Contents

1.ETL:

ETL stands for Extract, Transform, Load. It's a process used in data engineering and data warehousing to move and manage data from various sources into a centralized system like a data warehouse or data lake.

Here’s a breakdown of each step:

1. **Extract**

- What it does: Pulls data from different source systems.
- Sources: Databases (SQL, NoSQL), APIs, flat files (CSV, Excel), cloud services, etc.
- Goal: Gather raw data without altering it.

2. Transform

- What it does: Cleans, formats, and restructures the data.
- Tasks involved:
- Data cleaning (removing duplicates, handling missing values)
- Data mapping (converting formats, units)
- Aggregation (summing, averaging)
- Applying business rules
- Goal: Make the data usable and consistent for analysis.

3. Load

- What it does: Moves the transformed data into a target system.
- Targets: Data warehouses (like Snowflake, Redshift), data lakes, or analytics platforms.
- Goal: Store data in a way that supports reporting, analytics, and decision-making.

On-Premises ETL Vs ServerlessETL:

Aspect	On-Premises ETL	Serverless ETL
Infrastructure	Hosted on physical or virtual servers managed in-house	Fully managed by cloud provider; no server management
Scalability	Limited by hardware capacity; manual scaling	Automatically scales based on workload
Cost Model	High upfront cost (hardware, licenses); ongoing maintenance	Pay-as-you-go; cost based on usage
Maintenance	Requires manual updates, monitoring, and patching	Cloud provider handles maintenance and updates
Deployment Speed	Slower; requires setup and configuration	Faster; minimal setup required
Flexibility	Highly customizable; suitable for complex legacy systems	Easier integration with cloud services; limited customization
Security & Compliance	Full control over data and security policies	Depends on cloud provider's compliance and security features
Examples	Informatica PowerCenter, IBM DataStage, Talend (on-prem)	AWS Glue, Azure Data Factory, Google Cloud Dataflow

---

2.AWS-Glue:

AWS Glue is a serverless data integration service that simplifies the process of discovering, preparing, moving, and integrating data from multiple sources. It is designed for analytics, machine learning, and application development, providing a centralized data catalog and a variety of tools for authoring, running, and monitoring ETL (Extract, Transform, Load) jobs.

2.1 Components of Glue:

1.Glue Data Catalog

This is a central metadata repository where AWS Glue stores information about your data sources, such as table definitions, schemas, and job metadata. It helps other AWS services like Athena and Redshift understand the structure of your data.

2.Glue Crawlers

Crawlers automatically scan your data sources (like S3 buckets or databases), infer the schema, and populate the Data Catalog. They save you from manually defining metadata.

3.Glue Jobs

These are the actual ETL scripts that perform data extraction, transformation, and loading. You can write them in Python or Scala, and they run on a managed Apache Spark environment.

4.Glue Studio

A visual interface that lets you build, run, and monitor ETL jobs using a drag-and-drop editor. It's great for users who prefer low-code or no-code development.

5.Glue Triggers

Triggers allow you to schedule Glue jobs or start them based on specific events. You can set them to run at fixed intervals or in response to other job completions.

6.Glue Workflows

Workflows help you orchestrate multiple jobs and triggers into a single pipeline. This is useful for managing complex data processing tasks with dependencies.

7.Glue Notebooks

These are Jupyter-style notebooks integrated into Glue for interactive development and data exploration. You can write and test your ETL code here before deploying it.

8.Glue Dynamic Frames

A special data structure used in Glue that's similar to Spark DataFrames but with added flexibility for handling semi-structured data and schema changes.

9.Glue Marketplace

AWS Glue integrates with the AWS Marketplace, where you can find pre-built connectors and transformations from AWS and third-party vendors to extend Glue's capabilities.

2.2 Pros of AWS Glue:

1.Serverless Architecture

No need to manage infrastructure—AWS handles provisioning, scaling, and maintenance.

2.Integrated Data Catalog

Automatically catalogs metadata, making it easy to discover and query data using services like Athena and Redshift.

3.Scalable and Distributed

Built on Apache Spark, Glue can handle large-scale data processing efficiently.

4.Supports Multiple Languages

You can write ETL scripts in Python or Scala, giving flexibility to developers.

5.Visual ETL Development

Glue Studio provides a drag-and-drop interface for building ETL workflows without writing code.

6.Event-Driven and Scheduled Jobs

Supports triggers and workflows for automating data pipelines.

7.Cost-Effective for Variable Workloads

Pay only for the resources you use, which is ideal for sporadic or unpredictable workloads.

8.Good Integration with AWS Ecosystem

Seamlessly connects with S3, Redshift, RDS, DynamoDB, and other AWS services.

2.3 Cons of AWS Glue:

1.Cold Start Latency

Serverless jobs may take time to start, which can affect performance for time-sensitive tasks.

2.Limited Customization

Compared to self-managed Spark clusters, Glue offers less control over environment configuration.

3.Learning Curve

Concepts like Dynamic Frames and Glue Workflows may be unfamiliar to new users.

4.Debugging Can Be Tricky

Debugging distributed jobs in a serverless environment can be more complex than in local setups.

5.Cost for Continuous Use

For always-on or high-frequency jobs, Glue may become more expensive than other solutions.

6.Limited Third-Party Integration

While improving, Glue's support for external systems and connectors is not as extensive as some competitors.

2.4 Pricing Components

1. ETL Jobs

Spark Jobs (Glue 2.0+): $0.44 per DPU-hour, 1-minute minimum, default 10 DPUs.

Spark Streaming: Same rate, default 2 DPUs.

Flexible Execution (Glue 3.0+): $0.29 per DPU-hour.

Python Shell Jobs: $0.44 per DPU-hour, default 0.0625 DPU.

Ray Jobs (Preview): $0.44 per DPU-hour.

2. Interactive Development

Interactive Sessions: $0.44 per DPU-hour, 1-minute minimum, default 5 DPUs.

Development Endpoints: $0.44 per DPU-hour, 10-minute minimum.

3. Data Catalog

Storage: First 1 million metadata objects are free. Beyond that, $1.00 per 100,000 objects/month.

Access Requests: First 1 million requests/month are free. Beyond that, $1.00 per 1 million requests.

Table Statistics & Iceberg Compaction: $0.44 per DPU-hour, 1-minute minimum.

4. Crawlers

Automatically discover and catalog data.

Rate: $0.44 per DPU-hour.

Minimum Duration: 10 minutes per run.

5. DataBrew

Interactive Sessions: $1.00 per 30-minute session.

Jobs: $0.48 per DataBrew node-hour, 1-minute minimum.

6. Data Quality

Rate: $0.44 per DPU-hour.

Minimum: 2 DPUs per run.

Statistics Storage: Free up to 100,000 statistics.

7. Zero-ETL Integrations

Source Ingestion: $1.50 per GB.

Target Processing: $0.44 per DPU-hour.

2.5 WorkFlow :

The typical workflow involves:

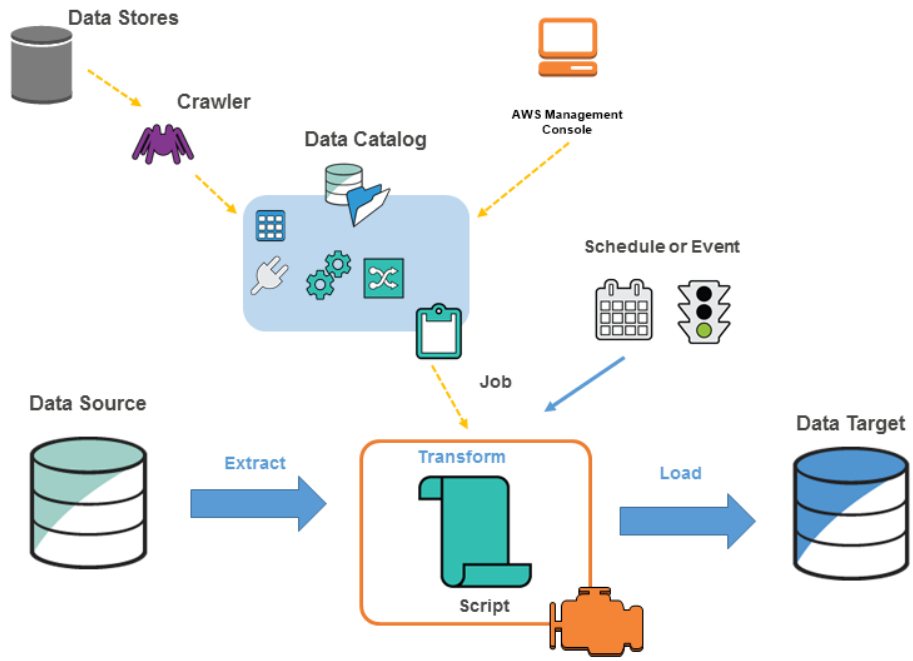
Define data sources and targets in the Data Catalog.

Use Crawlers to populate the Data Catalog with table metadata from data sources.

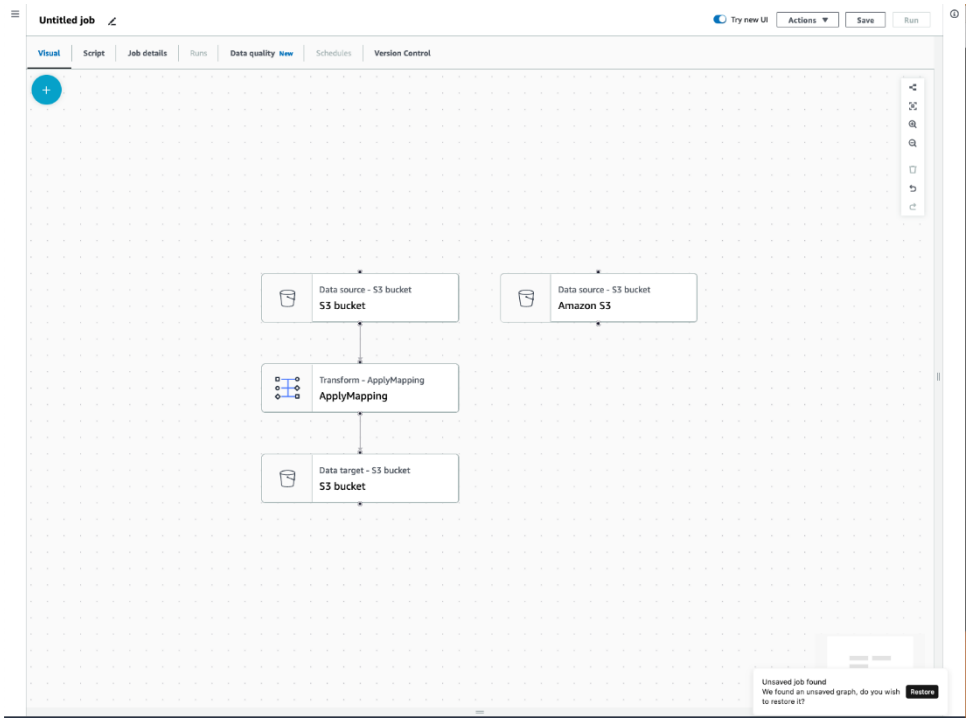
Define ETL jobs with transformation scripts to move and process data.

Run jobs on-demand or based on triggers.

Monitor job performance using dashboards.



3. Visual ETL components



- **Visual** – The Visual job editor canvas. This is where you can add nodes to create a job.
- **Script** – The script representation of your ETL job. AWS Glue generates the script based on the visual representation of your job. You can also edit your script or download it.
- **Job details** – The Job details tab allows you to configure your job by setting job properties. There are basic properties, such as name and description of your job, IAM role, job type, AWS Glue version, language, worker type, number of workers, job bookmark, flex execution, number of retires, and job timeout, and there are advanced properties, such as connections, libraries, job parameters, and tags.
- **Runs** – After your job runs, this tab can be accessed to view your past job runs.
- **Data quality** – Data quality evaluates and monitors the quality of your data assets. You can learn more about how to use data quality on this tab and add a data quality transform to your job.
- **Schedules** – Jobs that you've scheduled appear in this tab. If there are no schedules attached to this job, then this tab is not accessible.
- **Version control** – You can use Git with your job by configuring your job to a Git repository.

IAM Policy:

1.AwsGlueServiceRole

2.AwsGlueConsoleFullAccess

3.CloudWatchFullAccess

3.1 Creating an AWS Glue usage profile

1.In the left navigation menu, choose Cost management.

2.Choose Create usage profile.

3.Enter the Usage profile name for the usage profile.

4.Enter an optional description that will help others recognize the purpose of the usage profile.

5.Define at least one parameter in the profile. Any field in the form is a parameter. For example, the session idle t6.imeout minimum.

7.Define any optional tags that apply to the usage profile.

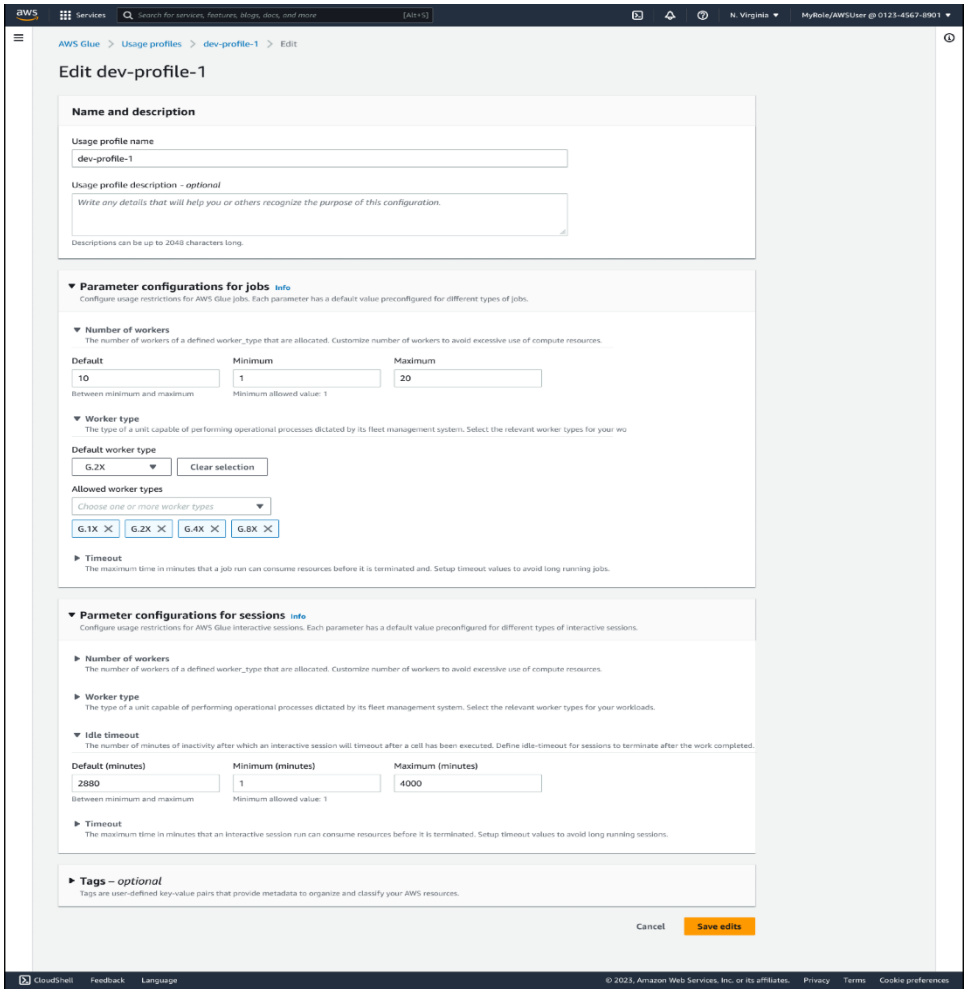
8.Choose Save.

Permissions Required

To use encryption features in AWS Glue, ensure the following permissions are granted:

For AWS Glue Console Users

- glue:GetDataCatalogEncryptionSettings
- glue:PutDataCatalogEncryptionSettings
- glue:CreateSecurityConfiguration
- glue:GetSecurityConfiguration
- glue:GetSecurityConfigurations
- glue:DeleteSecurityConfiguration



For Clients Accessing Encrypted Catalogs

Permissions to use AWS KMS keys:

```
{
 "Effect": "Allow",
 "Action": [
  "kms:Decrypt",
  "kms:Encrypt",
  "kms:GenerateDataKey"
 ],
 "Resource": "arn:aws:kms:us-east-1:111122223333:key/key-id"
}
```

For CloudWatch Logs Encryption

Key Policy must allow:

```
{
 "Effect": "Allow",
 "Principal": { "Service": "logs.region.amazonaws.com" },
 "Action": [
  "kms:Encrypt*", "kms:Decrypt*", "kms:ReEncrypt*",
  "kms:GenerateDataKey*", "kms:Describe*"
 ],
 "Resource": "<kms key arn>"</kms key arn>
}
```

IAM Policy must allow:

```
{
 "Effect": "Allow",
 "Principal": { "Service": "logs.region.amazonaws.com" },
 "Action": [ "logs:AssociateKmsKey" ],
 "Resource": "<kms key arn>"</kms key arn>
}
```

For Job Bookmark Encryption

Permissions

```
{
 "Effect": "Allow",
 "Action": [ "kms:Decrypt", "kms:Encrypt" ],
 "Resource": "arn:aws:kms:us-east-1:111122223333:key/*"
}
```

⚙️ **AWS Glue Console Configuration Steps**

- Data Catalog Encryption
- Go to Settings → Metadata encryption
- Choose AWS KMS key (e.g., aws/glue or a custom symmetric key)

- Security Configuration
- Go to Security configurations → Add security configuration
- Configure:
- S3 encryption: Use SSE-KMS with aws/s3 or custom key
- CloudWatch Logs encryption: Choose a CMK
- Job bookmark encryption: Use aws/glue or custom key
- Connection Setup
- Go to Connections → Add connection
- Enable Require SSL connection for JDBC targets
- Job Setup
- Go to Jobs → Add job
- Assign the created security configuration
- Run the job and verify encryption for:
- Amazon S3 output
- CloudWatch Logs
- Job bookmarks

---

4. AWS Glue Data Catalog:

A centralized metadata repository that stores information about your data sources—location, schema, and runtime metrics. It helps manage, discover, and analyze data across AWS services.

🔍 How to Populate the Data Catalog

Automatically using AWS Glue Crawlers:

* Scans internal (AWS) and external data sources.

* Extracts and updates metadata.

Manually by defining:

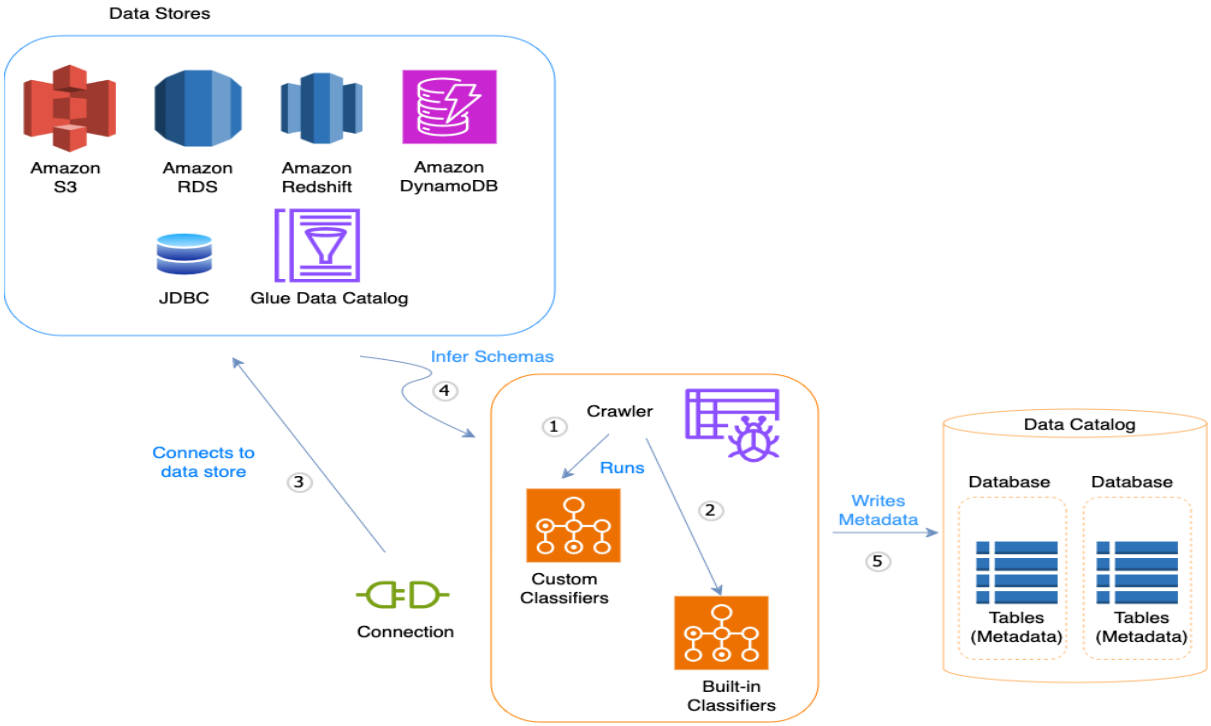
- Table structure
- Schema
- Partitioning

4.1 Key Features

- **Metadata Repository**
- Organized into databases and tables like a relational DB catalog.
- **Automatic Data Discoverability**
- Crawlers keep metadata up-to-date.
- Supports Amazon S3, RDS, Redshift, Hive, and more.
- **Schema Management**
- Handles schema inference, evolution, and versioning.
- **Table Optimization**
- Supports managed compaction for Iceberg tables to improve read performance.
- **Column Statistics**
- Computes stats like min/max, nulls, distinct values, etc., for formats like Parquet, ORC, JSON, CSV.
- **Data Lineage**
- Tracks transformations and operations for auditing and compliance.
- **Integration with AWS Services**
- Works with Athena, Lake Formation, EMR, SageMaker, Redshift Spectrum.
- **Security & Access Control**
- Integrates with Lake Formation for fine-grained access.
- Uses AWS KMS for metadata encryption.

