Layer: The processed and refined data is written to the S3 "processed" bucket in Delta Lake format, creating a curated data layer.

3. Data Warehousing (S3 Processed to Redshift):

- Redshift Table Management: After Spark processing, a Lambda function (triggered by AWS Step Functions) uses the Amazon Redshift Data API to manage the Redshift tables. It can dynamically create or modify table schemas to accommodate changes propagated from the Delta Lake layer.
- Data Loading: The refined data from the S3 "processed" bucket is loaded into Amazon Redshift tables. Spark applications or Lambda functions can facilitate this, often leveraging Redshift's `COPY` command for efficient bulk loading.
- Analytics: The data in Amazon Redshift is now available for analytical querying, business intelligence, and reporting.

4. Orchestration, Monitoring, and Notifications:

- Step Functions Orchestration: AWS Step Functions defines the entire workflow, including submitting jobs, waiting for their completion, and transitioning between steps.
- Error Handling and Notifications: If any step encounters an error, AWS Step Functions' error handling mechanisms trigger a Lambda function. This function retrieves error logs and publishes a detailed failure notification to an SNS topic.
- Success Notification: Upon successful completion of all steps, a Lambda function is triggered to send a success email notification to the SNS topic.
- Cluster Termination: A final Step Functions task ensures the EMR cluster is terminated, especially when using ephemeral clusters, to minimize running costs. This step also includes error handling for the termination process itself.
- Cost Optimization: The use of EMR Serverless for Spark processing and the automated termination of the cluster (when using ephemeral EMR) significantly reduce operational costs.

This pipeline provides a robust, automated, scalable, and cost-efficient solution for leveraging the power of cloud data processing and warehousing.

From <https://www.google.com/search?q=aws+dms+cdc+vs+spark+delta+package&sca_esv=fd80f86e9ec25f50&rlz=1C1GCEA_en-GBIN1171IN1171&ei=PICLaP_fM7CX4-EP4L-T2Ak&ved=2ahUKEwjO7uLzqeeOAxXK2TgGHeuuFVAQ0NsOegQINRAA&uact=5&sclient=gws-wiz-serp&udm=50&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZud1z6kQpMfoEdCJxnpm_3YlUqOpj4OTU_HmqxOd8LCYAmZcz3xp4-

s3ijYzIP40Lia9Q6yhIX2F8k3-TPnVLuf4vo2DKzxtKYVdS0l2dcByStNDjWIjl6lKjlPBgbneCbCrk0IwpQPyA9hdoscDz59XMWdfVb6wSUXBGtP5Q-cNQivZYhjaBoEniYvUm99WFRRAX0Q&aep=10&ntc=1&mstk=AUtExfCnlWIcoA8naZXw-cmfnL1vIdZAl6vdtLSr90unb8U1tvD3idyclylG7CCevAXqPIORlGyu6NSwrnJIsr_V45HlRaE7AHpKeKCPyPm4yCnRIu6Xt7dfZfCrFwO81dHBRmINd_1WgecJu3caQX_MCm_WrFHh-Z-XURoRmuAXMA8YpyWWUQNFSDJGc3gNPKnkYijnDTYYMeU_R1rdrC3RxcqczYcsvdCs_Y3ESnKREc-HBxsqWGmsyJavdg&csuir=1>

Clear Explanation of Delta table packages

Thursday, July 31, 2025 8:48 PM

1. DMS CDC to S3 Raw Bucket

- Change Capture: AWS DMS captures inserts, updates, and deletes by reading transaction logs from the SQL Server source.
- S3 Output: AWS DMS writes these changes to the S3 "raw" bucket as files, typically in Parquet format.
- Operational Indicators: AWS DMS includes an operation indicator (e.g., 'I' for insert, 'U' for update, 'D' for delete) as the first field in each record of the output files. This indicator is essential for Delta Lake to understand how to apply the changes.
- Daily Partitioning: AWS DMS can partition data in S3 based on a date-based folder structure, such as `s3://your-raw-bucket/table_name/year=YYYY/month=MM/day=DD/`. This helps organize data and improve query performance for specific date ranges.
- Transaction Order: By default, changes in AWS DMS are not logged in transaction order. This can be configured in AWS DMS using S3 endpoint settings.

2. Spark Delta Lake and the Curated Layer

- EMR Serverless Execution: Your PySpark scripts, stored in S3, run on an EMR Serverless application. EMR Serverless manages and scales the compute resources, reducing operational overhead and optimizing costs.
- Reading Raw Data: The Spark application reads the daily partitioned Parquet files from the S3 "raw" bucket, using the folder structure as partitions for efficient reading.
- Interpreting Operational Indicators: Spark processes the data, reading each record and extracting the operation indicator ('I', 'U', or 'D').
- Creating the Curated Delta Table:
- Initial Load: For the initial load, Spark can read the existing full load data (if any) and write it as a new Delta Lake table in the S3 "processed" bucket.
- Incremental Loads: The Spark application reads the new CDC files (representing a day's changes) and uses Delta Lake's `MERGE` operation to apply those changes to the existing Delta Lake table in the S3 "processed" bucket.
- Curated Delta Folder Structure: Delta Lake uses its own directory structure, including data files (Parquet) and a transaction log (`_delta_log` directory), to maintain the table's state. The S3 path for your curated Delta table would look something like `s3://your-processed-bucket/curated_database/curated_table/`.

3. How the Delta Lake MERGE operation works

The `MERGE` operation combines data from a source (your incoming CDC data) into a target (your existing Delta Lake table) in a single atomic transaction.

Here's how it works with DMS CDC data:

- Defining the Match Condition: The Spark code defines a condition to match rows between the incoming CDC data (source) and the existing Delta table (target). This is typically based on the primary key or a unique identifier of your data (e.g., `target.id = source.id`).
- Handling Inserts (`WHEN NOT MATCHED THEN INSERT`):
- If a record in the source (DMS CDC data) does not have a matching primary key in the target Delta table, and its operation indicator is 'I' (insert), the `MERGE` operation inserts that new record into the target table.
- Handling Updates (`WHEN MATCHED THEN UPDATE`):
- If a record in the source (DMS CDC data) has a matching primary key in the target Delta table, and its operation indicator is 'U' (update), the `MERGE` operation updates the specified columns of the matched row in the target table with the new values from the source.
- Handling Deletes (`WHEN MATCHED THEN DELETE`):
- If a record in the source (DMS CDC data) has a matching primary key in the target Delta table, and its operation indicator is 'D' (delete), the `MERGE` operation deletes the corresponding row from the target table.
- Atomic Operation: The entire `MERGE` operation is atomic, which means it either fully completes all changes or rolls back if an error occurs. This ensures data consistency and reliability in your Delta Lake table.
- Performance: Delta Lake's indexing, data skipping, and optimizations (like low shuffle merge) help improve the performance of `MERGE` operations, especially with large datasets. You can further optimize by reducing the search space in your match conditions and compacting small files.

Script execution flow

1. AWS Step Functions orchestrates the process and ensures that the steps of your data pipeline are executed in the correct order.
2. It triggers the EMR Serverless application to run a Spark job, submitting the PySpark script (stored in S3) as a step.
3. The Spark job reads the daily partitioned raw data from S3.
4. It identifies the 'I', 'U', 'D' indicators within each record.
5. Based on these indicators, it constructs and executes the `MERGE` operation against the target Delta Lake table in the S3 "processed" bucket.
6. Upon completion, AWS Step Functions proceeds to the next steps, including loading into Redshift and sending notifications.

This detailed explanation provides a clear understanding of how DMS CDC, Spark, and Delta Lake work together in your data pipeline to create a reliable and up-to-date curated data layer in S3.

From <https://www.google.com/search?q=aws+dms+cdc+vs+spark+delta+package&sca_esv=fd80f86e9ec25f50&rlz=1C1GCEA_en-GBIN1171IN1171&ei=PICLaP_fM7CX4-EP4L-T2Ak&ved=2ahUKEwjO7uLzqeeOAxXK2TgGHeuuFVAQ0NsOegQINRAA&uact=5&sclient=gws-wiz-serp&udm=50&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZud1z6kQpMfoEdCJxnpm_3YlUqOpj4OTU_HmqxOd8LCYAmZcz3xp4-s3ijYzIP40Lia9Q6yhIX2F8k3-TPnVLuf4vo2DKzxtKYVdS0l2dcByStNDjWIjl6lKjlPBgbneCbCrk0IwpQPyA9hdoscDz59XMWdfVb6wSUXBGtP5Q-cNQivZYhjaBoEniYvUm99WFRRAX0Q&aep=10&ntc=1&mstk=AUtExfAbIn5S8tOwcHfu8G_xzA3fVKteWE0jAsaKueKyRzM3Yj-0RW_NP5W33AmVs631VmDGtChwYTZ4mPjVjrJEFJmLFWyke9adDZStoSO5K-b9nfe0yfipYsr_A4emI4hqGl34IFKdWXdEL0sxX6qAofTNU5Or2Pz5wxoZ5ts2wA5y8ch0FXQuYaFrLq8iSik99osInOLFHQcFFje6rs-RjRuqx07A98kX3STYoGDDpFbta2V02bUSglWdPw&csuir=1>

Delta table Script

Thursday, July 31, 2025 9:51 PM

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, md5, concat_ws, expr
from delta.tables import DeltaTable
import sys
import logging
# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
def create_spark_session():
    """
    Creates and configures a SparkSession for Delta Lake and S3 access.
    """
    try:
        spark = (SparkSession.builder
                 .appName("DMS_CDC_to_DeltaLake")
                 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
                 .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
                 .getOrCreate())
        logger.info("SparkSession created successfully.")
        return spark
    except Exception as e:
        logger.error(f"Error creating SparkSession: {e}")
        sys.exit(1)
def process_dms_cdc(spark, raw_data_path, curated_delta_path, primary_key_cols=None,
hash_key_cols=None):
    """
    Reads DMS CDC files, performs merge operation on Delta Lake table,
    and writes to the curated S3 location.
Args:
        spark (SparkSession): The active SparkSession.
        raw_data_path (str): S3 path to the DMS raw data (e.g., s3://your-raw-bucket/table_name/).
        curated_delta_path (str): S3 path to the Delta Lake curated table
                                  (e.g., s3://your-processed-bucket/curated_database/curated_table/).
        primary_key_cols (list): List of column names that form the primary key
                                 in the source table (if available).
        hash_key_cols (list): List of column names to use for generating a surrogate key
                              if no primary key is available or desired.
    """
    try:
        # 1. Read the raw DMS CDC data
        logger.info(f"Reading raw DMS CDC data from: {raw_data_path}")
        dms_cdc_df = spark.read.format("parquet").load(raw_data_path)
# 2. Identify the operation indicator column
        # Assuming the first column is the operation indicator, or it's a named column '__op'
        if '__op' not in dms_cdc_df.columns:
            logger.warning("Operation indicator column '__op' not found. Assuming first column is the
indicator.")
            dms_cdc_df = dms_cdc_df.withColumnRenamed(dms_cdc_df.columns[0], "__op")

        dms_cdc_df.printSchema()
        dms_cdc_df.show(5, truncate=False)
# 3. Handle tables without primary keys (if applicable)
        merge_key_cols = primary_key_cols if primary_key_cols else ["surrogate_key"]
        if not primary_key_cols:
            logger.warning("No primary_key_cols specified. Generating surrogate key using
hashing.")
            if not hash_key_cols:
                logger.error("Neither primary_key_cols nor hash_key_cols provided. Cannot uniquely
```

```
identify rows.")
            raise ValueError("Must provide primary_key_cols or hash_key_cols for surrogate key
generation.")
concatenated_cols = concat_ws("||", *[col(c).cast("string") for c in hash_key_cols])
        dms_cdc_df = dms_cdc_df.withColumn("surrogate_key", md5(concatenated_cols))
# 4. Check if the Delta table exists and perform MERGE
        if not DeltaTable.isDeltaTable(spark, curated_delta_path):
            logger.info(f"Delta table not found at {curated_delta_path}. Performing initial write.")
            final_df_initial_load = dms_cdc_df.filter(col("__op") != 'D').drop("__op")
            if not primary_key_cols:
                final_df_initial_load = final_df_initial_load.drop_duplicates(merge_key_cols)

            final_df_initial_load.write.format("delta").mode("overwrite").save(curated_delta_path)
            logger.info(f"Initial Delta table created at {curated_delta_path}.")