4.2 populate the Data Catalog :

4.2.1Using Crawlers:



🛠️ **Crawler Workflow**

1. **Run Custom Classifiers (Optional)**
    - You can define and order custom classifiers.
    - The first one that matches the data structure is used to infer the schema.
    - Remaining classifiers are skipped.
2. **Fallback to Built-in Classifiers**
    - If no custom classifier matches, AWS Glue uses built-in classifiers (e.g., for JSON, CSV, etc.).
3. **Connect to Data Store**
    - The crawler connects to the source (e.g., S3, RDS).
    - Some sources require connection properties.
4. **Infer Schema**
    - Based on the data format and structure, the crawler infers the schema.
5. **Write Metadata to Data Catalog**
    - A **table definition** is created and stored in a **database** within the Data Catalog.
    - Includes attributes like **classification** (label from the classifier).

4.2.1.1 Creating a Crawler:

- **Step 1: Set crawler properties**
- **Step 2: Choose data sources and classifiers**
- **Step 3: Configure security settings**
- **Step 4: Set output and scheduling**
- **Step 5: Review and create**

Create a crawler schedule:

1. Sign in to the AWS Management Console, and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose Crawlers in the navigation pane.
3. Follow steps 1-3 in the Configuring a crawler section.
4. In Step 4: Set output and scheduling, choose a Crawler schedule to set the frequency of the run. You can choose the crawler to run hourly, daily, weekly, monthly or define custom schedule using cron expressions.

   A cron expression is a string representing a schedule pattern, consisting of 6 fields separated by spaces: * * * * * <minute> <hour> <day of month> <month> <day of week> <year>For example, to run a task every day at midnight, the cron expression is: 0 0 * * ? *.For more information, see Cron expressions.
5. Review the crawler settings you configured, and create the crawler to run on a schedule.

Create a schedule for an existing crawler:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose Crawlers in the navigation pane.
3. Choose a crawler that you want to schedule from the available list.
4. Choose Edit from the Actions menu.
5. Scroll down to Step 4: Set output and scheduling, and choose Edit.
6. Update your crawler schedule under Crawler schedule.
7. Choose Update.

Customizing crawler behavior:

- **Incremental crawls** – You can configure a crawler to run incremental crawls to add only new partitions to the table schema.
- **Partition indexes** – A crawler creates partition indexes for Amazon S3 and Delta Lake targets by default to provide efficient lookup for specific partitions.

- **Accelerate crawl time by using Amazon S3 events** – You can configure a crawler to use Amazon S3 events to identify the changes between two crawls by listing all the files from the subfolder which triggered the event instead of listing the full Amazon S3 or Data Catalog target.
- **Handling schema changes** – You can prevent a crawlers from making any schema changes to the existing schema. You can use the AWS Management Console or the AWS Glue API to configure how your crawler processes certain types of changes.
- **A single schema for multiple Amazon S3 paths** – You can configure a crawler to create a single schema for each S3 path if the data is compatible.
- **Table location and partitioning levels** – The table level crawler option provides you the flexibility to tell the crawler where the tables are located, and how you want partitions created.
- **Table threshold** – You can specify the maximum number of tables the crawler is allowed to create by specifying a table threshold.
- **AWS Lake Formation credentials** – You can configure a crawler to use Lake Formation credentials to access an Amazon S3 data store or a Data Catalog table with an underlying Amazon S3 location within the same AWS account or another AWS account.

Scheduling incremental crawls for adding new partitions:

AWS Glue crawlers can be configured to run **incremental crawls**. The first crawl performs a **full scan** of the data source to capture the complete schema and all existing partitions. After that, **subsequent crawls** only detect and add **new partitions**, making the process faster and more efficient by avoiding reprocessing of previously discovered data.

Steps:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose Crawlers under the Data Catalog.
3. Choose a crawler that you want to set up to crawl incrementally.
4. Choose Edit.
5. Choose Step 2. Choose data sources and classifiers.
6. Choose the data source that you want to incrementally crawl.
7. Choose Edit.
8. Choose Crawl new sub-folders only under Subsequent crawler runs.

9. Choose Update.

(or)

1. Go to AWS Glue Console → Crawlers.
2. Create or edit a crawler.
3. Set the data store (e.g., S3 path).
4. Choose “Add new partitions only” in the crawler configuration.
5. Set a schedule (e.g., cron expression or periodic).
6. Run the crawler or wait for the schedule.

Preventing a crawler from changing an existing schema:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose **Crawlers** under the **Data Catalog**.
3. Choose a crawler from the list, and choose **Edit**.
4. Choose **step 4, Set output and scheduling**.
5. Under **Advance options**, choose **Add new columns only** or **Ignore the change and don't update the table in the Data Catalog**.
6. You can also set a configuration option to **Update all new and existing partitions with metadata from the table**. This sets partition schemas to inherit from the table.
7. Choose **Update**.

Creating a single schema for each Amazon S3 include path:

To help illustrate this option, suppose that you define a crawler with an include path s3://amzn-s3-demo-bucket/table1/. When the crawler runs, it finds two JSON files with the following characteristics:

- **File 1** – S3://amzn-s3-demo-bucket/table1/year=2017/data1.json
- *File content* – {“A”: 1, “B”: 2}
- *Schema* – A:int, B:int
- **File 2** – S3://amzn-s3-demo-bucket/table1/year=2018/data2.json
- *File content* – {“C”: 3, “D”: 4}
- *Schema* – C: int, D: int

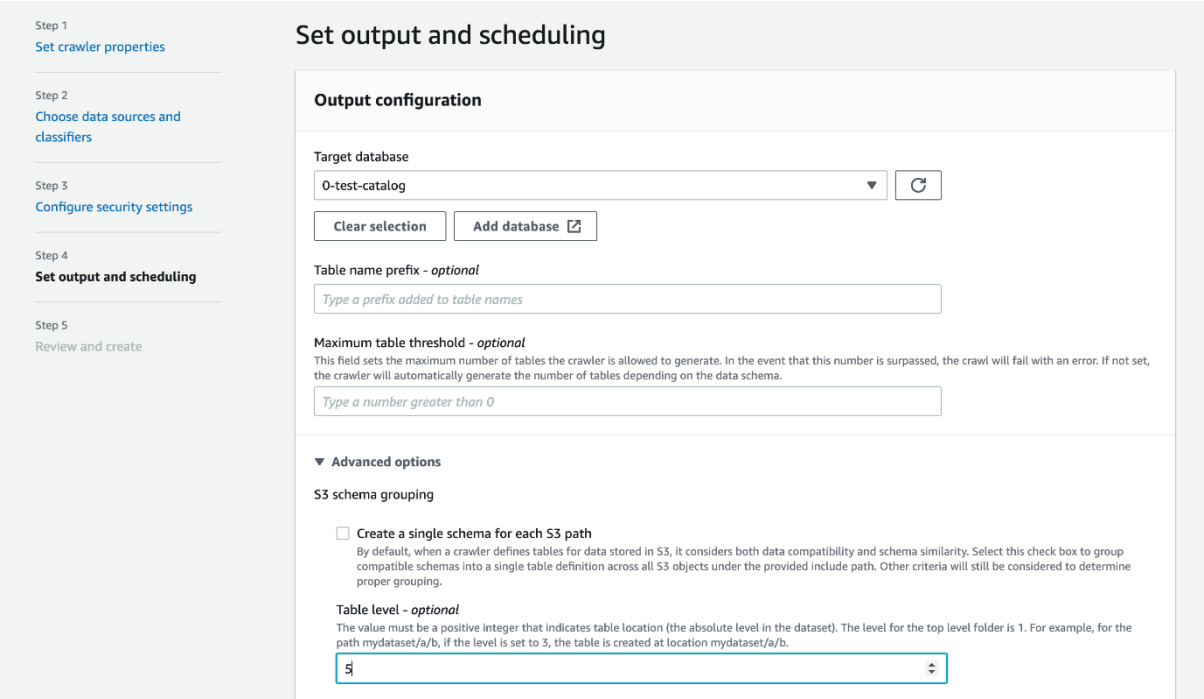
By default, the crawler creates two tables, named year_2017 and year_2018 because the schemas are not sufficiently similar. However, if the option **Create a single schema for each S3 path** is selected, and if the data is compatible, the crawler creates one table. The table has the schema A:int,B:int,C:int,D:int and partitionKey year:string.

Steps:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose **Crawlers** under the **Data Catalog**.
3. When you configure a new crawler, under **Output and scheduling** , select the option **Create a single schema for each S3 path** under Advance options.

Specifying the table location and partitioning level:

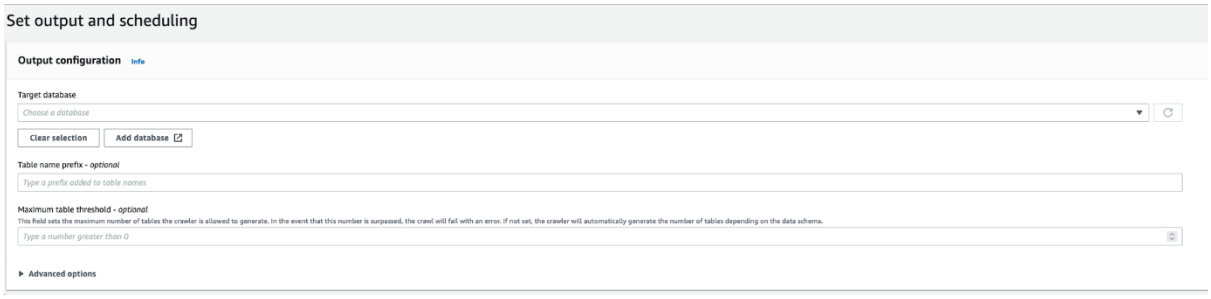
1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Choose **Crawlers** under the **Data Catalog**.
3. When you configure a crawler, under **Output and scheduling**, choose **Table level** under **Advance options**.



Specifying the maximum number of tables the crawler is allowed to create:

To set TableThreshold using the AWS Management Console:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. When configuring a crawler, in **Output and scheduling**, set the **Maximum table threshold** to the number of tables the crawler is allowed generate.



4.2.2 Accelerating crawls using Amazon S3 event notifications:

Amazon S3 event-based crawlers improve efficiency by using S3 event notifications to detect changes, allowing faster and more cost-effective recrawls. After an initial full crawl, subsequent crawls only scan updated subfolders triggered by events, rather than the entire S3 or Data Catalog target. This reduces crawl time and cost, and the crawler operates based on events from an SQS queue.

Step 1: Create an Amazon SQS queue

Follow the steps to create and subscribe to an Amazon Simple Queue Service (Amazon SQS) queue.

1. Using the Amazon SQS console, create a queue. For instructions, see Getting Started with Amazon SQS in the *Amazon Simple Queue Service Developer Guide*.
2. Replace the access policy that's attached to the queue with the following policy.
    a. In the Amazon SQS console, in the Queues list, choose the queue name.
    b. On the Access policy tab, choose Edit.
    c. Replace the access policy that's attached to the queue. In it, provide your Amazon SQS ARN, source bucket name, and bucket owner account ID.

```
{
  "Version": "2012-10-17",
  "Id": "example-ID",
  "Statement": [
    {
      "Sid": "example-statement-ID",
      "Effect": "Allow",
      "Principal": {
        "Service": "s3.amazonaws.com"
      },
```

```
        "Action": [
          "SQS:SendMessage"
        ],
        "Resource": "arn:aws:sqs:us-west-2:111122223333:s3-notification-queue",
        "Condition": {
          "ArnLike": {
            "aws:SourceArn": "arn:aws:s3:*:*:awsexamplebucket1"
          },
          "StringEquals": {
            "aws:SourceAccount": "bucket-owner-account-id"
          }
        }
      }
    ]
}
```

d. Choose Save.

3. Note the queue ARN.

   The SQS queue that you created is another resource in your AWS account. It has a unique Amazon Resource Name (ARN). You need this ARN in the next step. The ARN is of the following format:

   ```
   arn:aws:sqs:aws-region:account-id:queue-name
   ```

Step 2: Create an Amazon SNS topic

Follow the steps to create and subscribe to an Amazon SNS topic.

1. Using Amazon SNS console, create a topic. For instructions, see Creating an Amazon SNS topic in the *Amazon Simple Notification Service Developer Guide*.
2. Subscribe to the topic. For this exercise, use email as the communications protocol. For instructions, see Subscribing to an Amazon SNS topic in the *Amazon Simple Notification Service Developer Guide*.

You get an email requesting you to confirm your subscription to the topic. Confirm the subscription.

3. Replace the access policy attached to the topic with the following policy. In it, provide your SNS topic ARN, bucket name, and bucket owner's account ID.
4. Note the topic ARN.

The SNS topic you created is another resource in your AWS account, and it has a unique ARN. You will need this ARN in the next step. The ARN will be of the following format:

arn:aws:sns:*aws-region*:*account-id*:*topic-name*

Step 3: Add a notification configuration to your bucket:

1. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. In the left navigation pane, choose **General purpose buckets**.
3. In the buckets list, choose the name of the bucket that you want to enable events for.
4. Choose **Properties**.
5. Navigate to the **Event Notifications** section and choose **Create event notification**.
6. In the **General configuration** section, specify descriptive event name for your event notification. Optionally, you can also specify a prefix and a suffix to limit the notifications to objects with keys ending in the specified characters.
    a. Enter a description for the **Event name**.

       If you don't enter a name, a globally unique identifier (GUID) is generated and used for the name.
    b. (Optional) To filter event notifications by prefix, enter a **Prefix**.

       For example, you can set up a prefix filter so that you receive notifications only when files are added to a specific folder (for example, `images/`).
    c. (Optional) To filter event notifications by suffix, enter a **Suffix**.

       For more information, see Configuring event notifications using object key name filtering.
7. In the **Event types** section, select one or more event types that you want to receive notifications for.

   For a list of the different event types, see Supported event types for SQS, SNS, and Lambda.
8. In the **Destination** section, choose the event notification destination.
    a. Select the destination type: **Lambda Function**, **SNS Topic**, or **SQS Queue**.

b. After you choose your destination type, choose a function, topic, or queue from the list.
c. Or, if you prefer to specify an Amazon Resource Name (ARN), select **Enter ARN** and enter the ARN.

For more information, see Supported event destinations.

9. Choose **Save changes**, and Amazon S3 sends a test message to the event notification destination.

Step 4: Test the setup

Now, you can test the setup by uploading an object to your bucket and verifying the event notification in the Amazon SQS console. For instructions, see Receiving a Message in the *Amazon Simple Queue Service Developer Guide "Getting Started" section*.

After Creating Event in s3 Add the following SQS policy to the role used by the crawler.

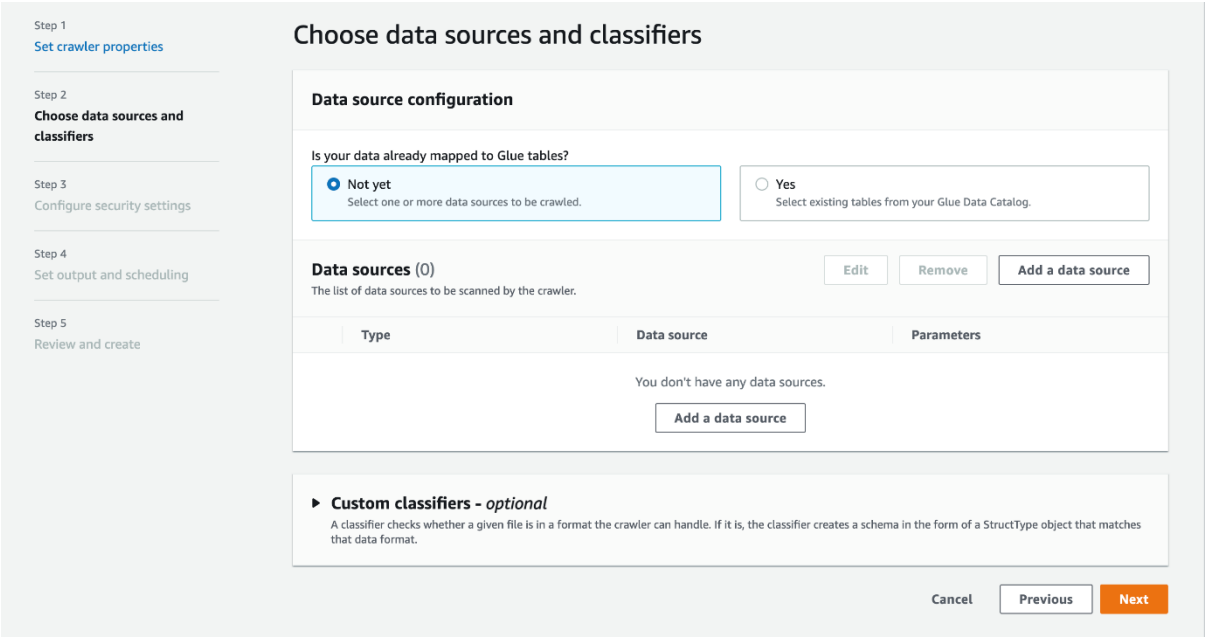
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "sqs:DeleteMessage",
        "sqs:GetQueueUrl",
        "sqs:ListDeadLetterSourceQueues",
        "sqs:ReceiveMessage",
        "sqs:GetQueueAttributes",
        "sqs:ListQueueTags",
```