        else:
            logger.info(f"Delta table found at {curated_delta_path}. Performing MERGE operation.")
            deltaTable = DeltaTable.forPath(spark, curated_delta_path)
            merge_condition = " AND ".join([f"target.{col} = source.{col}" for col in merge_key_cols])
(deltaTable.alias("target")
             .merge(
                dms_cdc_df.alias("source"),
                merge_condition
             )
             .whenMatchedUpdate(condition=col("source.__op") == 'U', set={col_name:
col(f"source.{col_name}") for col_name in dms_cdc_df.columns if col_name not in ['__op',
'surrogate_key']})
             .whenMatchedDelete(condition=col("source.__op") == 'D')
             .whenNotMatchedInsert(condition=col("source.__op") == 'I', values={col_name:
col(f"source.{col_name}") for col_name in dms_cdc_df.columns if col_name not in ['__op',
'surrogate_key']})
             .execute())
            logger.info(f"Merge operation completed successfully for Delta table at
{curated_delta_path}.")
except Exception as e:
        logger.error(f"Error during DMS CDC processing: {e}")
        raise
if __name__ == "__main__":
    if len(sys.argv) < 3:
        logger.error("Usage: spark-submit --conf <conf> your_script.py <raw_data_path>
<curated_delta_path> [primary_key_cols_comma_separated]
[hash_key_cols_comma_separated]")
        sys.exit(1)

    RAW_DATA_PATH = sys.argv[1]
    CURATED_DELTA_PATH = sys.argv[2]
    PRIMARY_KEY_COLS_STR = sys.argv[3] if len(sys.argv) > 3 else None
    PRIMARY_KEY_COLS = PRIMARY_KEY_COLS_STR.split(',') if
PRIMARY_KEY_COLS_STR else None
    HASH_KEY_COLS_STR = sys.argv[4] if len(sys.argv) > 4 else None
    HASH_KEY_COLS = HASH_KEY_COLS_STR.split(',') if HASH_KEY_COLS_STR else None
if not PRIMARY_KEY_COLS and not HASH_KEY_COLS:
        logger.error("Error: Must provide either primary_key_cols or hash_key_cols.")
        sys.exit(1)
spark = create_spark_session()
    process_dms_cdc(spark, RAW_DATA_PATH, CURATED_DELTA_PATH,
PRIMARY_KEY_COLS, HASH_KEY_COLS)
    spark.stop()
    logger.info("SparkSession stopped.")
```

Use code with caution.

Key features and explanations

1. Spark Session Configuration: Sets up the Spark session and enables Delta Lake extensions and catalog.

2. Reading Raw DMS Data: Reads Parquet files from the specified S3 path. AWS DMS includes an operation indicator (`I`, `U`, or `D`) in the first field of each CDC record by default. The script attempts to identify this column.

3. Surrogate Key Generation: If primary keys are not provided, a `surrogate_key` is generated by hashing the specified `hash_key_cols` using `md5`.

4. Delta Lake `MERGE` Operation: Checks if the Delta Lake table exists. If not, an initial write is performed, excluding delete records. If it exists, a `MERGE` operation is performed based on the primary key or surrogate key, handling inserts, updates, and deletes based on the operation indicator.

5. Error Handling and Logging: Includes `try-except` blocks for error handling and uses Python's `logging` module for informative messages.

6. Running the Script: Designed to be run as a PySpark application on EMR. Use `spark-submit`, providing the raw data path, curated delta path, and optionally primary/hash key columns as arguments. Ensure Delta Lake dependencies are available. For transaction logging on S3 with concurrent writes, you might need to configure `spark.delta.logStore.s3.impl` and potentially use a DynamoDB table . A simple single-driver scenario might use `org.apache.spark.sql.delta.storage.S3SingleDriverLogStore` .

From <https://www.google.com/search?q=aws+dms+cdc+vs+spark+delta+package&sca_esv=fd80f86e9ec25f50&rlz=1C1GCEA_en-GBIN1171IN1171&ei=PICLaP_fM7CX4-EP4L-T2Ak&ved=2ahUKEwjO7uLzqeeOAxXK2TgGHeuuFVAQ0NsOegQINRAA&uact=5&sclient=gws-wiz-serp&udm=50&fbs=AIIjpHxU7SXXniUZfeShr2fp4giZud1z6kQpMfoEdCJxnpm_3YlUqOpj4OTU_HmqxOd8LCYAmZcz3xp4-s3ijYzIP40Lia9Q6yhlX2F8k3-TPnVLuf4vo2DKzxtKYVdS0l2dcByStNDjWIjl6lKjlPBgbneCbCrk0IwpQPyA9hdoscDz59XMWdfVb6wSUXBGtP5Q-cNQivZYhjaBoEniYvUm99WFRRAX0Q&aep=10&ntc=1&mstk=AUtExfBTO7DvJ8dbF4meipq5tz-RvxetrclF3zPt2xcSPgM9pZDRnbFfaZ-1nShwulMPY8N4nWHM_duEWG57A-cuF242IUf3mrpYCZ6V3ZVDCVi4ZMt_RYQHxgsmqf0Ukhg9zlgjjR8Y-8awRF4Pc6ckBKy5H8LEUsFlKX63e7DJAtRJOD1BksHJQw630F4CJi0pzYum_yRp2KdOElOMYhsDRZwcLpDJrp608QyXivjXqaUUXrp-fMK7TmiEfg&csuir=1>

Event Bridge Cron Job Sample

Thursday, July 31, 2025 9:55 PM

cron(Minutes Hours Day-of-month Month Day-of-week Year)

Example cron(0 2 10 * ? *)

0 minutes, 2am, 10th day of the month, all the month,
don't care about the day of the week, all the year

* Indicates all value
? Indicates don't care about the value. If you have you can mention FRI (Friday)