```
        "sqs:SetQueueAttributes",
        "sqs:PurgeQueue"
      ],
      "Resource": "arn:aws:sqs:us-east-1:111122223333:cfn-sqs-queue"
    }
  ]
}
```

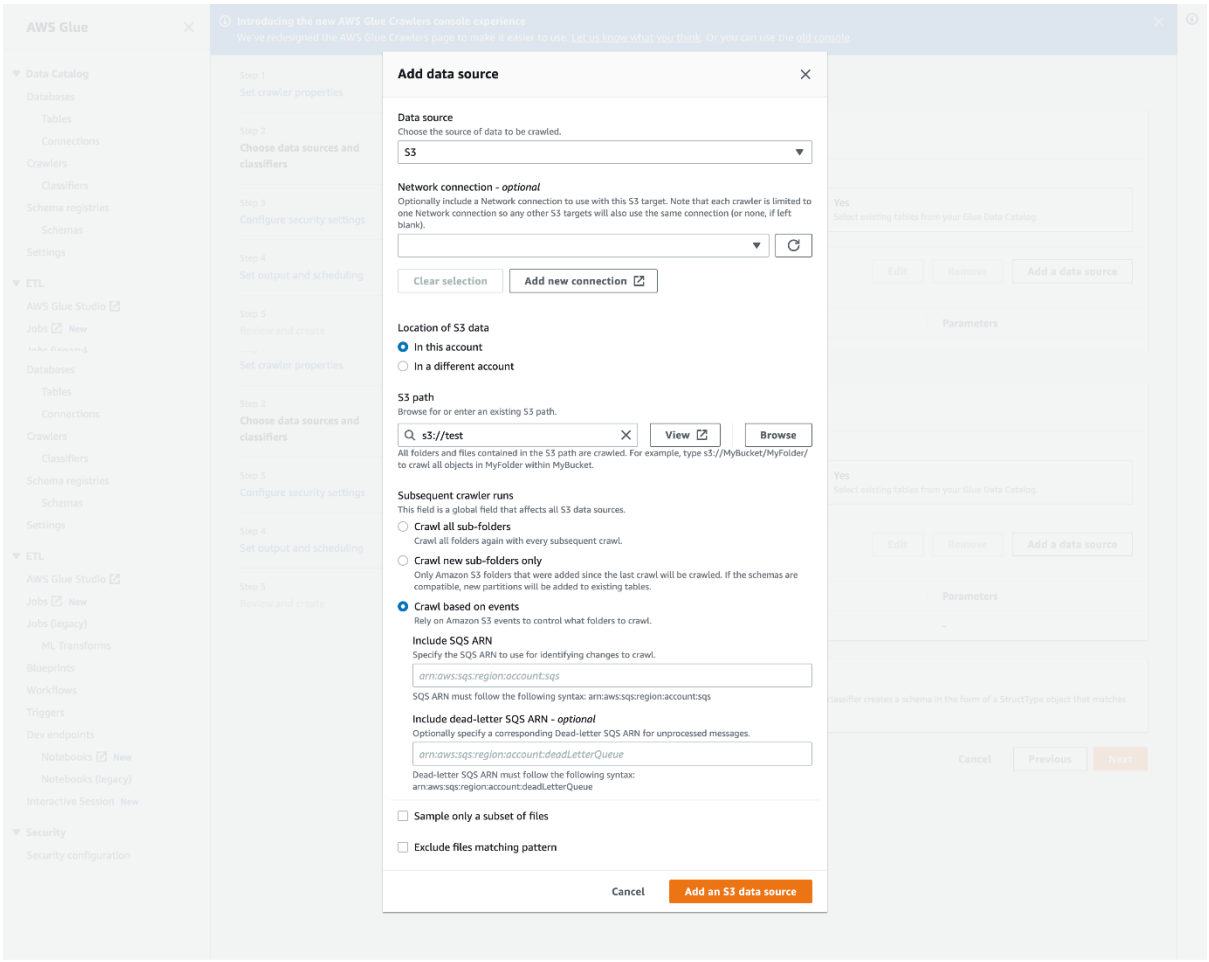
4.2.3 Setting up a crawler for Amazon S3 event notifications for an Amazon S3 target:

1. Sign in to the AWS Management Console and open the GuardDuty console at https://console.aws.amazon.com/guardduty/.
2. Set your crawler properties. For more information, see Setting Crawler Configuration Options on the AWS Glue console .
3. In the section **Data source configuration**, you are asked *Is your data already mapped to AWS Glue tables?*

   By default **Not yet** is already selected. Leave this as the default as you are using an Amazon S3 data source and the data is not already mapped to AWS Glue tables.
4. In the section **Data sources**, choose **Add a data source**.



5. In the **Add data source** modal, configure the Amazon S3 data source:
    - **Data source**: By default, Amazon S3 is selected.
    - **Network connection** (Optional): Choose **Add new connection**.
    - **Location of Amazon S3 data**: By default, **In this account** is selected.
    - **Amazon S3 path**: Specify the Amazon S3 path where folders and files are crawled.
    - **Subsequent crawler runs**: Choose **Crawl based on events** to use Amazon S3 event notifications for your crawler.
    - **Include SQS ARN**: Specify the data store parameters including the a valid SQS ARN. (For example, `arn:aws:sqs:region:account:sqs`).
    - **Include dead-letter SQS ARN** (Optional): Specify a valid Amazon dead-letter SQS ARN. (For example, `arn:aws:sqs:region:account:deadLetterQueue`).
    - Choose **Add an Amazon S3 data source**.



---

5. Integrating with other AWS services:

While you can use AWS Glue crawlers to populate the AWS Glue Data Catalog, there are several AWS services that can automatically integrate with and populate the catalog for you. The following sections provide more information about the specific use cases supported by AWS services that can populate the Data Catalog.

- AWS Lake Formation
- Amazon Athena

5.1 Amazon Athena:

1. In the Athena console, create a database that will store the table metadata in the Data Catalog.
2. Use the `CREATE EXTERNAL TABLE` statement to define the schema of your data source.

3. Use the `PARTITIONED BY` clause to define any partition keys if your data is partitioned.
4. Use the `LOCATION` clause to specify the Amazon S3 path where your actual data files are stored.
5. Run the `CREATE TABLE` statement.

   This query creates the table metadata in the Data Catalog based on your defined schema and partitions, without actually crawling the data.

You can query the table in Athena, and it will use the metadata from the Data Catalog to access and query your data files in Amazon S3.

6.Managing transactional tables:

6.1Creating Apache Iceberg tables:

It is an open table format for very large analytic datasets. Iceberg allows for easy changes to your schema, also known as schema evolution, meaning that users can add, rename, or remove columns from a data table without disrupting the underlying data. Iceberg also provides support for data versioning, which allows users to track changes to data overtime. This enables the time travel feature, which allows users to access and query historical versions of data and analyze changes to the data between updates and deletes.

Steps:

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. Under Data Catalog, choose **Tables**, and use the **Create table** button to specify the following attributes:
   - **Table name** – Enter a name for the table. If you're using Athena to access tables, use these naming tips in the Amazon Athena User Guide.
   - **Database** – Choose an existing database or create a new one.
   - **Description** – The description of the table. You can write a description to help you understand the contents of the table.
   - **Table format** – For **Table format**, choose Apache Iceberg.
   - **Enable compaction** – Choose **Enable compaction** to compact small Amazon S3 objects in the table into larger objects.

- **IAM role** – To run compaction, the service assumes an IAM role on your behalf. You can choose an IAM role using the drop-down. Ensure that the role has the permissions required to enable compaction.

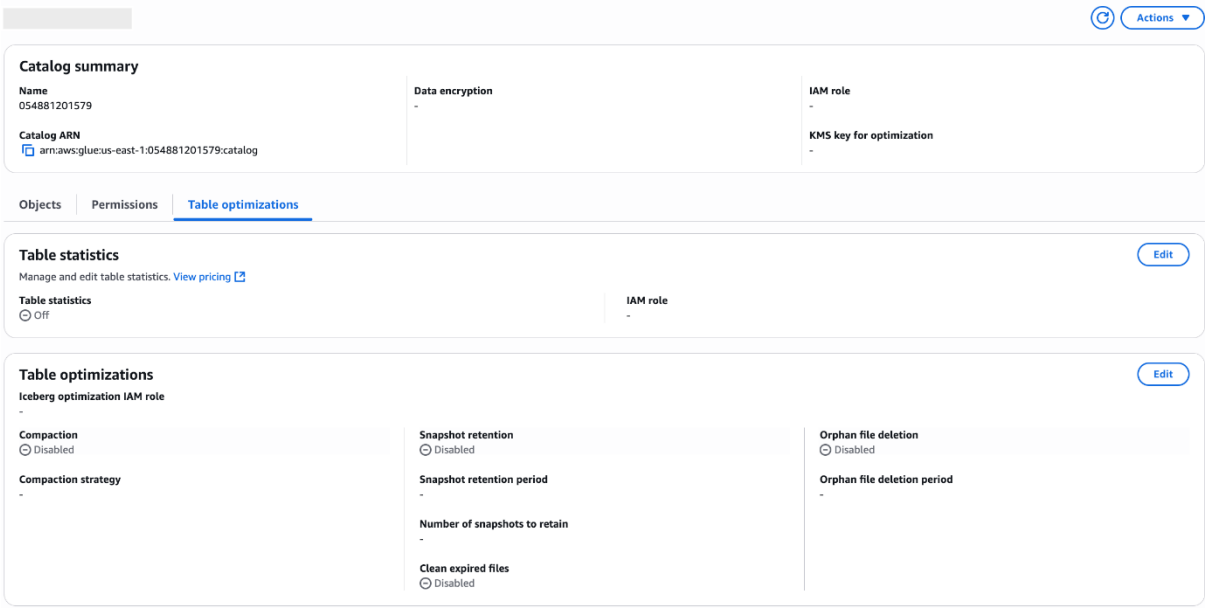
  To learn more about the required permissions, see Table optimization prerequisites .
- **Location** – Specify the path to the folder in Amazon S3 that stores the metadata table. Iceberg needs a metadata file and location in the Data Catalog to be able to perform reads and writes.
- **Schema** – Choose **Add columns** to add columns and data types of the columns. You have the option to create an empty table and update the schema later. Data Catalog supports Hive data types. For more information, see Hive data types.

  Iceberg allows you to evolve schema and partition after you create the table. You can use Athena queries to update the table schema and Spark queries for updating partitions.

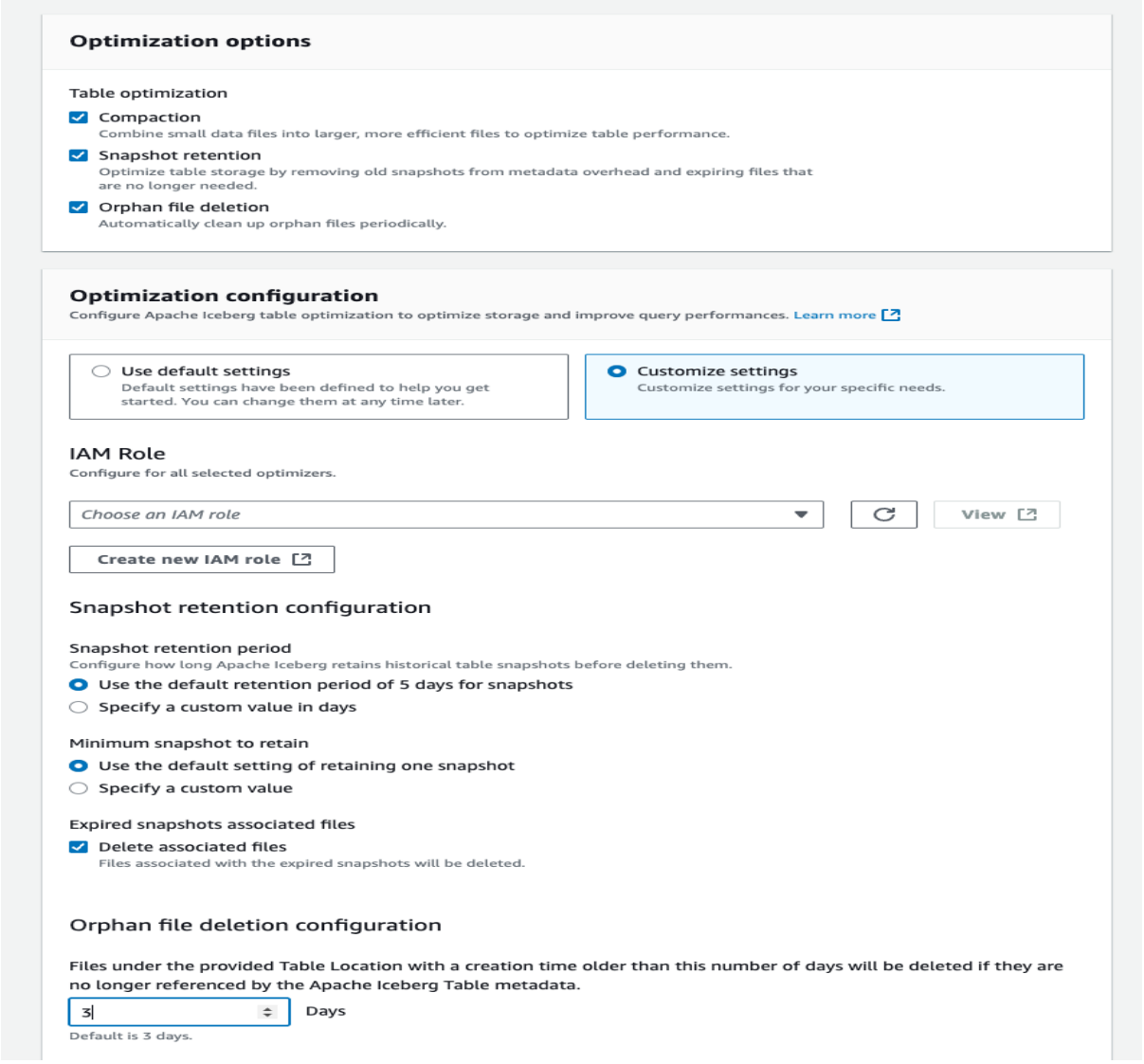
---

7. Enabling catalog-level automatic table optimization:

1. Open the Lake Formation console at https://console.aws.amazon.com/lakeformation/.
2. In the navigation pane, choose **Data Catalog**.
3. Select the **Catalogs** tab.
4. Choose the account-level catalog.
5. Choose **Table optimizations**, **Edit** under **Table optimizations** tab. You can also choose **Edit optimizations** from **Actions**.



6. On the **Table optimization** page, configure the following options:



a. Configure **Compaction** settings:
   - Enable/disable compaction.
   - Choose the IAM role that has the necessary permissions to run the optimizers.

     For more information on the permission requirements for the IAM role, see Table optimization prerequisites .

b. Configure **Snapshot retention** settings:
   - Enable/disable retention.
   - Set snapshot retention period in days - default is 5 days.
   - Set number of snapshots to retain - default is 1 snapshot.
   - Enable/disable cleaning of expired files.

c. Configure **Orphan file deletion** settings:
   - Enable/disable orphan file deletion.
   - Set orphan file retention period in days - default is 3 days.

7. Choose **Save**.

---

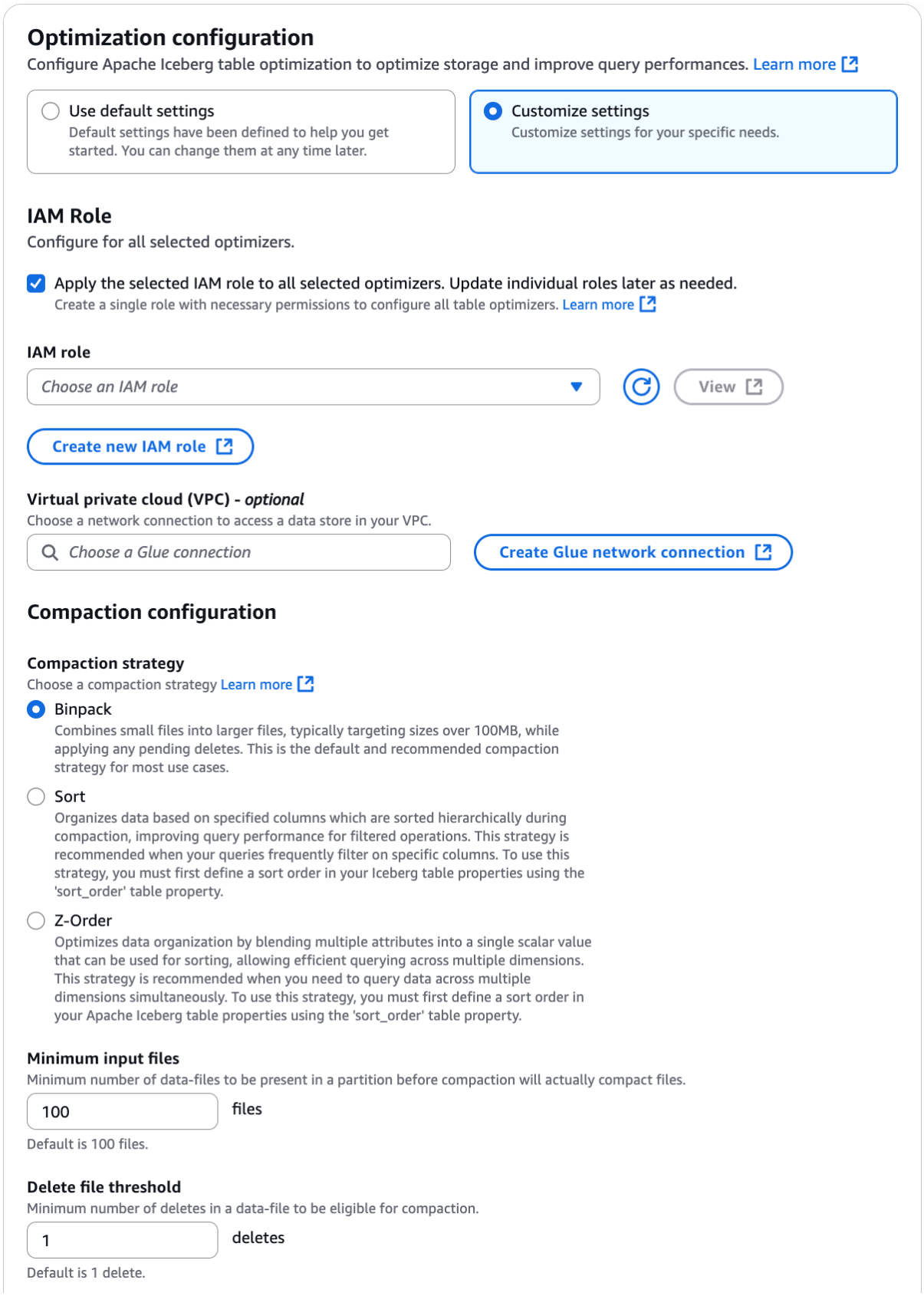
8. Compaction optimization:

TO ENABLE COMPACTION

1. Open the AWS Glue console at https://console.aws.amazon.com/glue/ and sign in as a data lake administrator, the table creator, or a user who has been granted the `glue:UpdateTable` and `lakeformation:GetDataAccess` permissions on the table.
2. In the navigation pane, under **Data Catalog**, choose **Tables**.
3. On the **Tables** page, choose a table in open table format that you want to enable compaction for, then under **Actions** menu, choose **Optimization**, and then choose **Enable**.

   You can also enable compaction by selecting the **Table optimization** tab on the **Table details** page. Choose the **Table optimization** tab on the lower section of the page, and choose **Enable compaction**.

   The **Enable optimization** option is also available when you create a new Iceberg table in the Data Catalog.
4. On the **Enable optimization** page, choose **Compaction** under **Optimization options**.



5. Next, select an IAM role from the drop down with the permissions shown in the Table optimization prerequisites section.

You can also choose **Create a new IAM role** option to create a custom role with the required permissions to run compaction.

Follow the steps below to update an existing IAM role:

a. To update the permissions policy for the IAM role, in the IAM console, go to the IAM role that is being used for running compaction.
b. In the **Add permissions** section, choose Create policy. In the newly opened browser window, create a new policy to use with your role.
c. On the Create policy page, choose the `JSON` tab. Copy the JSON code shown in the Prerequisites into the policy editor field.

6. If you have security policy configurations where the Iceberg table optimizer needs to access Amazon S3 buckets from a specific Virtual Private Cloud (VPC), create an AWS Glue network connection or use an existing one.

   If you don't have an AWS Glue VPC connection set up already, create a new one by following the steps in the Creating connections for connectors section using the AWS Glue console or the AWS CLI/SDK.
7. Choose a compaction strategy. The available options are:
   - **Binpack** – Binpack is the default compaction strategy in Apache Iceberg. It combines smaller data files into larger ones for optimal performance.
   - **Sort** – Sorting in Apache Iceberg is a data organization technique that clusters information within files based on specified columns, significantly improving query performance by reducing the number of files that need to be processed. You define the sort order in Iceberg's metadata using the sort-order field, and when multiple columns are specified, data is sorted in the sequence the columns appear in the sort order, ensuring records with similar values are stored together within files. The sorting compaction strategy takes the optimization further by sorting data across all files within a partition.
   - **Z-order** – Z-ordering is a way to organize data when you need to sort by multiple columns with equal importance. Unlike traditional sorting that prioritizes one column over others, Z-ordering gives balanced weight to each column, helping your query engine read fewer files when searching for data.

     The technique works by weaving together the binary digits of values from different columns. For example, if you have the numbers 3 and 4 from two columns, Z-ordering first converts them to binary (3 becomes 011 and 4 becomes 100), then interleaves these digits to create a new value: 011010. This interleaving creates a pattern that keeps related data physically close together.

Z-ordering is particularly effective for multi-dimensional queries. For example, a customer table Z-ordered by income, state, and zip code can deliver superior performance compared to hierarchical sorting when querying across multiple dimensions. This organization allows queries targeting specific combinations of income and geographic location to quickly locate relevant data while minimizing unnecessary file scans.

8. **Minimum input files** – The number of data files required in a partition before compaction is triggered.
9. **Delete files threshold** – Minimum delete operations required in a data file before it becomes eligible for compaction.
10. Choose **Enable optimization**.

9. Snapshot retention optimization:

- **Retention Configuration**:
  You can set both a retention period (in days) and a maximum number of snapshots to keep for a table.
- **Automatic Cleanup**:
  AWS Glue automatically removes snapshots that are older than the retention period, but always keeps the most recent snapshots up to the configured limit.
- **Data Deletion**:
  After removing old snapshots from metadata, AWS Glue deletes the actual data and metadata files that are no longer referenced by any retained snapshot.
- **Time Travel Limitation**:
  Time travel queries are only possible up to the range of the retained snapshots.
- **Storage Optimization**:
  This process helps reclaim storage space by deleting data associated with expired snapshots.

Steps:

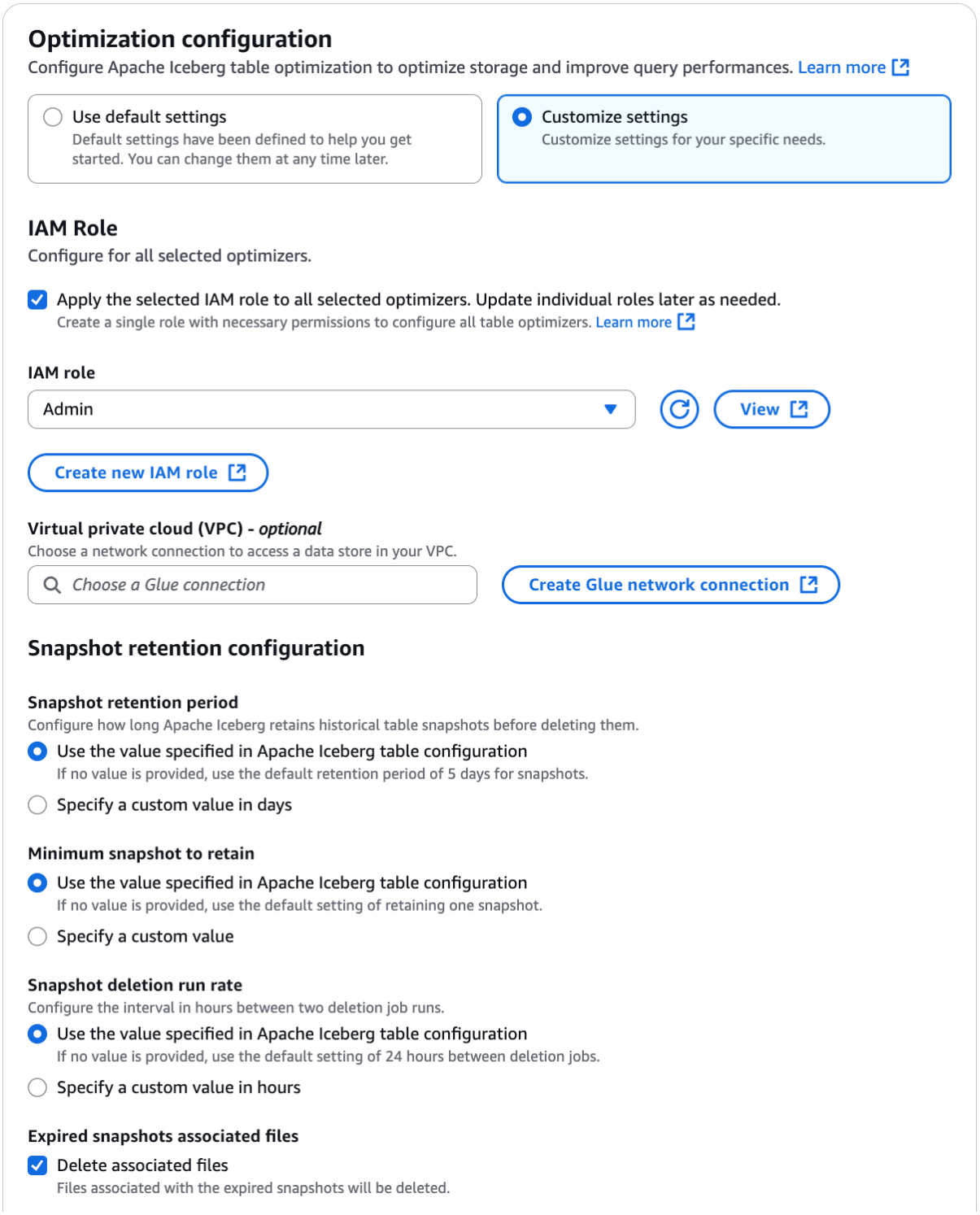
TO ENABLE SNAPSHOT RETENTION OPTIMIZER

1. Open the AWS Glue console at https://console.aws.amazon.com/glue/ and sign in as a data lake administrator, the table creator, or a user who has been granted the `glue:UpdateTable` and `lakeformation:GetDataAccess` permissions on the table.
2. In the navigation pane, under **Data Catalog**, choose **Tables**.

3. On the **Tables** page, choose an Iceberg table that you want to enable snapshot retention optimizer for, then under **Actions** menu, choose **Enable** under **Optimization**.

   You can also enable optimization by selecting the table and opening the **Table details** page. Choose the **Table optimization** tab on the lower section of the page, and choose **Enable snapshot retention**.
4. On the **Enable optimization** page, under **Optimization configuration**, you have two options: **Use default setting** or **Customize settings**. If you choose to use the default settings, AWS Glue utilizes the properties defined in the Iceberg table configuration to determine the snapshot retention period and the number of snapshots to be retained. In the absence of this configuration, AWS Glue retains one snapshot for five days, and deletes files associated with the expired snapshots.
5. Next, choose an IAM role that AWS Glue can assume on your behalf to run the optimizer. For details about the permissions required for the IAM role, see the Table optimization prerequisites section.

   Follow the steps below to update an existing IAM role:
   a. To update the permissions policy for the IAM role, in the IAM console, go to the IAM role that is being used for running compaction.
   b. In the Add permissions section, choose Create policy. In the newly opened browser window, create a new policy to use with your role.
   c. On the Create policy page, choose the JSON tab. Copy the JSON code shown in the Prerequisites into the policy editor field.
6. If you prefer to set the values for the **Snapshot retention configuration** manually, choose **Customize settings**.



10. Deleting orphan files:

- **Orphan Files Definition**:
  Files in Amazon S3 under the table location that are **not referenced by Iceberg metadata** and are **older than a configured age limit**.
- **Causes of Orphan Files**:
  - Failed operations like compaction

    - Dropped partitions
    - Table rewrites
- **Optimizer Functionality**:
    - Scans both **table metadata** and **actual data files**
    - Identifies and deletes orphan files to **reclaim storage space**
- **Deletion Criteria**:
    1. **Date Check**:
        - If file creation date is **older than or equal to optimizer creation date**, it is **skipped**.
    2. **Configuration Check**:
        - If file is **newer than optimizer creation date**, it is evaluated against the **configured age limit**.
        - If it meets the criteria, it is **deleted**; otherwise, it is **skipped**.

The steps for set the Orphan file option is same as snapshot retention but choose Orphan file and mention 3 days.

11. Managing the data catalog

11.1Updating the schema, and adding new partitions:

11.1.1New Partitons:

- When the job finishes, view the new partitions on the console right away, without having to rerun the crawler. You can enable this feature by adding a few lines of code to your ETL script, as shown in the following examples. The code uses the `enableUpdateCatalog` argument to indicate that the Data Catalog is to be updated during the job run as the new partitions are created.

Method 1

Pass `enableUpdateCatalog` and `partitionKeys` in an options argument.