```
// This is a sample configuration for an AWS EventBridge rule that uses a cron schedule.
// You would typically define this using AWS CloudFormation, AWS CDK, or the AWS CLI.
{
  "Rule": {
    "Name": "daily-data-pipeline-trigger",
    "Description": "Triggers the data pipeline every day at 02:00 AM UTC.",
    "ScheduleExpression": "cron(0 2 * * ? *)",
    "State": "ENABLED",
    "EventBusName": "default"
  },
  "Targets": [
    {
      // The target is the resource that the rule will invoke when triggered.
      // In your case, this would likely be a Step Function or a Lambda function.
      "Id": "1",
      "Arn":
"arn:aws:states:REGION:ACCOUNT_ID:stateMachine:YOUR_STEP_FUNCTION_NAME",
      "RoleArn": "arn:aws:iam::ACCOUNT_ID:role/SERVICE_ROLE_FOR_EVENTBRIDGE"
    }
  ]a
}
// Breakdown of the cron expression: cron(Minutes Hours Day-of-month Month Day-of-week
Year)
//
// "cron(0 2 * * ? *)" translates to:
// - 0: At the 0th minute of the hour.
// - 2: At the 2nd hour (2 AM) UTC.
// - *: Every day of the month.
// - *: Every month.
// - ?: Any day of the week (used when Day-of-month is specified).
// - *: Every year.
//
// For example, if you wanted the job to run at 8:00 AM UTC every Monday:
// "cron(0 8 ? * MON *)"
```

Step Function for EMR Job

```
{
  "Comment": "Data pipeline from S3 Raw to S3 Processed (EMR Serverless) to Redshift.",
  "StartAt": "SubmitEMRServerlessJob",
  "States": {
    "SubmitEMRServerlessJob": {
      "Comment": "Submits a PySpark job to an EMR Serverless application to process raw data.",
      "Type": "Task",
      "Resource": "arn:aws:states:::emr-serverless:startJobRun.sync",
      "Parameters": {
        "ApplicationId": "your-emr-serverless-application-id",
        "ExecutionRoleArn": "arn:aws:iam::your-aws-account-
id:role/EMRServerlessJobExecutionRole",
        "JobDriver": {
          "SparkSubmit": {
            "EntryPoint": "s3://your-scripts-bucket/pyspark/delta_merge_job.py",
            "EntryPointArguments": [
              "--input-path",
              "s3://your-raw-bucket/cdc-data/",
              "--output-path",
              "s3://your-processed-bucket/curated-delta-lake/"
            ]
          }
        },
        "ConfigurationOverrides": {
          "MonitoringConfiguration": {
            "S3MonitoringConfiguration": {
              "LogUri": "s3://your-log-bucket/emr-serverless-logs/"
            }
          }
        }
      },
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "SendFailureNotification"
        }
      ],
      "Retry": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "IntervalSeconds": 30,
          "MaxAttempts": 3,
          "BackoffRate": 2
        }
      ],
      "Next": "UpdateRedshiftSchema"
    },
    "UpdateRedshiftSchema": {
      "Comment": "Invokes a Lambda function to dynamically update the Redshift table schema.",
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda::your-aws-account-
```

```
id:function:RedshiftSchemaUpdateLambda",
        "Payload": {
          "TableName.$": "$.ProcessedTableName",
          "SchemaDefinition.$": "$.SchemaDefinition"
        }
      },
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "SendFailureNotification"
        }
      ],
      "Next": "LoadDataToRedshift"
    },
    "LoadDataToRedshift": {
      "Comment": "Invokes a Lambda function to load refined data from S3 to Redshift using the
COPY command.",
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda::your-aws-account-
id:function:RedshiftDataLoaderLambda",
        "Payload": {
          "S3Path.$": "$.S3ProcessedPath",
          "RedshiftTable.$": "$.RedshiftTableName"
        }
      },
      "Catch": [
        {
          "ErrorEquals": [
            "States.TaskFailed"
          ],
          "Next": "SendFailureNotification"
        }
      ],
      "Next": "SendSuccessNotification"
    },
    "SendSuccessNotification": {
      "Comment": "Invokes a Lambda function to send a success email notification via SNS.",
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda::your-aws-account-id:function:NotificationLambda",
        "Payload": {
          "NotificationType": "SUCCESS",
          "Message": "Pipeline execution completed successfully."
        }
      },
      "End": true
    },
    "SendFailureNotification": {
      "Comment": "Invokes a Lambda function to send a failure email notification via SNS with
detailed error logs.",
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "arn:aws:lambda::your-aws-account-id:function:NotificationLambda",
        "Payload.$": "States.Format('Pipeline execution failed at state {}. Error: {}', $.Cause,
$.Error)"
      },
      "End": true
    }
  }
```