```
additionalOptions = {"enableUpdateCatalog": True}

additionalOptions["partitionKeys"] = ["region", "year",
"month", "day"]




sink =
glueContext.write_dynamic_frame_from_catalog(frame=last_trans
form, database=<target_db_name>,
```

```
table_name=<target_table_name>,
transformation_ctx="write_sink",


additional_options=additionalOptions
```

11.1.2 Updating table schema:

- When the job finishes, view the modified schema on the console right away, without having to rerun the crawler. You can enable this feature by adding a few lines of code to your ETL script, as shown in the following examples. The code uses `enableUpdateCatalog` set to true, and also `updateBehavior` set to `UPDATE_IN_DATABASE`, which indicates to overwrite the schema and add new partitions in the Data Catalog during the job run.

```
additionalOptions = {

    "enableUpdateCatalog": True,

    "updateBehavior": "UPDATE_IN_DATABASE"}

additionalOptions["partitionKeys"] = ["partition_key0",
"partition_key1"]



sink =
glueContext.write_dynamic_frame_from_catalog(frame=last_transform,
database=<dst_db_name>,

    table_name=<dst_tbl_name>, transformation_ctx="write_sink",

    additional_options=additionalOptions)

job.commit()
```

12. Steps to Create a Connection in AWS Glue

1. Open AWS Glue Console

- Go to the AWS Glue Console.
- Make sure you're in the correct region.

2. Navigate to Connections

- In the left-hand menu, click on **"Connections"** under the **Data Catalog** section.

3. Create a New Connection

- Click the **"Add connection"** button.

4. Choose Connection Type

- Select the type of connection:
  - **JDBC** (for databases like MySQL, PostgreSQL, Oracle, etc.)
  - **Amazon Redshift**
  - **MongoDB**
  - **Network (VPC)**
  - **Custom connectors** (if using Glue Studio with custom sources)

5. Configure Connection Details

- Provide the following:
  - **Name**: A unique name for the connection.
  - **Description** (optional).
  - **Connection type**: Choose based on your data source.
  - **JDBC URL**: For example, `jdbc:mysql://your-db-endpoint:3306/dbname`
  - **Username and Password**: Credentials for the database.
  - **VPC, Subnet, and Security Group**: Required if the data source is inside a VPC.

6. Test the Connection (Optional but Recommended)

- After entering the details, you can test the connection to ensure it works.

7. Save the Connection

- Click **"Create connection"** to save it.

13.Build visual ETL jobs with AWS Glue Studio:

- AWS Glue Studio offers a **visual interface** to create, run, and monitor ETL (Extract, Transform, Load) jobs.
- It simplifies ETL development using a **drag-and-drop interface**, eliminating the need to learn Apache Spark or write code manually.

ETL Job Capabilities

- Jobs in AWS Glue:
  - Connect to **source data**
  - Perform **transformations**
  - Write to **target data stores**
- Supports **Apache Spark**, **Ray**, and **Python shell** runtimes.

Job Triggers and Monitoring

- Jobs can be triggered:
  - **On demand**
  - **On a schedule**
  - **Based on events**
- You can monitor jobs for:
  - **Completion status**
  - **Duration**
  - **Start time**

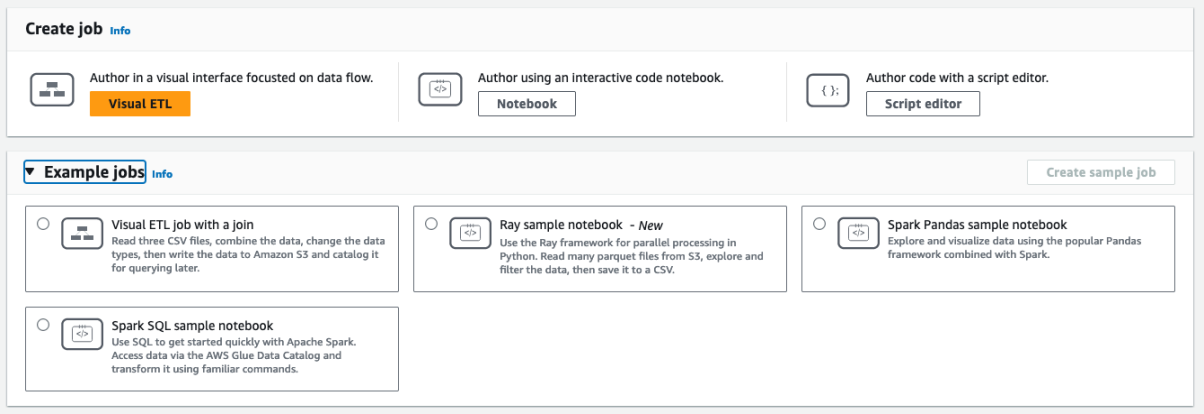
Script Generation and Customization

- AWS Glue Studio can **auto-generate PySpark scripts** based on source and target schemas.
- You can **edit the generated scripts** or provide your own custom scripts.

13.1 Creating a job in AWS Glue Studio from an example job

You can choose to create a job from an example job. In the **Example jobs** section, choose a sample job, then choose **Create sample job**. Creating a sample job from one of the options provides a quick template you can work from.

1. Sign in to the AWS Management Console and open the AWS Glue Studio console at https://console.aws.amazon.com/gluestudio/.
2. Choose **ETL jobs** from the navigation pane.
3. Select an option create a job from a sample job:
    - **Visual ETL job to join multiple sources** – Read three CSV files, combine the data, change the data types, then write the data to Amazon S3 and catalog it for querying later.
    - **Spark notebook using Pandas** – Explore and visualize data using the popular Pandas framework combined with Spark.
    - **Spark notebook using SQL** – Use SQL to get started quickly with Apache Spark. Access data through the AWS Glue Data Catalog and transform it using familiar commands.
4. Choose **Create sample job**.

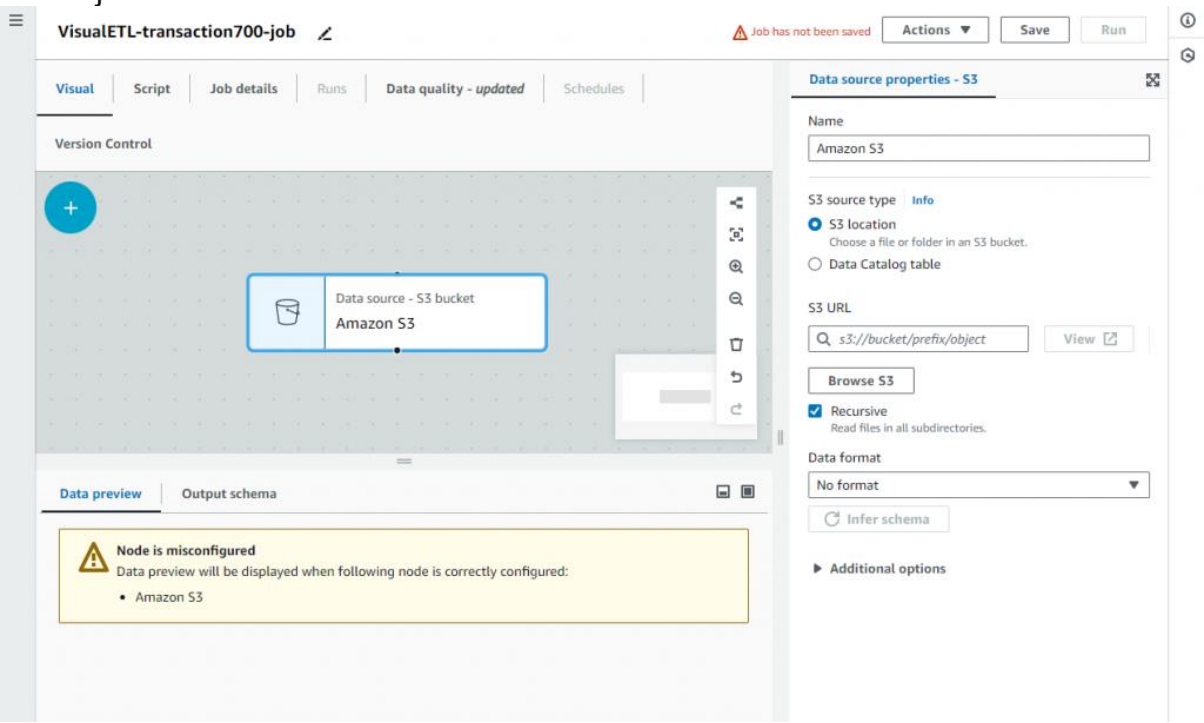


The job editor provides the following features for creating and editing jobs.

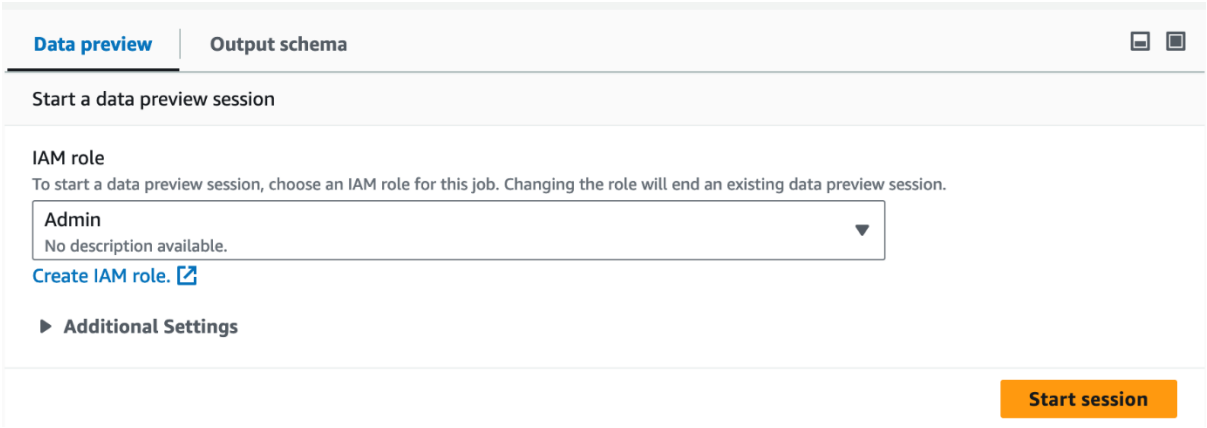
- A visual diagram of your job, with a node for each job task: Data source nodes for reading the data; transform nodes for modifying the data; data target nodes for writing the data.

  You can view and configure the properties of each node in the job diagram. You can also view the schema and sample data for each node in the job diagram. These features help you to verify that your job is modifying and transforming the data in the right way, without having to run the job.

- A Script viewing and editing tab, where you can modify the code generated for your job.
- A Job details tab, where you can configure a variety of settings to customize the environment in which your AWS Glue ETL job runs.
- A Runs tab, where you can view the current and previous runs of the job, view the status of the job run, and access the logs for the job run.
- A Data quality tab, where you can apply data quality rules to your job.
- A Schedules tab, where you can configure the start time for you job, or set up a recurring job runs.
- A Version Control tab, where you can configure a Git service to use with your job.



While creating or editing your job, you can use the **Data preview** tab beneath the job canvas to view a sample of your data. A new data preview session will start automatically when the role is already configured on the job or a default IAM role has been set up in the account. If a role has not been previously configured, you can start a session by selecting the role.



Key Restrictions in Data Previews

1. **IAM Role Required**
    - You must select an IAM role with proper permissions before using data preview.
2. **Loading Time**
    - Small datasets (<1 GB): ~1 minute to load.
    - Large datasets: Use **partitions** to improve performance.
    - **S3** offers best performance.
3. **Timeouts**
    - Preview request times out if it takes **>15 minutes**.
    - **30-minute idle timeout** applies.
4. **Column Display Limits**
    - Only **first 50 columns** shown by default.
    - If columns have no data, preview shows "no data to display."
5. **Unsupported Sources**
    - **Streaming data** and **custom connectors** are not supported.
6. **Error Propagation**
    - Errors in one node affect all connected nodes in the preview.
7. **Schema Mismatch**
    - Changing a data source may require updating child nodes (e.g., ApplyMapping).
8. **SQL Query Errors**
    - Incorrect field names in SQL transform nodes cause preview errors.

13.2 AWS Glue-native transforms:

The following built-in transforms are available with AWS Glue Studio:

- **ChangeSchema**: Map data property keys in the data source to data property keys in the data target. You can rename keys, modify the data types for keys, and choose which keys to drop from the dataset.
- **SelectFields**: Choose the data property keys that you want to keep.
- **DropFields**: Choose the data property keys that you want to drop.
- **RenameField**: Rename a single data property key.
- **Spigot**: Write samples of the data to an Amazon S3 bucket.
- **Join**: Join two datasets into one dataset using a comparison phrase on the specified data property keys. You can use inner, outer, left, right, left semi, and left anti joins.
- **Union**: Combine rows from more than one data source that have the same schema.
- **SplitFields**: Split data property keys into two `DynamicFrames`. Output is a collection of `DynamicFrames`: one with selected data property keys, and one with the remaining data property keys.
- **SelectFromCollection**: Choose one `DynamicFrame` from a collection of `DynamicFrames`. The output is the selected `DynamicFrame`.
- **FillMissingValues**: Locate records in the dataset that have missing values and add a new field with a suggested value that is determined by imputation
- **Filter**: Split a dataset into two, based on a filter condition.
- **Drop Null Fields**: Removes columns from the dataset if all values in the column are 'null'.
- **Drop Duplicates**: Removes rows from your data source by choosing to match entire rows or specify keys.
- **SQL**: Enter SparkSQL code into a text entry field to use a SQL query to transform the data. The output is a single `DynamicFrame`.
- **Aggregate**: Performs a calculation (such as average, sum, min, max) on selected fields and rows, and creates a new field with the newly calculated value(s).
- **Flatten**: Extract fields inside structs into top level fields.
- **UUID**: Add a column with a Universally Unique Identifier for each row.
- **Identifier**: Add a column with a numeric identifier for each row.
- **To timestamp**: Convert a column to timestamp type.
- **Format timestamp**: Convert a timestamp column to a formatted string.

- **Conditional Router transform**: Apply multiple conditions to incoming data. Each row of the incoming data is evaluated by a group filter condition and processed into its corresponding group.
- **Concatenate Columns transform**: Build a new string column using the values of other columns with an optional spacer.
- **Split String transform**: Break up a string into an array of tokens using a regular expression to define how the split is done.
- **Array To Columns transform**: Extract some or all the elements of a column of type array into new columns.
- **Add Current Timestamp transform**: Mark the rows with the time on which the data was processed. This is useful for auditing purposes or to track latency in the data pipeline.
- **Pivot Rows to Columns transform**: Aggregate a numeric column by rotating unique values on selected columns which become new columns. If multiple columns are selected, the values are concatenated to name the new columns.
- **Unpivot Columns To Rows transform**: Convert columns into values of new columns generating a row for each unique value.
- **Autobalance Processing transform**: Redistribute the data better among the workers. This is useful where the data is unbalanced or as it comes from the source doesn't allow enough parallel processing on it.
- **Derived Column transform**: Define a new column based on a math formula or SQL expression in which you can use other columns in the data, as well as constants and literals.
- **Lookup transform**: Add columns from a defined catalog table when the keys match the defined lookup columns in the data.
- **Explode Array or Map Into Rows transform**: Extract values from a nested structure into individual rows that are easier to manipulate.
- **Record matching transform**: Invoke an existing Record Matching machine learning data classification transform.
- **Remove null rows transform**: Remove from the dataset rows that have all columns as null, or empty.
- **Parse JSON column transform**: Parse a string column containing JSON data and convert it to a struct or an array column, depending if the JSON is an object or an array, respectively.
- **Extract JSON path transform**: Extract new columns from a JSON string column.

- **Extract string fragments from a regular expression**: Extract string fragments using a regular expression and create new column out of it, or multiple columns if using regex groups.
- **Custom transform**: Enter code into a text entry field to use custom transforms. The output is a collection of `DynamicFrames`.

14. Transform data with custom visual transforms:

14.1 Steps for creating custom visual Transform:

Setting up the Amazon S3 bucket

Transforms you create are stored in Amazon S3 and is owned by your AWS account. You create new custom visual transforms by simply uploading files (json and py) to the Amazon S3 assets folder where all job scripts are currently stored (for example, `s3://aws-glue-assets-<accountid>-<region>/transforms`). If using a custom icon, upload it as well. By default, AWS Glue Studio will read all .json files from the /transforms folder in the same S3 bucket.

Step 1. Create a JSON config file:

JSON file structure

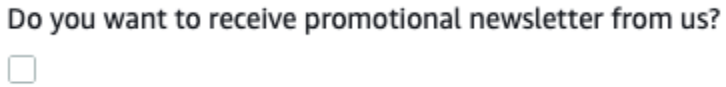
Fields

- `name: string` – (required) the transform system name used to identify transforms. Follow the same naming rules set for python variable names (identifiers). Specifically, they must start with either a letter or an underscore and then be composed entirely of letters, digits, and/or underscores.
- `displayName: string` – (optional) the name of the transform displayed in the AWS Glue Studio visual job editor. If no `displayName` is specified, the `name` is used as the name of the transform in AWS Glue Studio.
- `description: string` – (optional) the transform description is displayed in AWS Glue Studio and is searchable.
- `functionName: string` – (required) the Python function name is used to identify the function to call in the Python script.
- `listOptions: An array of TransformParameterListOption object` OR a `string` or the string value ‘column’ – (optional) options to display in Select or Multiselect UI control. Accepting a list of comma separated value or a strongly type JSON

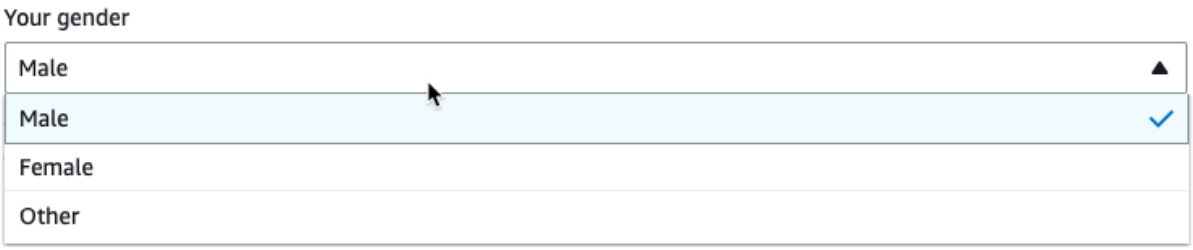
object of type `TransformParameterListOption`. It can also dynamically populate the list of columns from the parent node schema by specifying the string value “column”.

- `listType: string` – (optional) Define options types for type = 'list'. Valid values: 'str' | 'int' | 'float' | 'list' | 'bool'. Parameter type accepting common python data types.
- `validationType: string` – (optional) defines the way this parameter is validated. Currently, it only supports regular expressions. By default, the validation type is set to `RegularExpression`.
- `validationRule: string` – (optional) regular expression used to validate form input before submit when `validationType` is set to `RegularExpression`. Regular expression syntax must be compatible with [RegExp Ecmascript specifications].
- `validationMessage: string` – (optional) the message to display when validation fails.
- `type: string` – (required) the parameter type accepting common Python data types. Valid values: 'str' | 'int' | 'float' | 'list' | 'bool'.
- When `type` is any of the following: `str`, `int` or `float`, a text input field is displayed. For example, the screenshot shows input fields for 'Email Address' and 'Your age' parameters.

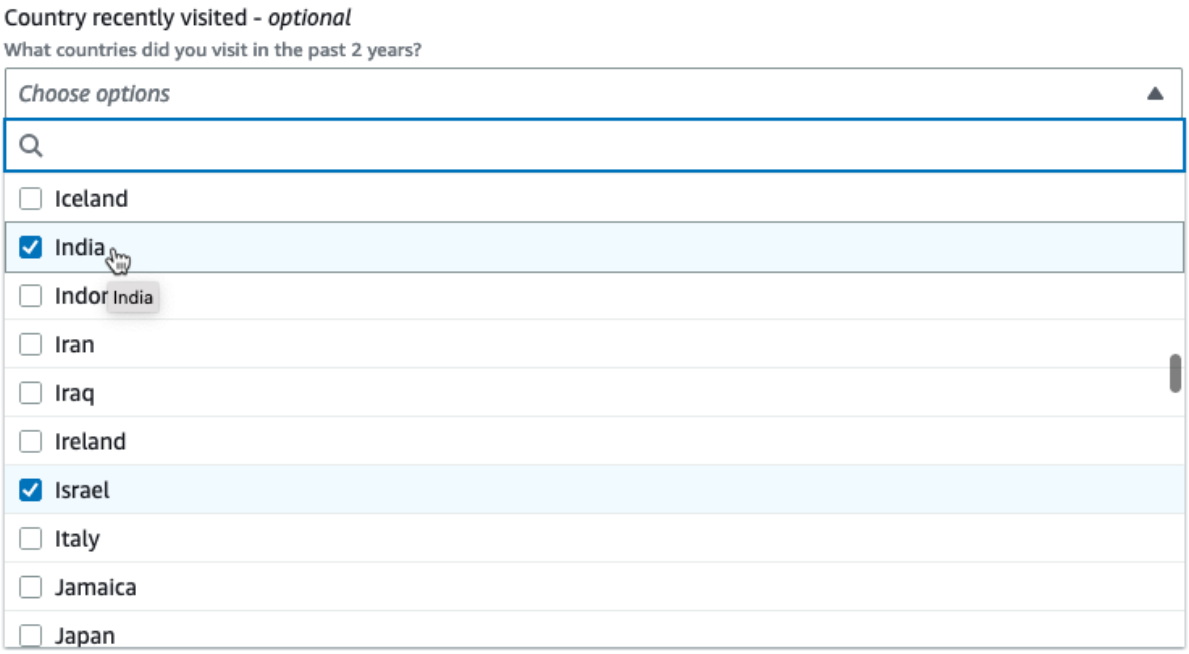
- When `type` is `bool`, a checkbox is displayed.

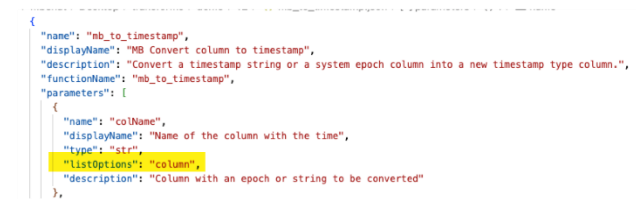


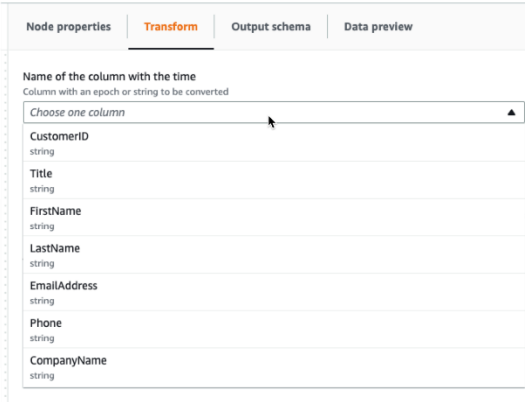
- When `type` is `str` and `listOptions` is provided, a single select list is displayed.



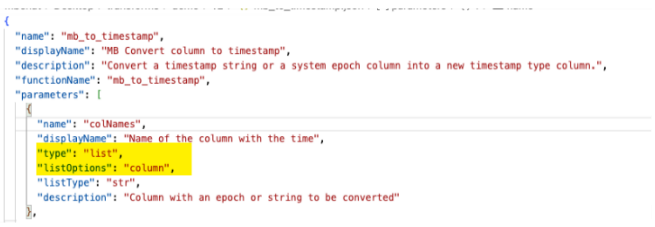
- When `type` is `list` and `listOptions` and `listType` are provided, a multi-select list is displayed.

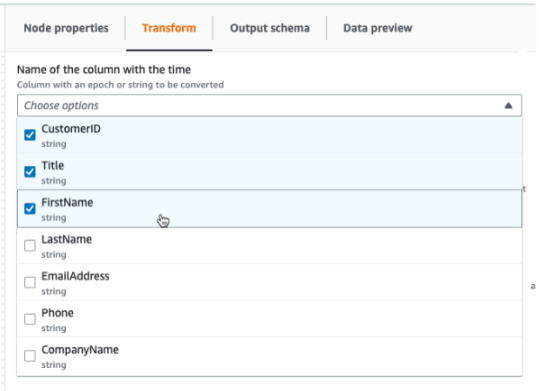






1. You can also allow multiple columns selection by defining the parameter as:
    - `listOptions: "column"`
    - `type: "list"`





Step 2. Implement the transform logic:

```
{
"name": "custom_filter_state",
"displayName": "Filter State",
"description": "A simple example to filter the data to keep only the
state indicated.",
"functionName": "custom_filter_state",
"parameters": [
   {
    "name": "colName",
    "displayName": "Column name",
    "type": "str",
    "description": "Name of the column in the data that holds the
state postal code"
   },
   {
    "name": "state",
    "displayName": "State postal code",
    "type": "str",
    "description": "The postal code of the state whole rows to keep"
   }
  ]
}
```

```
Python Script:

from awsglue import DynamicFrame

def custom_filter_state(self, colName, state):
    return self.filter(lambda row: row[colName] == state)

DynamicFrame.custom_filter_state = custom_filter_state
```

The following is an example of all possible parameters in a .json config file.

```
{
  "name": "MyTransform",
  "displayName": "My Transform",
  "description": "This transform description will be displayed in
UI",
  "functionName": "myTransform",
  "parameters": [
      {
      "name": "email",
      "displayName": "Email Address",
      "type": "str",
      "description": "Enter your work email address below",
      "validationType": "RegularExpression",
      "validationRule": "^\\w+([\\.-]?\\w+)*@\\w+([\\.-
]?\\w+)*(\\.\\w{2,3})+$",
      "validationMessage": "Please enter a valid email address"
    },
    {
      "name": "phone",
      "displayName": "Phone Number",
      "type": "str",
      "description": "Enter your mobile phone number below",
```

```
      "validationRule": "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[-
]?(\\d{4})$",
      "validationMessage": "Please enter a valid US number"
    },
    {
      "name": "age",
      "displayName": "Your age",
      "type": "int",
      "isOptional": true
    },
    {
      "name": "gender",
      "displayName": "Your gender",
      "type": "str",
      "listOptions": [
            {"label": "Male", "value": "male"},
            {"label": "Female", "value": "female"},
            {"label": "Other", "value": "other"}
        ],
      "isOptional": true
    },
    {
      "name": "country",
      "displayName": "Your origin country ?",
```

```
      "type": "list",

      "listOptions": "Afghanistan,Albania,Algeria,American
Samoa,Andorra,Angola,Anguilla,Antarctica,Antigua and
Barbuda,Argentina,Armenia,Aruba,Australia,Austria,Azerbaijan,Bahamas
,Bahrain,Bangladesh,Barbados,Belarus,Belgium,Belize,Benin,Bermuda,Bh
utan,Bolivia,Bosnia and Herzegovina,Botswana,Bouvet
Island,Brazil,British Indian Ocean Territory,Brunei
Darussalam,Bulgaria,Burkina
Faso,Burundi,Cambodia,Cameroon,Canada,Cape Verde,Cayman
Islands,Central African Republic,Chad,Chile,China,Christmas
Island,Cocos (Keeling Islands),Colombia,Comoros,Congo,Cook
Islands,Costa Rica,Cote D'Ivoire (Ivory Coast),Croatia
(Hrvatska,Cuba,Cyprus, ",

      "description": "What country were you born in?",

      "listType": "str",

      "isOptional": true

    },

    {

      "name": "promotion",

      "displayName": "Do you want to receive promotional newsletter
from us?",

      "type": "bool",

      "isOptional": true

    }

  ]