```
}
```

HDFS

Friday, August 1, 2025 12:07 AM

https://www.datacamp.com/blog/hadoop-interview-questions

Hadoop is an open-source framework for handling large datasets distributed across multiple computers. It stores data across multiple machines as small blocks using the Hadoop Distributed File System (HDFS). With Hadoop, you can add more nodes to a cluster and handle extensive data without expensive hardware upgrades.

two main components of Hadoop?

The following are the two main components of Hadoop:

- **Hadoop Distributed File System (HDFS):** It manages data storage by breaking large files into block-sized chunks, each 128 MB by default, and distributes them across multiple nodes in a cluster.
- **MapReduce:** It is a programming model that Hadoop relies on for data processing. It's a two-phase process consisting of mapping and reducing jobs. Mapping takes the data and converts it into another set of data. The reducer takes the value pairs output from the map job as input and combines them into smaller pairs. The goal is to process them into a user-defined reduce function.

role of NameNode and DataNode in Hadoop.

HDFS consists of a NameNode and multiple DataNodes for data management, as seen in the image above. Here's how they work:

- As a master server in HDFS, **NameNode** handles operations like opening, closing, and renaming files. It maintains metadata of permissions and the location of the blocks.
- **DataNodes**, on the other hand, act as worker nodes to store the actual data blocks of a file. When a client needs to read or write data, it gets the location of the data blocks from NameNodes. After that, the client communicates directly with the relevant DataNodes to perform the required operations.

Storage Unit In Hadoop (HDFS):

Data Nodes: Block & replication. Default size of block is 128MB

Name Mode - There is another back up for name node is Secondary name node.

YARN components.

Resource Manager: It runs on a master daemon and controls the resource allocation in the cluster.

Node Manager: It runs on the slave daemons and executes a task on each single Data Node.

Application Master: It controls the user job lifecycle and resource demands of single applications. It works with the Node Manager and monitors the execution of tasks.

Container: It is a combination of resources, including RAM, CPU, Network, HDD, etc., on a single node.

- **MapReduce:** A batch processing model that writes intermediate results to disk after each Map or Reduce phase. This disk I/O can lead to higher latency, especially for iterative algorithms.
- **Spark:** Primarily an in-memory processing engine that stores intermediate results in RAM. This significantly reduces disk I/O and enables much faster execution, particularly for iterative computations and interactive queries.

-c --> display file paths & directories
-d --> Directories
-h -->Format the file for human readable format
-q --> print
-r --> Recursively
-t --> Sort files by modification time
-s --> sort files by size
-r --> reverse the order

Git clone <link>
ls -lrt --> list all the files in repo
Cd datafiles
Unzip salesdata.zip
Rm salesdata.zip
Clear
Mkdir -p
Put data files
Ls newfolder
Rmdir
Rm
Mkdir
Hadoop fs -get sourcefilepath destinationfilepath
Hadoop fs -put sourcefilepath destinationfilepath
Head
Tail
Cat
First10
Last 10
Haddop fs -cat filepath\ .last 10
Df --> file size, available, use %
Dy --> Size in gb's and mb's

Python

Friday, August 1, 2025 12:17 AM

Modules
Counter
elements
Data types - Numeric, Text, Sequence, Mapping, Boolean, Binary
Comprehension data types
Date & Time
If Else
For loop
While
UDF
Lambda Function
Map - map(lambda x: x * 2, numbers)
Filter- filter(lambda x: x % 2 == 0, numbers)
Reduce- Reduce(lambda x,y: x+y, numbers)
File handling
Magic Method
Enumerator - list to Index 0: apple
Decorator
Generators
Array
kwargs
Python OOPs
Class
Object
inheritance
Environment variable
Copy vs Deep copy
Dataframe- read, writer, fillna, dropna, create, delete, update records & columns., replacing, joins,renaming, loc vs iloc,
Garbage collect
Multi threading
cache
__init__
__main__
__name__
@getter
@setter
Exception Handling
collections
Traceback Exception
operators
List of list,
Hasp map
Binary search
Sorted list
Breadth-First-Search
Palindrome
Optimise the python code for better performance
Gil Interpreter Lock
Json.loads
Json.dumps
ever debugged a memory issue in a Python job? How did you handle it?

generators and functions difference ?
what are GIN (Python Global Interpreter Lock)
what is deep copy and shallow copy in python ? what are differences ?
Multi-Processing & Multi-Threading
generatorz,decorators

Pyspark

Friday, August 1, 2025 12:17 AM

Architecture - DAG, stage, task
select, withColumn, drop, filter, distinct, orderBy, groupBy
Column expressions (when, col, lit, expr)
Common transformations vs actions
Null handling (fillna, dropna, isNull, isNotNull)
Schema inference vs manual schema
Window functions (row_number, lead, lag, etc.)
Aggregate functions (countDistinct, approx_count_distinct)
Broadcast join
Skew join problem and solutions
bucketing/partitioning/Salting
repartition vs coalesce
Shuffle – what causes it?
Wide vs narrow transformations
map() vs mapPartitions()
Caching vs checkpointing vs persist
CTE
Window Functions
Query Optimization - Explain plan
Functions - Date,string,Regex
Higher Order functions (map and array)
udf

Core Concepts:

- **What is PySpark and how does it relate to Apache Spark?** Understand the role of PySpark as the Python API for Spark.
- **RDDs vs. DataFrames vs. Datasets:** Explain the differences, advantages, and use cases for each data abstraction.
- **SparkSession and SparkContext:** Describe their roles and importance in a PySpark application.
- **Lazy Evaluation:** Explain the concept of lazy evaluation and its impact on performance.
- **Transformations and Actions:** Differentiate between transformations (e.g., map, filter) and actions (e.g., collect, count).

Data Manipulation and Processing:

- **Creating DataFrames:** How to create DataFrames from various sources (e.g., CSV, JSON, Pandas DataFrame).
- **Common DataFrame Operations:** Demonstrate usage of functions like select, filter, groupBy, agg, withColumn, join.
- **User-Defined Functions (UDFs):** Explain UDFs and provide examples of their creation and application.
- **Handling Missing Data:** Strategies for dealing with null values (e.g., fillna, dropna).
- **Window Functions:** Explain and provide examples of window functions for

analytical tasks.

Performance and Optimization:

- **Partitioning:** Explain the concept of partitioning and its impact on performance. Discuss repartition and coalesce.
- **Caching and Persistence:** Understand cache() and persist(), and different storage levels.
- **Data Skew:** Identify and discuss strategies for handling data skew.