}
```

15. Working with Spark Jobs:

15.1 Run Spark code

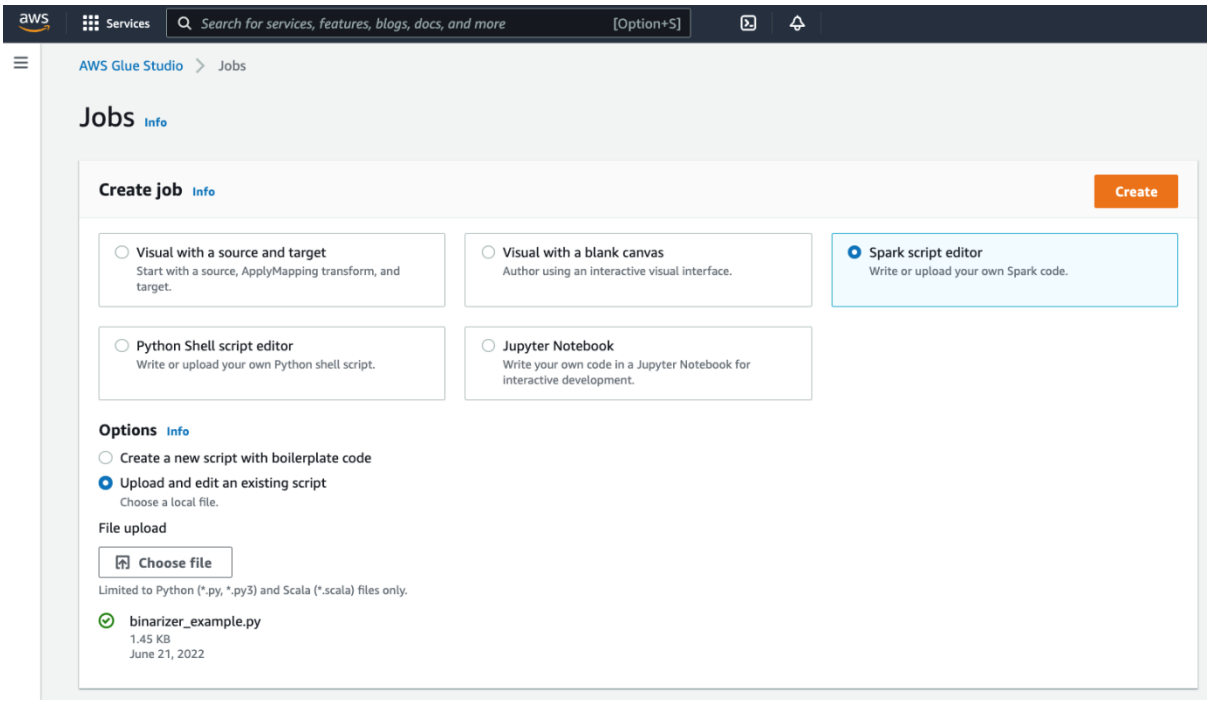
Native Spark code can be run in a AWS Glue environment out of the box. Scripts are often developed by iteratively changing a piece of code, a workflow suited for an Interactive Session. However, existing code is more suited to run in a AWS Glue job, which allows you to schedule and consistently get logs and metrics for each script run. You can upload and edit an existing script through the console.

1. Acquire the source to your script. For this example, you will use an example script from the Apache Spark repository. [Binarizer Example]
2. In the AWS Glue Console, expand the left-side navigation pane and select **ETL** > **Jobs**

   In the **Create job** panel, select **Spark script editor**. An **Options** section will appear. Under **Options**, select **Upload and edit an existing script**.

   A **File upload** section will appear. Under **File upload**, click **Choose file**. Your system file chooser will appear. Navigate to the location where you saved `binarizer_example.py`, select it and confirm your selection.

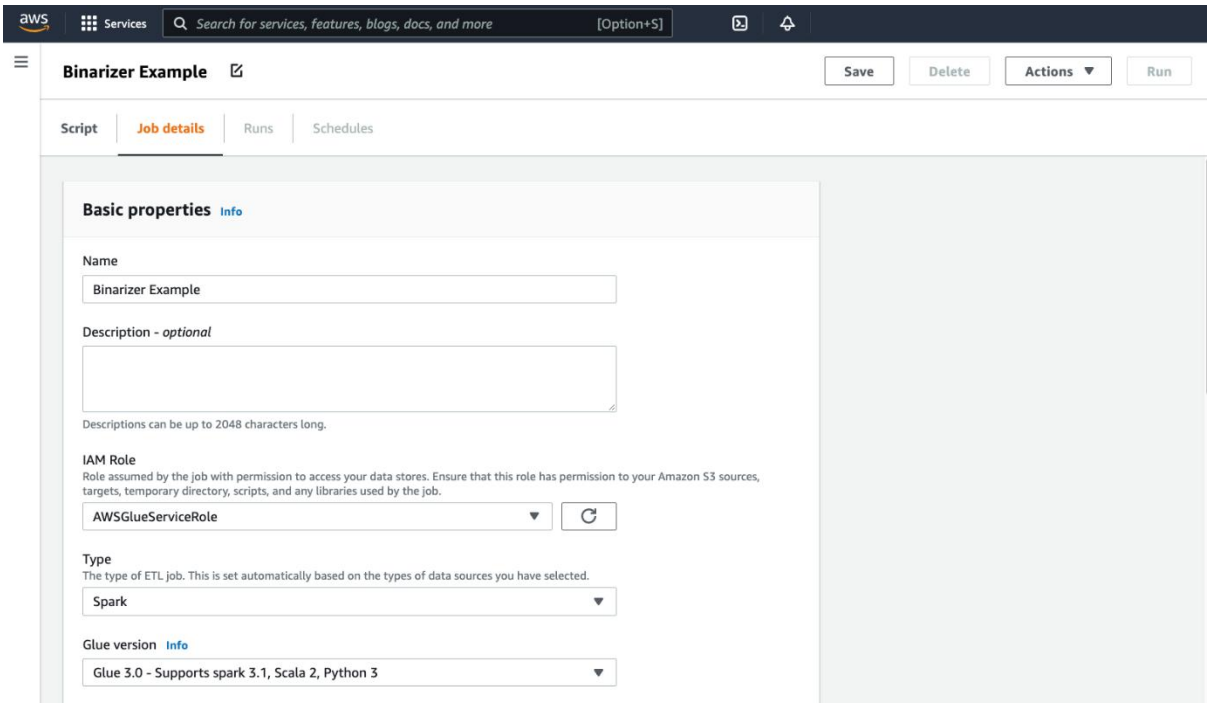
   A **Create** button will appear on the header for the **Create job** panel. Click it.



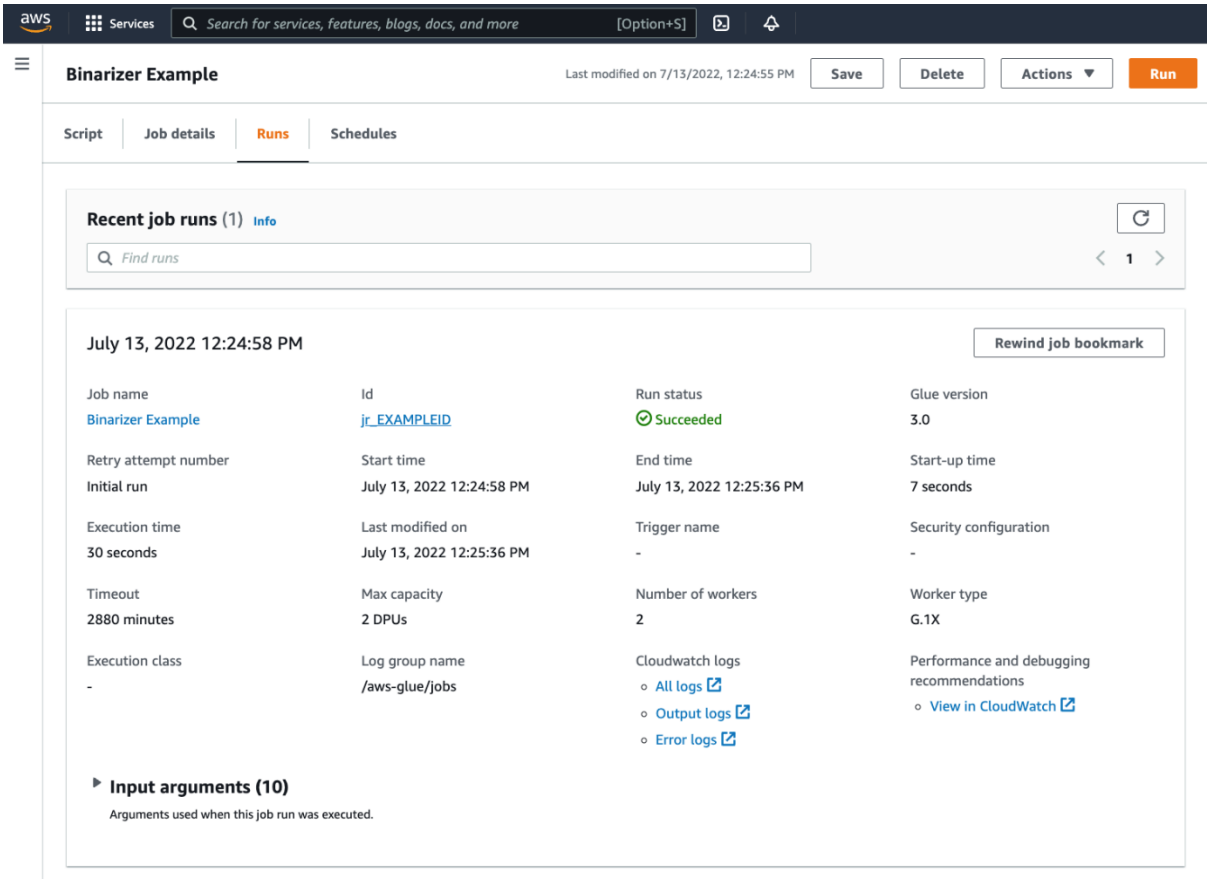
3. Your browser will navigate to the script editor. On the header, click the **Job details** tab. Set the **Name** and **IAM Role**. For guidance around AWS Glue IAM roles, consult [Setting up IAM permissions for AWS Glue].

Optionally - set **Requested number of workers** to `2` and **Number of retries** to `1`. These options are valuable when running production jobs, but turning them down will streamline your experience while testing out a feature.

In the title bar, click **Save**, then **Run**



4. Navigate to the **Runs** tab. You will see a panel corresponding to your job run. Wait a few minutes and the page should automatically refresh to show **Succeeded** under **Run status**.

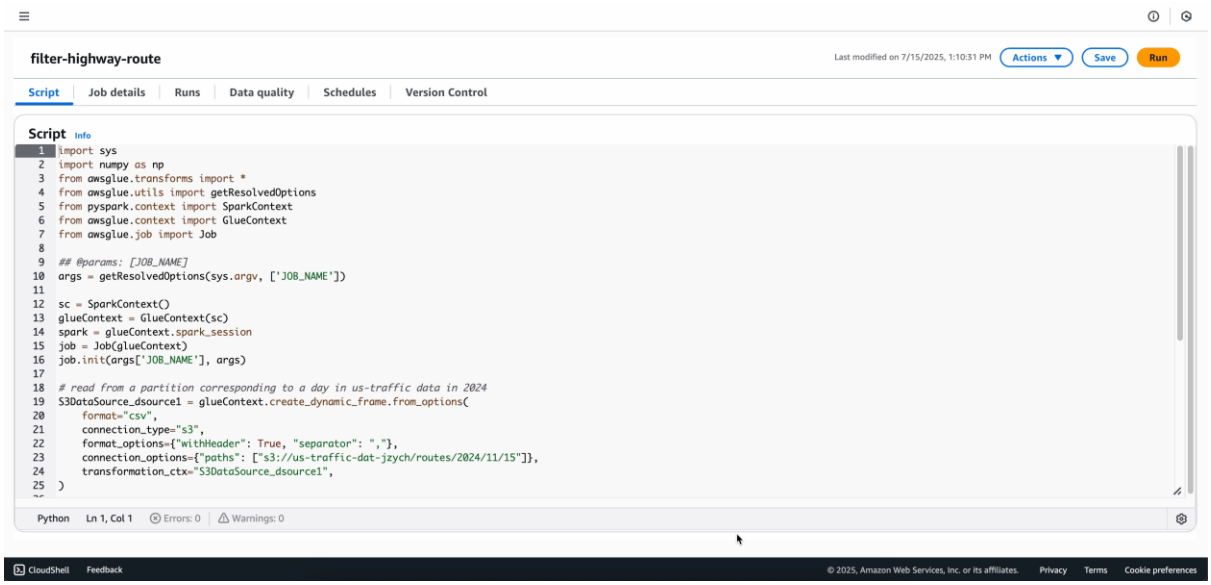


15.2 Running troubleshooting analysis from a failed job run

You can access the troubleshooting feature through multiple paths in the AWS Glue console. Here's how to get started:

Option 1: From the Jobs List page

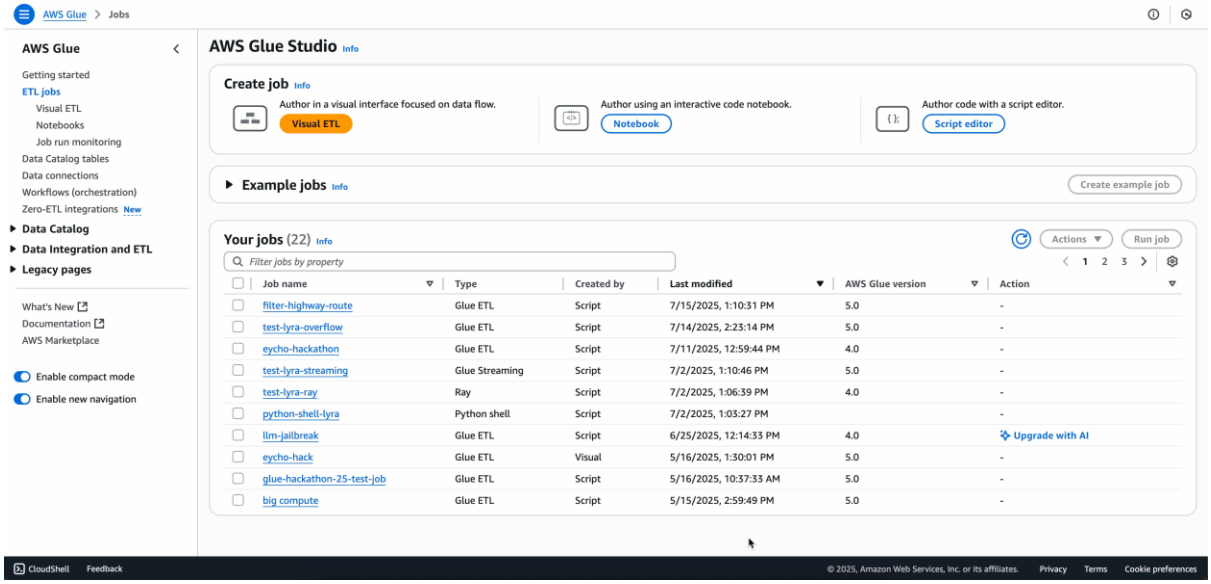
1. Open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, choose **ETL Jobs**.
3. Locate your failed job in the jobs list.
4. Select the **Runs** tab in the job details section.
5. Click on the failed job run you want to analyze.
6. Choose **Troubleshoot with AI** to start the analysis.
7. When the troubleshooting analysis is complete, you can view the root-cause analysis and recommendations in the **Troubleshooting analysis** tab at the bottom of the screen.



ABOUT

Option 2: Using the Job Run Monitoring page

1. Navigate to the **Job run monitoring** page.
2. Locate your failed job run.
3. Choose the **Actions** drop-down menu.
4. Choose **Troubleshoot with AI**.



ABOUT

Option 3: From the Job Run Details page

1. Navigate to your failed job run's details page by either clicking **View details** on a failed run from the **Runs** tab or selecting the job run from the **Job run monitoring** page.
2. In the job run details page, find the **Troubleshooting analysis** tab.

Available worker types

G.1X - Standard Worker

- **DPU**: 1 DPU (4 vCPUs, 16 GB memory)
- **Storage**: 94GB disk (approximately 44GB free)
- **Use Case**: Data transforms, joins, and queries - scalable and cost-effective for most jobs

G.2X - Standard Worker

- **DPU**: 2 DPU (8 vCPUs, 32 GB memory)
- **Storage**: 138GB disk (approximately 78GB free)
- **Use Case**: Data transforms, joins, and queries - scalable and cost-effective for most jobs

G.4X - Large Worker

- **DPU**: 4 DPU (16 vCPUs, 64 GB memory)
- **Storage**: 256GB disk (approximately 230GB free)
- **Use Case**: Demanding transforms, aggregations, joins, and queries

G.8X - Extra Large Worker

- **DPU**: 8 DPU (32 vCPUs, 128 GB memory)
- **Storage**: 512GB disk (approximately 485GB free)
- **Use Case**: Most demanding transforms, aggregations, joins, and queries

G.12X - Very Large Worker*

- **DPU**: 12 DPU (48 vCPUs, 192 GB memory)
- **Storage**: 768GB disk (approximately 741GB free)
- **Use Case**: Very large and resource-intensive workloads requiring significant compute capacity

G.16X - Maximum Worker*

- **DPU**: 16 DPU (64 vCPUs, 256 GB memory)
- **Storage**: 1024GB disk (approximately 996GB free)
- **Use Case**: Largest and most resource-intensive workloads requiring maximum compute capacity

R.1X - Memory-Optimized Small*

- **DPU**: 1 M-DPU (4 vCPUs, 32 GB memory)
- **Use Case**: Memory-intensive workloads with frequent out-of-memory errors or high memory-to-CPU ratio requirements

R.2X - Memory-Optimized Medium*

- **DPU**: 2 M-DPU (8 vCPUs, 64 GB memory)
- **Use Case**: Memory-intensive workloads with frequent out-of-memory errors or high memory-to-CPU ratio requirements

R.4X - Memory-Optimized Large*

- **DPU**: 4 M-DPU (16 vCPUs, 128 GB memory)
- **Use Case**: Large memory-intensive workloads with frequent out-of-memory errors or high memory-to-CPU ratio requirements

R.8X - Memory-Optimized Extra Large*

- **DPU**: 8 M-DPU (32 vCPUs, 256 GB memory)
- **Use Case**: Very large memory-intensive workloads with frequent out-of-memory errors or high memory-to-CPU ratio requirements

15.3 Configuring job properties for Python shell jobs in AWS Glue:

You can use a Python shell job to run Python scripts as a shell in AWS Glue. With a Python shell job, you can run scripts that are compatible with Python 3.6 or Python 3.9.

Defining job properties for Python shell jobs

These sections describe defining job properties in AWS Glue Studio, or using the AWS CLI.

AWS Glue Studio

When you define your Python shell job in AWS Glue Studio, you provide some of the following properties:

IAM role

Specify the AWS Identity and Access Management (IAM) role that is used for authorization to resources that are used to run the job and access data stores. For more information about permissions for running jobs in AWS Glue, see Identity and access management for AWS Glue.

Type

Choose **Python shell** to run a Python script with the job command named `pythonshell`.

Python version

Choose the Python version. The default is Python 3.9. Valid versions are Python 3.6 and Python 3.9.

Load common analytics libraries (Recommended)

Choose this option to include common libraries for Python 3.9 in the Python shell.

If your libraries are either custom or they conflict with the pre-installed ones, you can choose not to install common libraries. However, you can install additional libraries besides the common libraries.

When you select this option, the `library-set` option is set to `analytics`. When you de-select this option, the `library-set` option is set to `none`.

Script filename and Script path

The code in the script defines your job's procedural logic. You provide the script name and location in Amazon Simple Storage Service (Amazon S3). Confirm that there isn't a file with the same name as the script directory in the path. To learn more about using scripts, see AWS Glue programming guide.

Script

The code in the script defines your job's procedural logic. You can code the script in Python 3.6 or Python 3.9. You can edit a script in AWS Glue Studio.

Data processing units

The maximum number of AWS Glue data processing units (DPUs) that can be allocated when this job runs. A DPU is a relative measure of processing power that consists of 4 vCPUs of compute capacity and 16 GB of memory. For more information, see AWS Glue pricing.

You can set the value to 0.0625 or 1. The default is 0.0625. In either case, the local disk for the instance will be 20GB.

Supported libraries for Python shell jobs

In Python shell using Python 3.9, you can choose the library set to use pre-packaged library sets for your needs. You can use the `library-set` option to choose the library set. Valid values are `analytics`, and `none`.

The environment for running a Python shell job supports the following libraries:

Python version	**Python 3.6**	**Python 3.9**	
Library set	**N/A**	**analytics**	**none**
avro		1.11.0	
awscli	116.242	1.23.5	1.23.5
awswrangler		2.15.1	
botocore	1.12.232	1.24.21	1.23.5
boto3	1.9.203	1.21.21	
elasticsearch		8.2.0	
numpy	1.16.2	1.22.3	
pandas	0.24.2	1.4.2	
psycopg2		2.9.3	
pyathena		2.5.3	
PyGreSQL	5.0.6		
PyMySQL		1.0.2	
pyodbc		4.0.32	

Python version	Python 3.6	Python 3.9	