- **Catalyst Optimizer:** Explain the role of the Catalyst optimizer in optimizing Spark SQL queries.
- **Broadcast Variables:** Describe broadcast variables and their use cases for optimizing joins.
- **Shuffle Operations:** Explain shuffle operations and how to minimize their impact.

Advanced Topics:

- **Structured Streaming:** Explain how PySpark handles real-time streaming data.
- **Troubleshooting and Debugging:** Techniques for debugging PySpark applications and handling common issues.
- **Machine Learning with MLlib:** If applicable to the role, questions on using PySpark's MLlib for machine learning tasks.

From <https://www.google.com/search?q=pyspark+interview+questions&sca_esv=d81d33885b31fd73&rlz=1C1GCEA_en-GBIN1171IN1171&ei=5sKLaIOlDKKQ4-EPuNfssQY&oq=Pyspark&gs_lp=Egxnd3Mtd2l6LXNlcnAiB1B5c3BhcmsqAggAMg4QABiABBiRAhixAxiKBTIKEAAYgAQYQxiKBTINEAAYgAQYsQMYQxiKBTIOEAAYgAQYkQIYsQMYigUyChAAGIAEGEMYigUyChAAGIAEGEMYigUyChAAGIAEGEMYigUyBRAAGIAEMgoQABiABBhDGIoFSJS8AVD3GliZrgFwBHgBkAEAmAHIAaAB8geqAQUzLjQuMbgBA8gBAPgBAZgCDKACoAioAhTCAgoQABiwAxjWBBhHwgINEAAYgAQYsAMYQxiKBcICBhAAGAcYHsICExAAGIAEGEMYtAIYigUY6gLYAQHCAhAQABgDGLQCGOoCGI8B2AECwgISEAAYAxi0AhjqAhgKGI8B2AECwgIQEAAYgAQYsQMYQxiDARiKBcICCxAAGIAEGLEDGIMBwgIIEAAYgAQYsQPCAgsQABiABBiRAhiKBZgDBPEFTcHeWxAOH8uIBgGQBgq6BgQIARgHugYGCAIQARgKkgcFNy40LjGgB4UusgcFMy40LjG4B4wIwgcFMC43LjXIByc&sclient=gws-wiz-serp>

EMR

Friday, August 1, 2025 12:39 AM

https://interviewprep.org/amazon-elastic-mapreduce-emr-interview-questions/

1. Explain the architecture of Amazon Elastic MapReduce (EMR) and how it enables effective data processing and analysis?
2. How does Amazon EMR differ from traditional Hadoop and Spark clusters? What are the specific advantages and limitations of using Amazon EMR?
3. How do you optimize the performance of an EMR job? What factors should be considered?
4. Describe the process of resizing an Amazon EMR cluster. What are some best practices to maintain high availability and optimal performance while resizing a cluster?
5.Discuss the use of EMR File System (EMRFS) in Amazon EMR. What benefits does it provide when compared to HDFS?
6.Explain the role of spot instances in Amazon EMR, and how it can be used to achieve cost-effective resource allocation.
7.What are the different security configurations available in Amazon EMR, and how can the security of an Amazon EMR cluster be improved?
8.Describe the different types of EMR clusters (transient and long-running) and their appropriate use cases.
9.Can you explain how Amazon EMR supports the use of custom machine learning (ML) algorithms? What is the process for integrating custom ML libraries into an EMR cluster?
10.Discuss how Amazon EMR integrates with AWS Glue, AWS Lake Formation, and Amazon Athena. How can these services complement each other?
11.Explain how to monitor the performance of an Amazon EMR cluster in real-time. What metrics and tools are used for this purpose?
12.How does Amazon EMR handle data durability and data loss prevention? Discuss the various data backup options available.
13.Describe the bootstrap actions in Amazon EMR. What are their use cases, and how can you create custom bootstrap actions?
14.How does Amazon EMR handle instance failures? Explain the process of automatic failover and recovery.
15.Can you discuss the concept of data locality in Amazon EMR and its impact on job performance?
16.Describe how Amazon EMR can be integrated with Amazon S3, and discuss the benefits of storing input and output data in S3.
17.

Sagemaker

Friday, August 1, 2025 12:48 AM

SageMaker Pipelines Fundamentals:
What is AWS SageMaker Pipelines, and what problems does it solve in the MLOps lifecycle?
Explain the key components of a SageMaker Pipeline (e.g., Steps, DAG, Parameters).
How do you define and orchestrate a SageMaker Pipeline? (e.g., using the SageMaker Python SDK).
Describe the benefits of using SageMaker Pipelines over manual execution of ML tasks.
Pipeline Steps and Execution:
Explain different types of steps you can include in a SageMaker Pipeline (e.g., Processing, Training, Model, RegisterModel, CreateModel).
How do you pass data and artifacts between different steps in a pipeline?
How do you handle dependencies between pipeline steps?
Explain how to manage pipeline execution, including starting, stopping, and monitoring runs.
Data and Model Management:
How do you integrate S3 for data input and output within SageMaker Pipelines?
How do you version control your pipeline definitions and models within SageMaker?
Explain how SageMaker Pipelines facilitates model registration and deployment to endpoints.
Advanced Concepts and Best Practices:
How do you implement CI/CD for SageMaker Pipelines?
Discuss how to handle errors and failures within a pipeline.
How would you monitor a SageMaker Pipeline for performance and data drift?
Explain how to parameterize a pipeline for different environments or datasets.
Describe how to implement conditional logic within a SageMaker Pipeline.
How do you ensure reproducibility in your SageMaker Pipelines?
Troubleshooting and Optimization:
What are common challenges faced when building and maintaining SageMaker Pipelines, and how do you address them?
How do you debug a failed pipeline run?
Discuss strategies for optimizing pipeline performance and cost.
Example Scenario-Based Questions:
Design a SageMaker Pipeline for a specific use case (e.g., building a sentiment analysis model, a fraud detection system).
You have a large dataset that needs pre-processing before training. How would you incorporate this into a SageMaker Pipeline?
How would you set up automated retraining of a model in a SageMaker Pipeline based on performance degradation or data drift?

Step Function

Friday, August 1, 2025 12:53 AM

- **What is AWS Step Functions and what problem does it solve?**
  - Explain its role as a serverless workflow orchestration service for coordinating distributed applications and microservices.
- **Describe the key components of a Step Functions state machine.**
  - Discuss states (Task, Choice, Parallel, Map, Pass, Wait, Succeed, Fail) and how they define the workflow logic.
- **What is Amazon States Language (ASL)?**
  - Explain its purpose as a JSON-based, structured language for defining state machines.
- **When would you choose Step Functions over other orchestration tools like AWS SQS or SNS?**
  - Highlight the benefits of visual workflows, state management, error handling, and long-running processes that Step Functions provides.
- **How does Step Functions handle errors and retries?**
  - Discuss Retry and Catch clauses within states for robust error handling.

Practical and Scenario-Based Questions:

- **How do you integrate Step Functions with AWS Lambda functions?**
- Explain how Lambda functions can be invoked as Task states within a workflow.
- Describe a scenario where you would use a Choice state.
- Provide an example of conditional logic based on input data or task results.
- **How would you implement parallel execution of tasks in Step Functions?**
- Explain the Parallel state and its use cases for concurrent processing.
- **How can you pass data between steps in a Step Functions workflow?**
- Discuss InputPath, ResultPath, and OutputPath for managing data flow.
- **How do you monitor and troubleshoot Step Functions executions?**
- Mention integration with CloudWatch and CloudTrail for logging and diagnostics.
- **How can you ensure the security of your Step Functions workflows?**
- Discuss IAM roles and policies for granting necessary permissions to Step Functions to interact with other AWS services.

Advanced Questions:

- **Explain the difference between Standard and Express Workflows.**
- Discuss their respective use cases, billing models, and execution characteristics.
- **How can you implement long-running processes with Step Functions?**
- Discuss the Wait state and its role in pausing workflows.
- **How would you handle external callbacks or human approvals in a Step Functions workflow?**
- Explain the concept of Task Tokens for integrating with external systems.
- **How can you use Step Functions for data processing pipelines or machine learning workflows?**
- Provide examples of orchestrating various AWS services for these purposes.

From <https://www.google.com/search?q=aws+Step+function+interview+questions&sca_esv=d81d33885b31fd73&rlz=1C1GCEA_en-GBIN1171IN1171&ei=HcKLaNyVONyp4-EPmp2AyQg&ved=0ahUKEwicmPTV6OeOAxXc1DgGHZoOIIkQ4dUDCBA&uact=5&oq=aws+Step+function+interview+questions&gs_lp=Egxnd3Mtd2l6LXNlcnAiJWF3cyBTdGVwIGZ1bmN0aW9uIGludGVydmlldyB

xdWVzdGlvbnMyBhAAGAcYHjILEAAYgAQYhgMYigUyCBAAGIAEGKIEMggQABiABBiiBEiiKVC4BFiVJ3ACeAGQAQCYAXygAdQQqgEE
MTQuOLgBA8gBAPgBAZgCFqACxw_CAgoQABiwAxjWBBhHwgIHEAAYgAQYDcICCBAAGAgYDRgewgIIECEYoAEYwwTCAgUQABjvBc
ICBhAAGA0YHsICDhAAGIAEGJECGLEDGIoFwgIFEAAYgATCAggQABgHGAgYHpgDAIgGAZAGCJIHBTEyLjEwoAeBfbIHBTEwLjEwuAe_
D8IHBjAuMTYuNsgHMw&sclient=gws-wiz-serp>

Lambda

Friday, August 1, 2025 12:54 AM

What is AWS Lambda?
Explain its role as a serverless compute service and how it eliminates the need for managing servers.
How does Lambda differ from EC2?
Discuss the key distinctions in terms of server management, scaling, and pricing models.
What are common use cases for Lambda?
Provide examples like real-time data processing, backend for mobile/web apps, and event-driven architectures.
Which programming languages does Lambda support?
List the supported runtimes.
Explain serverless computing.
Define the concept and its benefits (reduced operational overhead, automatic scaling, pay-per-execution).
Event Triggers and Integrations:
What are common services that can trigger a Lambda function? Discuss integrations with S3, DynamoDB, API Gateway, SQS, etc.
How would you trigger a Lambda on an S3 file upload? Describe the setup process.
How can you use API Gateway with Lambda to build a RESTful API? Explain the integration and request/response handling.
Architecture and System Design:
How would you design a serverless application using Lambda? Discuss considerations like data storage, authentication, and inter-service communication.
What are common design patterns in serverless applications? Examples include fan-out, scatter-gather, and event sourcing.
How do you handle stateful applications in a serverless environment? Discuss options like DynamoDB, S3, or external databases.
Performance and Optimization:
Explain the cold start problem in AWS Lambda and how to mitigate it. Discuss techniques like Provisioned Concurrency, optimizing package size, and language choice.
What is Provisioned Concurrency? Explain its purpose and how it helps reduce cold starts.
How can you optimize Lambda function performance? Discuss memory allocation, code optimization, and efficient resource utilization

Friday, August 1, 2025 12:57 AM

Import sagemake
Import sagemaker.amazon.common as smac
from sagemaker import image_uris

Sagemaker.session = sagemaker.session()
Sagemaker.execution_role()

Container=get_image_uri(boto3.session().region_name,'linear_learner')

Converting training data in record I/O format protopuf format - this is the linear learner will understand the training data
Efficiency for Large-Scale Training:

Smac:
Binary Format: Unlike text-based formats like CSV, Protobuf serializes data into a compact binary form. This drastically reduces file size, which means:

Less Storage Cost: Storing data in S3 is cheaper.

Faster Data Transfer: Less data needs to be moved from S3 to the training instances, reducing training start-up times and overall training duration.

Optimized I/O with RecordIO: The RecordIO wrapper allows for high-throughput, sequential reading of records. When SageMaker training instances pull data from S3, they can stream it very efficiently without needing to load entire files into memory at once or perform complex parsing. This is crucial for large datasets that don't fit into memory.

Once algorithm builted, output data stored in s3 path

A Docker container bundles your model's code, all its exact dependencies, configuration files, and even a miniature operating system environment into a single, self-contained, portable unit. This ensures that the model runs consistently regardless of the underlying infrastructure.

When you create a SageMaker Estimator (for training) or Model (for deployment), you need to specify the Docker image to use.

Sagemaker.estimator.estimator(container,
role,train_instance_count,train_instance_type,outputpath,sagemaker_session)

Linear.set_hyperparameters(feature_dim,predictor_type,loss)

Linear.fit()

Linear.deploy(initial_instance_count,instance_type)

From sagemaker.predictor import csv_seriliazer, json_serialiser

Linerar_regressor.predict(x_test)

https://www.datacamp.com/blog/pyspark-interview-questions

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
        .appName("MySparkApp") \
        .master("local[*]") \
        .getOrCreate()

df_from_csv = spark.read.csv("my_file.csv", header=True)
df_from_parquet = spark.read.parquet("my_file.parquet")
df_from_json = spark.read.json("my_file.json")

# How to drop rows
df_from_csv.dropna(how="any")

# How to fill missing values with a constant
df_from_parquet.fillna(value=2)

# How to impute values with median
from pyspark.ml.feature import Imputer
imputer = Imputer(strategy="median", inputCols=["price","rooms"],
outputCols=["price_imputed","rooms_imputed"])
model = imputer.fit(df_from_json)
df_imputed = model.transform(df_from_json)

# How to cache data in memory
df_from_csv.cache()

# How to persist data in local disk
df_from_csv.persist(storageLevel=StorageLevel.DISK_ONLY)

# How to inner join two datasets
df_from_csv.join(df_from_json, on="id", how="inner")

# How to outer datasets
df_from_json.join(df_from_parquet, on="product_id", how="outer")
```