pyorc		0.6.0	
redshift-connector		2.0.907	
requests	2.22.0	2.27.1	
scikit-learn	0.20.3	1.0.2	
scipy	1.2.1	1.8.0	
SQLAlchemy		1.4.36	
s3fs		2022.3.0	

16. AWS Glue Streaming:

AWS Glue Streaming is a **serverless data integration service** designed for **real-time streaming data processing**. It leverages **Apache Spark Streaming** and offers several enhancements:

- **Real-time data ingestion and processing** for tasks like analytics and machine learning.
- **Serverless infrastructure**: No need to manage servers.
- **Auto-scaling**: Automatically adjusts resources based on workload.
- **Visual job development**: Simplifies building and managing streaming jobs.
- **Instant-on notebooks**: Quickly prototype and test streaming workflows.
- **Performance optimizations**: Improves efficiency over standard Apache Spark.

Use cases for streaming

Some common use cases for AWS Glue Streaming include:

Near-real-time data processing: AWS Glue Streaming allows organizations to process streaming data in near real-time, enabling them to derive insights and make timely decisions based on the latest information.

Fraud detection: You can utilize AWS Glue Streaming for real-time analysis of streaming data, making it valuable for detecting fraudulent activities, such as credit card fraud, network intrusion, or online scams. By continuously processing and analyzing incoming data, you can swiftly identify suspicious patterns or anomalies.

Social media analytics: AWS Glue Streaming can process real-time social media data, such as tweets, posts, or comments, enabling organizations to monitor trends, sentiment analysis, and manage brand reputation in real-time.

Internet of Things (IoT) analytics: AWS Glue Streaming is suitable for handling and analyzing high-velocity streams of data generated by IoT devices, sensors, and connected machinery. It allows for real-time monitoring, anomaly detection, predictive maintenance, and other IoT analytics use cases.

Clickstream analysis: AWS Glue Streaming can process and analyze real-time clickstream data from websites or mobile applications. This enables businesses to gain insights into user behavior, personalize user experiences, and optimize marketing campaigns based on real-time clickstream data.

Log monitoring and analysis: AWS Glue Streaming can continuously process and analyze log data from servers, applications, or network devices in real-time. This helps in detecting anomalies, troubleshooting issues, and monitoring system health and performance.

Recommendation systems: AWS Glue Streaming can process user activity data in real-time and update recommendation models dynamically. This allows for personalized and real-time recommendations based on user behavior and preferences.

Supported data sources

AWS Glue Streaming supports the following data sources:

- Amazon Kinesis
- Amazon MSK (Managed Streaming for Apache Kafka)
- Self-managed Apache Kafka

Supported data targets

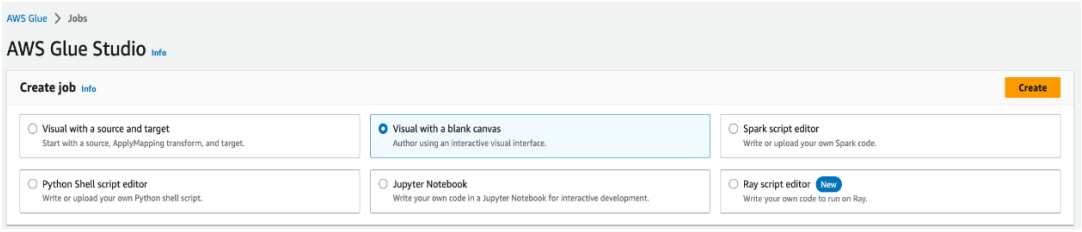
AWS Glue Streaming supports a variety of data targets such as:

- Data targets supported by AWS Glue Data Catalog
- Amazon S3
- Amazon Redshift
- MySQL
- PostgreSQL
- Oracle
- Microsoft SQL Server
- Snowflake
- Any database that can be connected using JDBC
- Apache Iceberg, Delta and Apache Hudi
- AWS Glue Marketplace connectors

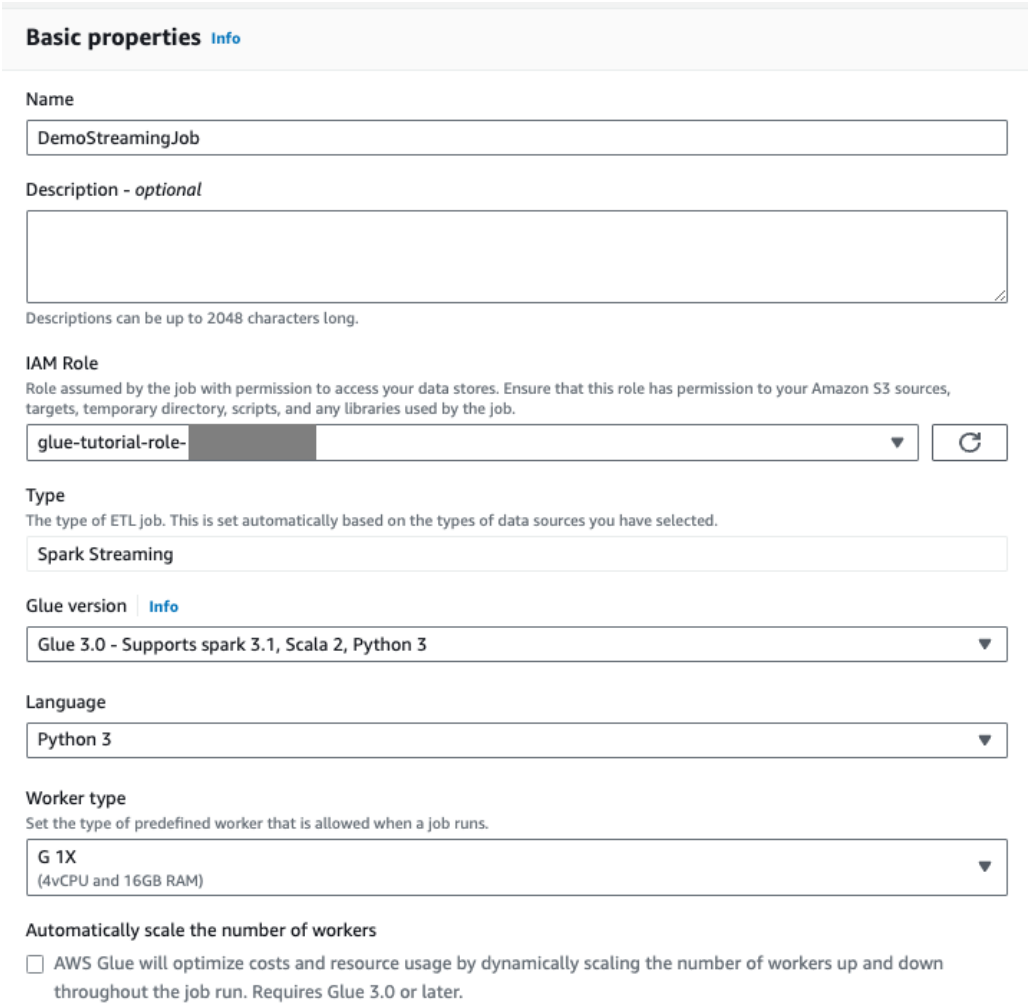
16.1 Tutorial: Build your first streaming workload using AWS Glue Studio:

Creating an AWS Glue streaming job with AWS Glue Studio

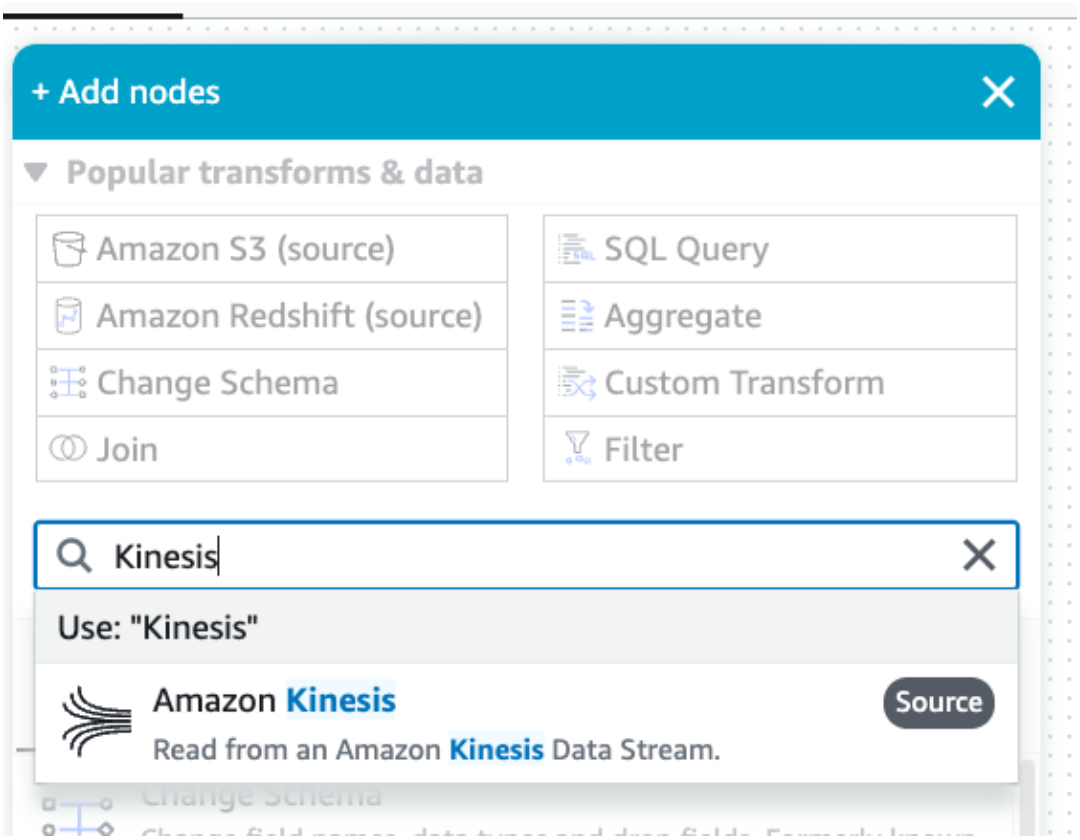
1. Navigate to AWS Glue in the console on the same Region.
2. Select **ETL jobs** under the left side navigation bar under **Data Integration and ETL**.
3. Create an AWS Glue Job via **Visual with a blank canvas**.



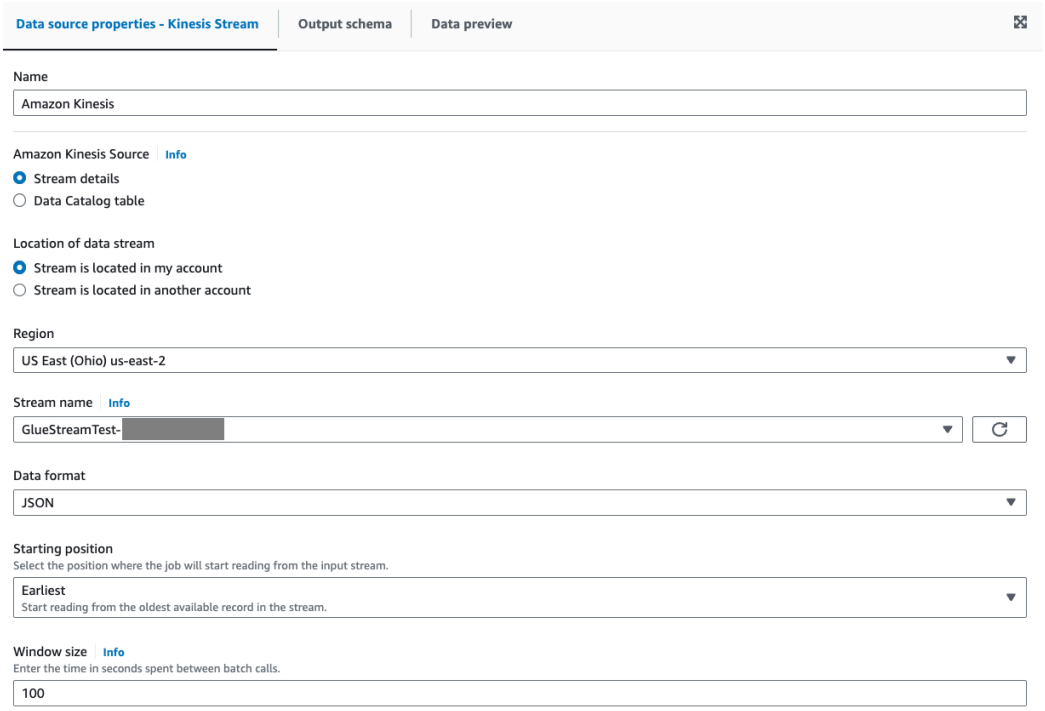
4. Navigate to the **Job Details** tab.
5. For the AWS Glue job name, enter `DemoStreamingJob`.
6. For **IAM Role**, select the role provisioned by the CloudFormation template, `glue-tutorial-role-${AWS::AccountId}`.
7. For **Glue version**, select **Glue 3.0**. Leave all other options as default.



8. Navigate to the **Visual tab**.
9. Click on the plus icon. Enter **Kinesis** in the search bar. Select the **Amazon Kinesis** data source.



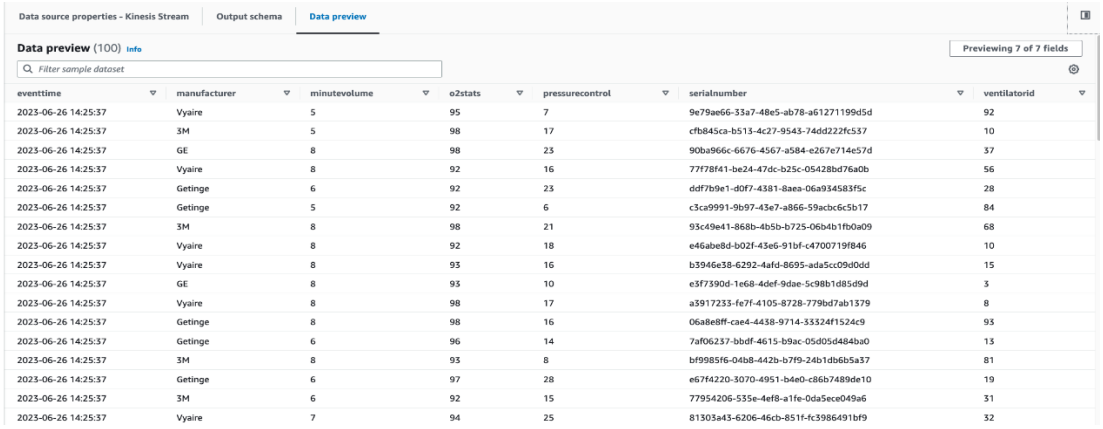
10. Select **Stream details** for **Amazon Kinesis Source** under the tab **Data source properties - Kinesis Stream**.
11. Select **Stream is located in my account** for **Location of data stream**.
12. Select the Region you are using.
13. Select the `GlueStreamTest-{AWS::AccountId}` stream.
14. Keep all other settings as default.



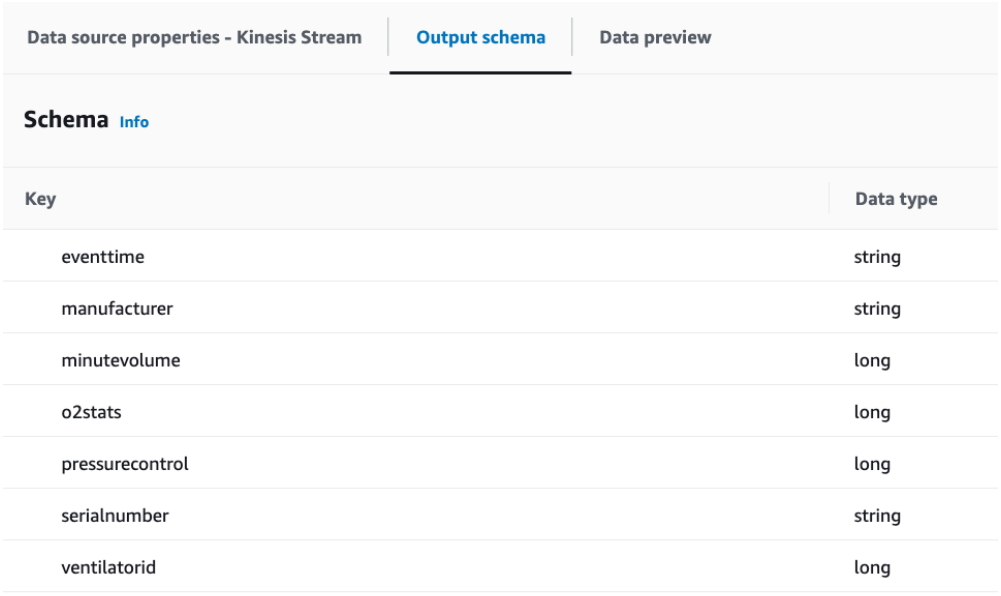
15. Navigate to the **Data preview** tab.
16. Click **Start data preview session**, which previews the mock data generated by KDG. Pick the Glue Service Role you previously created for the AWS Glue Streaming job.

    It takes 30-60 seconds for the preview data to show up. If it shows **No data to display**, click the gear icon and change the **Number of rows to sample** to `100`.

    You can see the sample data as below:

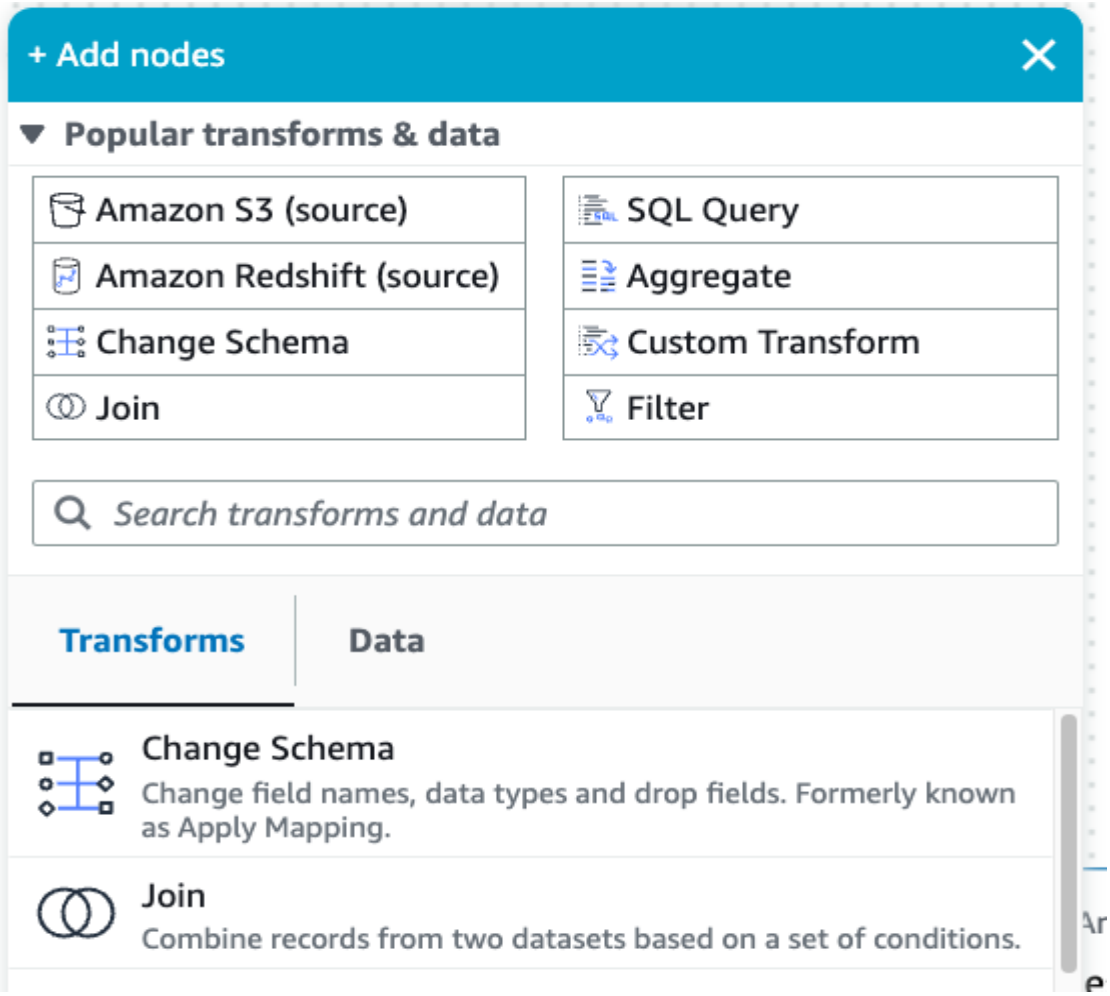


    You can also see the inferred schema in the **Output schema** tab.

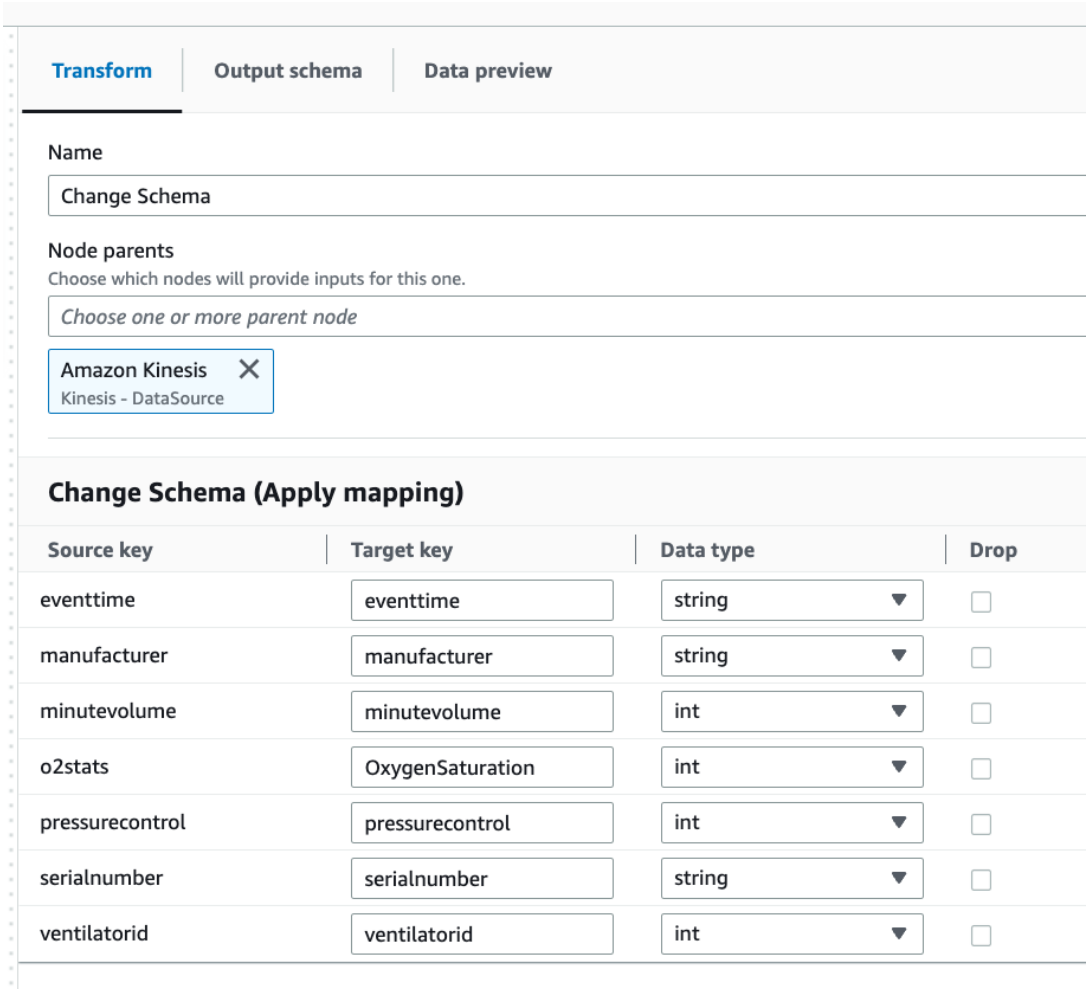


Performing a transformation and storing the transformed result in Amazon S3

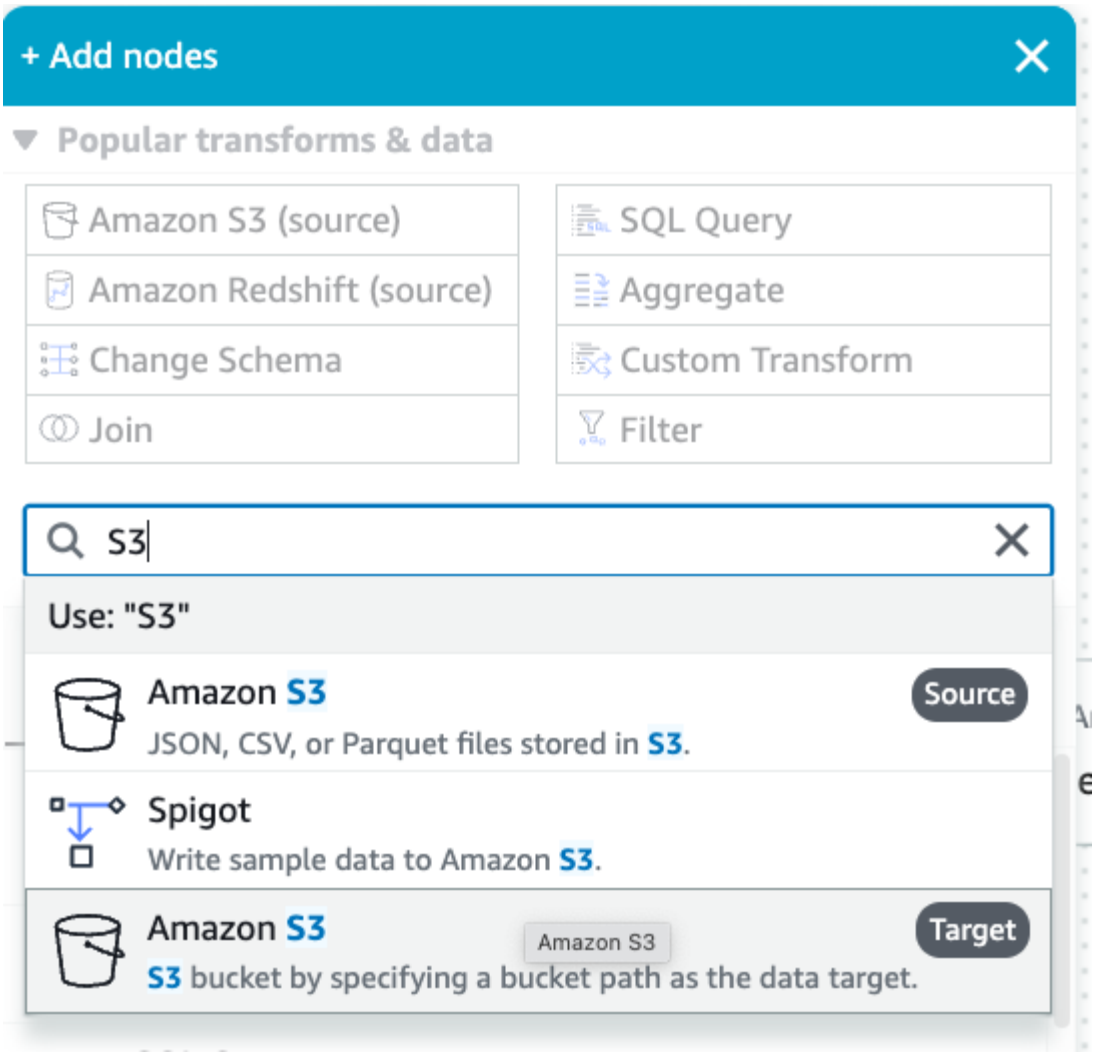
17. With the source node selected, click on the plus icon on the top left to add a **Transforms** step.
18. Select the **Change Schema** step.



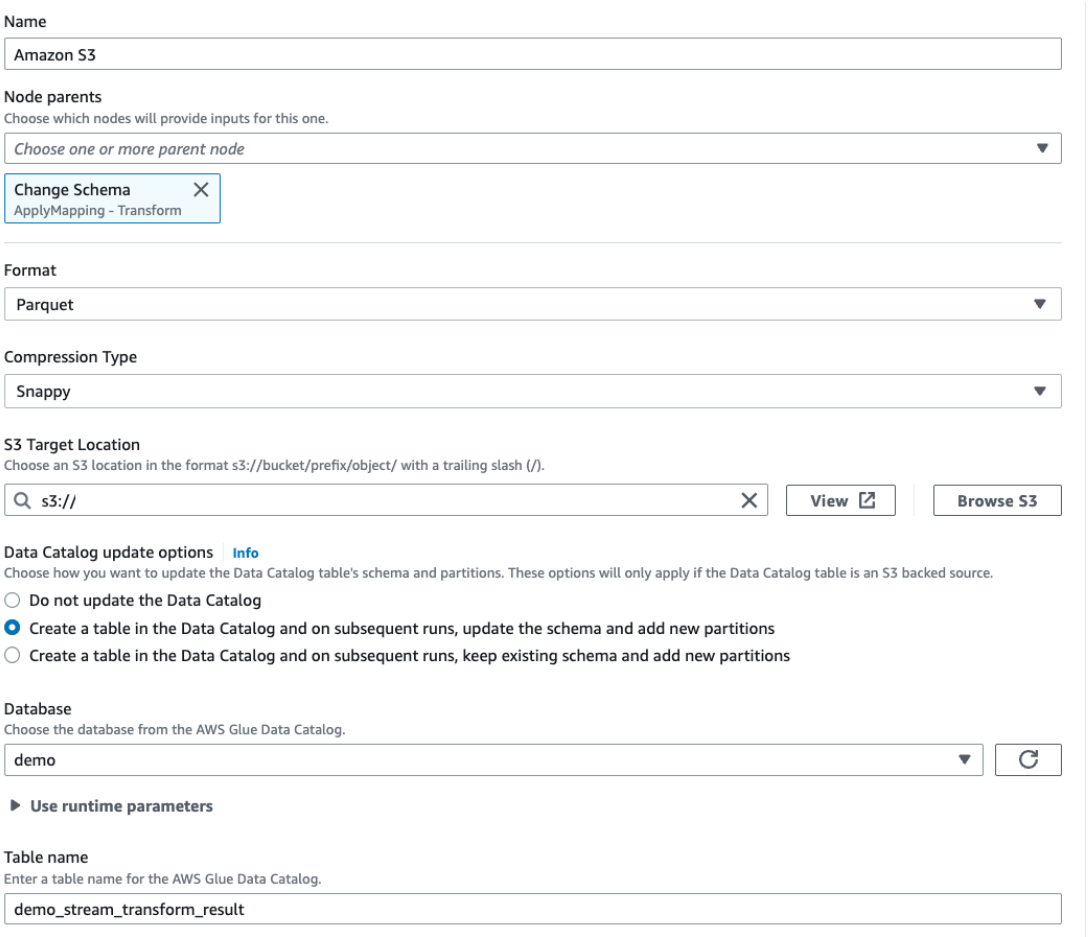
19. You can rename fields and convert the data type of fields in this step. Rename the `o2stats` column to `OxygenSaturation` and convert all `long` data type to `int`.



20. Click on the plus icon to add an **Amazon S3** target. Enter S3 in the search box and select the **Amazon S3 - Target** transform step.

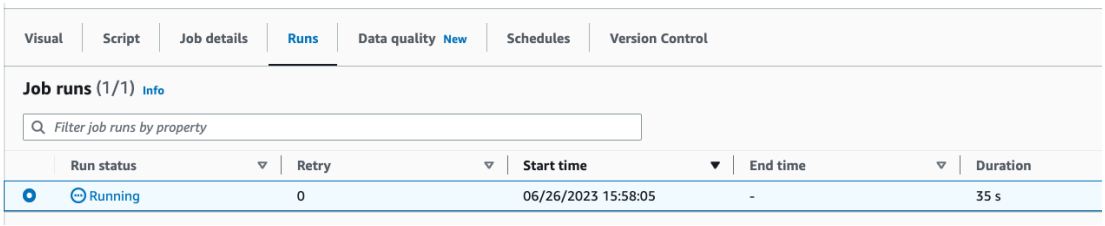


21. Select **Parquet** as the target file format.
22. Select **Snappy** as the compression type.
23. Enter an **S3 Target Location** created by the CloudFormation template, `streaming-tutorial-s3-target-{AWS::AccountId}`.
24. Select to **Create a table in the Data Catalog and on subsequent runs, update the schema and add new partitions**.
25. Enter the target **Database** and **Table** name to store the schema of the Amazon S3 target table.



26. Click on the **Script** tab to view the generated code.
27. Click **Save** on the top right to save the ETL code and then click **Run** to kick-off the AWS Glue streaming job.

    You can find the **Run status** in the **Runs** tab. Let the job run for 3-5 minutes and then stop the job.

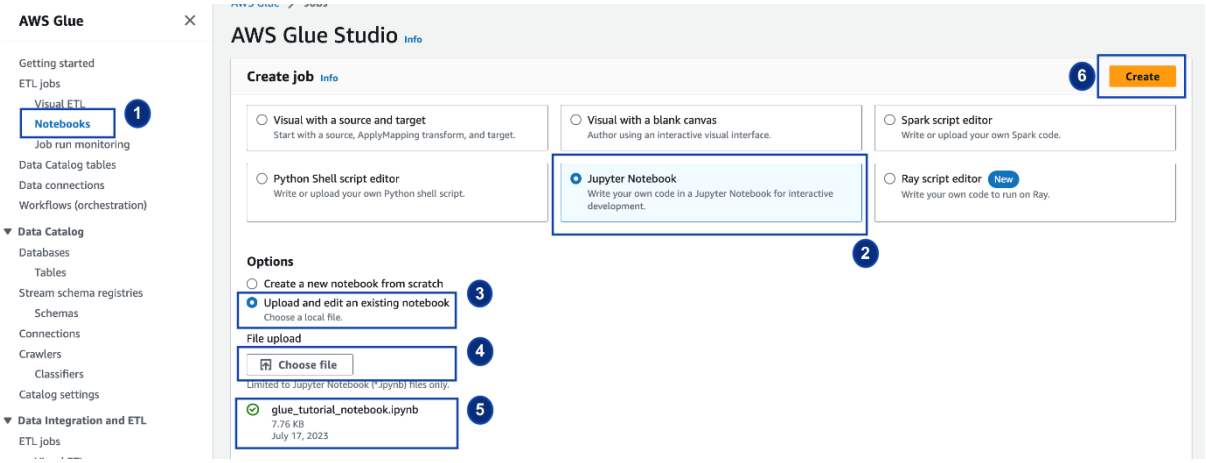


16.2 Tutorial: Build your first streaming workload using AWS Glue Studio notebooks

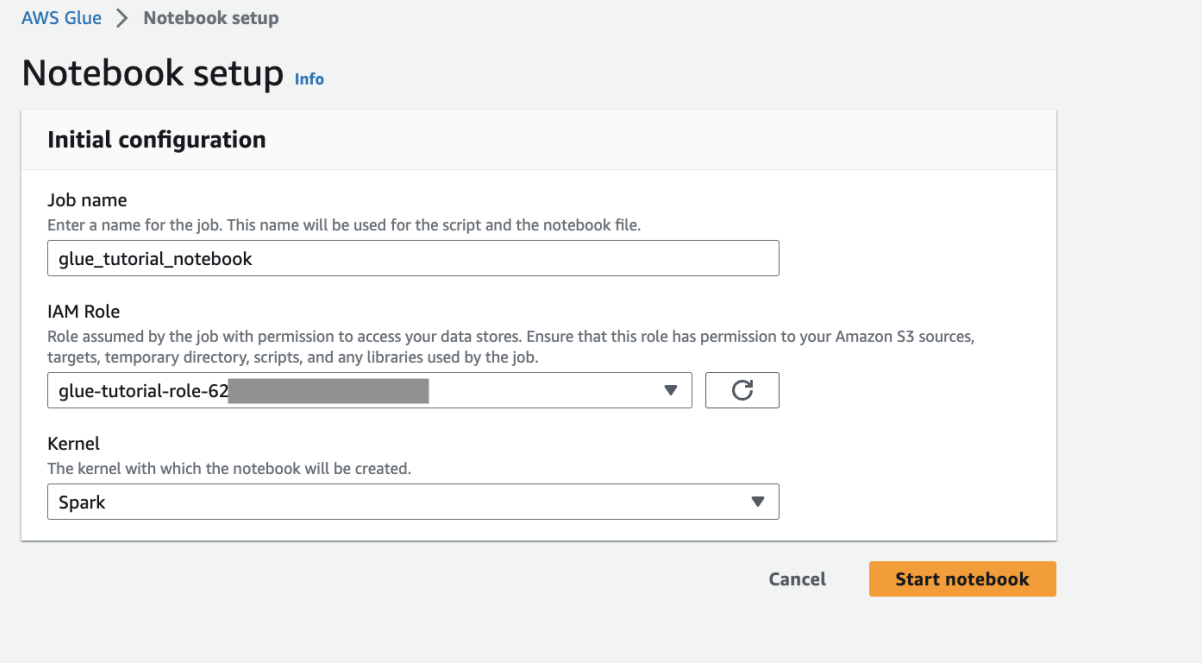
AWS Glue Studio is a visual interface that simplifies the process of designing, orchestrating, and monitoring data integration pipelines. It enables users to build data transformation pipelines without writing extensive code. Apart from the visual job authoring experience, AWS Glue Studio also includes a Jupyter notebook backed by AWS Glue Interactive sessions, which you will be using in the remainder of this tutorial.

Set up the AWS Glue Streaming interactive sessions job

1. Download the provided notebook file and save it to a local directory
2. Open the AWS Glue Console and on the left pane click **Notebooks** > **Jupyter Notebook** > **Upload and edit an existing notebook**. Upload the notebook from the previous step and click **Create**.



3. Provide the job a name, role and select the default Spark kernel. Next click **Start notebook**. For the **IAM Role**, select the role provisioned by the CloudFormation template. You can see this in the **Outputs** tab of CloudFormation.



The notebook has all necessary instructions to continue the tutorial. You can either run the instructions on the notebook or follow along with this tutorial to continue with the job development.

Run the notebook cells

1. (Optional) The first code cell, `%help` lists all available notebook magics. You can skip this cell for now, but feel free to explore it.
2. Start with the next code block `%streaming`. This magic sets the job type to streaming which lets you develop, debug and deploy an AWS Glue streaming ETL job.
3. Run the next cell to create an AWS Glue interactive session. The output cell has a message that confirms the session creation.

Run this cell to set up and start your interactive session.