RDDs are low-level APIs that lack a schema and offer control over the data. They are immutable collections of objects
DataFrames are high-level APIs built on top of RDDs optimized for performance but are not safe-type. They organize structured and semi-structured data into named columns.
Datasets combine the benefits of RDDs and DataFrames. They are high-level APIs that provide safe-type abstraction. They support Python and Scala and provide compile-time type checking while being faster than DataFrames.

```
import pandas as pd
from pyspark.sql import SparkSession

# Initialize SparkSession
```

```
spark = SparkSession.builder.appName("Example").getOrCreate()

# Create Pandas DataFrame
pdf = pd.DataFrame({'id': [1, 2, 3], 'value': [10, 20, 30]})

# Convert to PySpark DataFrame
df_spark = spark.createDataFrame(pdf)

# Convert back to Pandas DataFrame
pdf_new = df_spark.toPandas()
```

Speech to Hilton

Tuesday, August 12, 2025 2:55 PM

1. The Orchestration Layer: driver.py and job_framework.py

The system's control flow is managed by driver.py. This script acts as the entry point, executing the following steps:

1. **Configuration and Context Creation**: driver.py collects all necessary inputs. This includes command-line arguments (like operation), environment variables (MODEL_NAME, ENVIRONMENT, etc.), and static configurations from various **YAML files**.
2. **Configuration Loading**: It loads configuration files like path_config.yml (file paths), model_config.yml (model-specific parameters), inf_props.yml (infrastructure details), and sql.yml (SQL query templates).
3. **JobContext Object**: All of this collected information is encapsulated within a single **JobContext object**. This object serves as a central, immutable data store that is passed to every subsequent component, ensuring consistency and a decoupled architecture.
4. **Dynamic Class Loading**: Using the operation and module_class_name from the JobContext, the driver script dynamically loads and instantiates the correct Python class (a subclass of Joblet from job_framework.py) that will perform the specific task.

This design, facilitated by JobContext and the Joblet base class, allows the system to be highly modular and extensible. The driver's role is to set up the environment and hand off the execution to the appropriate specialized class.

2. The Configuration and Data Layers: The YAML Files

The system is configured through several YAML files, which separate code from configuration.

- path_config.yml: Defines S3 paths and local file paths for different operations (e.g., train, serve).
- model-specific-config.yml: A critical file that acts as the manifest for the entire pipeline. It defines:
  - **Phase Mapping**: It maps each pipeline phase (e.g., train, process_predictpostprocess) to a specific Python class (module_class_name).
  - **Hyperparameters**: It stores the model-specific hyperparameters and other configurations.
- sql.yml: Stores a collection of parameterized SQL queries used for database operations, particularly for data enrichment and archiving.
- inf_props.yml: Contains environment-specific connection details, such as Redshift credentials for UAT and PCI environments.

3. The Utility Layer: Utils.py

Utils.py is a collection of static helper methods that provide shared functionality for the entire system. Key functions include:

- **Argument Validation**: mandatory_args_check_helper ensures required environment variables are present.
- **Database Management**: execute_redshift handles database connections and query execution. It uses get_inf_props and pass_decrypt to securely retrieve and decrypt credentials.
- **Dynamic Loading**: get_class is used by the driver.py to dynamically import and return class objects based on a string name.
- **SQL Templating**: replace_sql_params and craft_copy_sql manipulate SQL strings by replacing placeholders with values from the JobContext.

4. The Business Logic Layers: train.py, serve.py, and process.py

These modules contain the core business logic, with each component inheriting from job_framework.Joblet and accessing data via the JobContext.

- **train.py**: This module contains the **BaseTrainer** class and its subclasses. The process() method orchestrates the machine learning training lifecycle:
  1. **Data Acquisition**: Loads data from the S3 path specified in JobContext.
  2. **Model Training**: Initializes and fits a model (e.g., LightGBMClassifier) using hyperparameters from JobContext.
  3. **Evaluation**: Runs the trained model on a test set to calculate performance metrics.
  4. **Artifact Registration**: Persists the trained model (as a pickled file) and performance metrics (as a JSON file) to S3.
- **process.py**: A family of classes that handle data processing and I/O. For example, PublishPreProcess handles the final data preparation before model scores are archived. It retrieves SQL from sql.yml, replaces parameters, executes the query against Redshift via Utils.py, and unloads the final enriched data back to S3.
- **serve.py**: Manages the deployment of a real-time prediction server.

1. It launches two processes: an **Nginx** reverse proxy and a **Gunicorn** application server.
2. It configures Gunicorn to use serve_wsgi.py as its entry point, passing the model path and other configurations as arguments.
3. **serve_wsgi.py** then initializes the **Serve_app.py** Flask application, setting configuration variables that the application uses.

- **serve_app.py**: The Flask application that exposes the API endpoints.
  - **Singleton Model Loading**: It uses a **ScoringService** class with a singleton pattern to load the trained model into memory only once, ensuring efficient inference.
  - **Endpoints**: It provides a health check (/ping) and a prediction endpoint (/invocations) that handles data deserialization (JSON, CSV, Parquet), runs the model's predict or predict_proba method, and serializes the results back to the client.

Terraform Intro

Monday, August 18, 2025 10:14 AM

1) Create, Change & Manage your infrastructure with configuration files