```
[1]: %glue_version 3.0

import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue import DynamicFrame
from datetime import datetime
from pyspark.sql.types import StructType, StructField, StringType, LongType
from pyspark.sql.functions import lit,col,from_json
import boto3

sc = SparkContext.getOrCreate()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
```

```
Setting Glue version to: 3.0
Authenticating with environment variables and user-defined glue_role_arn: arn:aws:iam::62        10:role/glue-tutorial-role
Trying to create a Glue session for the kernel.
Worker Type: G.1X
Number of Workers: 5
Session ID:                                    af
Job Type: gluestreaming
Applying the following default arguments:
--glue_kernel_version 0.37.3
--enable-glue-datacatalog true
Waiting for session 4                                  to get into ready status...
Session 48                                    has been created.
```

4. The next cell defines the variables. Replace the values with ones appropriate to your job and run the cell. For example:

```
output_database_name="default"
output_table_name="test_stream_001"

account_id = boto3.client("sts").get_caller_identity()["Account"]
region_name=boto3.client('s3').meta.region_name
stream_arn_name = "arn:aws:kinesis:{}:{}:stream/GlueStreamTest-{}".format(region_name,account_id,account_id)
s3_bucket_name = "streaming-tutorial-s3-target-{}".format(account_id)

output_location = "s3://{}/streaming_output/".format(s3_bucket_name)
checkpoint_location = "s3://{}/checkpoint_location/".format(s3_bucket_name)
```

5. Since the data is being streamed already to Kinesis Data Streams, your next cell will consume the results from the stream. Run the next cell. Since there are no print statements, there is no expected output from this cell.
6. In the following cell, you explore the incoming stream by taking a sample set and print its schema and the actual data. For example:

Sample and print the incoming records

the sampling is for debugging purpose. You may comment off the entire code cell below, before deploying the actual code

```
[4]: options = {
    "pollingTimeInMs": "20000",
    "windowSize": "5 seconds"
}
sampled_dynamic_frame = glueContext.getSampleStreamingDynamicFrame(data_frame, options, None)

count_of_sampled_records = sampled_dynamic_frame.count()

print(count_of_sampled_records)

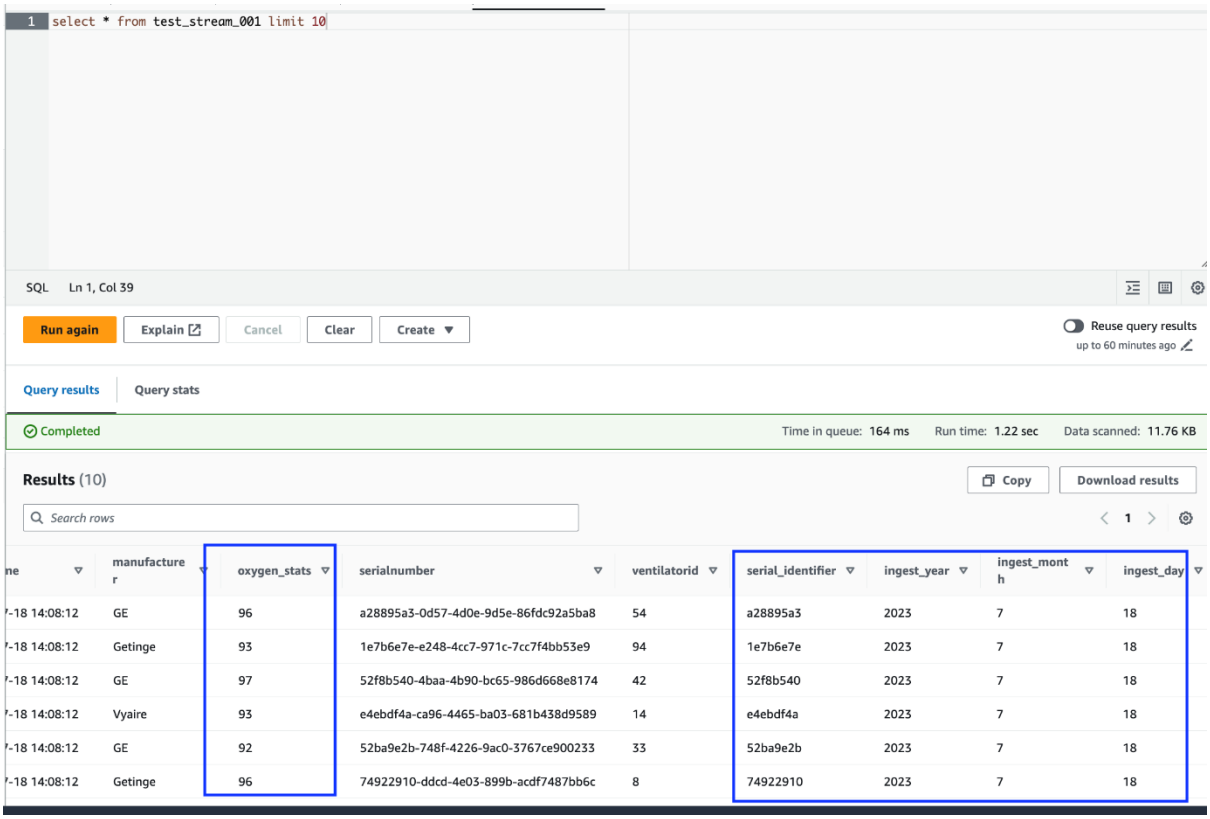
sampled_dynamic_frame.printSchema()

sampled_dynamic_frame.toDF().show(10,False)
```

```
100
root
|-- eventtime: string
|-- manufacturer: string
|-- minutevolume: long
|-- o2stats: long
|-- pressurecontrol: long
|-- serialnumber: string
|-- ventilatorid: long

+-------------------+------------+------------+-------+---------------+------------------------------------+------------+
|eventtime          |manufacturer|minutevolume|o2stats|pressurecontrol|serialnumber                        |ventilatorid|
+-------------------+------------+------------+-------+---------------+------------------------------------+------------+
|2023-07-18 10:20:11|3M          |6           |92     |24             |a3e860ba-24b9-41c4-bc10-91c6b35e1406|6           |
|2023-07-18 10:20:11|Vyaire      |6           |95     |6              |96101dca-3e88-457f-b390-e3291df48a81|26          |
|2023-07-18 10:20:12|Getinge     |8           |96     |24             |18f3d448-1dee-4c80-835b-1a0daa818915|22          |
|2023-07-18 10:20:12|Getinge     |7           |98     |30             |25f425cd-b978-4953-9a03-4d607a639364|91          |
|2023-07-18 10:20:12|GE          |5           |93     |25             |2cd7cdc2-f5f5-4ff2-ae32-45e5a8922d53|93          |
```

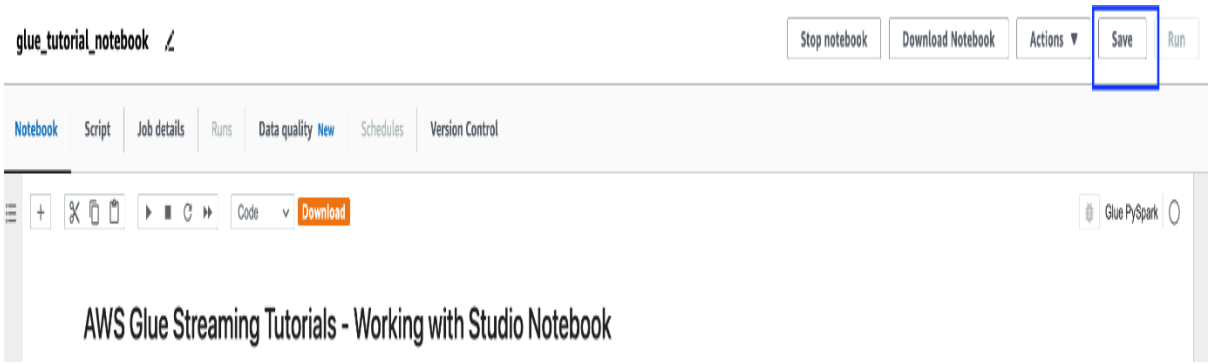
7. Next, define the actual data transformation logic. The cell consists of the `processBatch` method that is triggered during every micro-batch. Run the cell. At a high level, we do the following to the incoming stream:
    a. Select a subset of the input columns.
    b. Rename a column (o2stats to oxygen_stats).
    c. Derive new columns (serial_identifier, ingest_year, ingest_month and ingest_day).
    d. Store the results into an Amazon S3 bucket and also create a partitioned AWS Glue catalog table
8. In the last cell, you trigger the process batch every 10 seconds. Run the cell and wait for about 30 seconds for it to populate the Amazon S3 bucket and the AWS Glue catalog table.
9. Finally, browse the stored data using the Amazon Athena query editor. You can see the renamed column and also the new partitions.



The notebook has all necessary instructions to continue the tutorial. You can either run the instructions on the notebook or follow along with this tutorial to continue with the job development.

Save and run the AWS Glue job

With the development and testing of your application complete using the interactive sessions notebook, click **Save** at the top of the notebook interface. Once saved you can also run the application as a job.



Anatomy of a AWS Glue streaming job

AWS Glue streaming jobs operate on the Spark streaming paradigm and leverage structured streaming from the Spark framework. Streaming jobs constantly poll on the streaming data source, at a specific interval of time, to fetch records as micro batches. The following sections examine the different parts of a AWS Glue streaming job.

forEachBatch

The `forEachBatch` method is the entry point of a AWS Glue streaming job run. AWS Glue streaming jobs uses the `forEachBatch` method to poll data functioning like an iterator that remains active during the lifecycle of the streaming job and regularly polls the streaming source for new data and processes the latest data in micro batches.

```
glueContext.forEachBatch(

    frame=dataFrame_AmazonKinesis_node1696872487972,
```

```
    batch_function=processBatch,
    options={
        "windowSize": "100 seconds",
        "checkpointLocation": args["TempDir"] + "/" +
args["JOB_NAME"] + "/checkpoint/",
    },
)
```

Configure the `frame` property of `forEachBatch` to specify a streaming source. In this example, the source node that you created in the blank canvas during job creation is populated with the default DataFrame of the job. Set the `batch_function` property as the `function` that you decide to invoke for each micro batch operation. You must define a function to handle the batch transformation on the incoming data.

Source

In the first step of the `processBatch` function, the program verifies the record count of the DataFrame that you defined as frame property of `forEachBatch`. The program appends an ingestion time stamp to a non-empty DataFrame.
The `data_frame.count()>0` clause determines whether the latest micro batch is not empty and is ready for further processing.

```
def processBatch(data_frame, batchId):
    if data_frame.count() >0:
        AmazonKinesis_node1696872487972 = DynamicFrame.fromDF(
            glueContext.add_ingestion_time_columns(data_frame,
"hour"),
            glueContext,
            "from_data_frame",
        )
```

Mapping

The next section of the program is to apply mapping. The `Mapping.apply` method on a spark DataFrame allows you to define transformation rule around data elements. Typically you can rename, change the data type, or apply a custom function on the source data column and map those to the target columns.

```
#Script generated for node ChangeSchema

    ChangeSchema_node16986872679326 = ApplyMapping.apply(

        frame =  AmazonKinesis_node1696872487972,

        mappings = [

            ("eventtime", "string", "eventtime", "string"),

            ("manufacturer", "string", "manufacturer", "string"),

            ("minutevolume", "long", "minutevolume", "int"),

            ("o2stats", "long", "OxygenSaturation", "int"),

            ("pressurecontrol", "long", "pressurecontrol", "int"),

            ("serialnumber", "string", "serialnumber", "string"),

            ("ventilatorid", "long", "ventilatorid", "long"),

            ("ingest_year", "string", "ingest_year", "string"),

            ("ingest_month", "string", "ingest_month", "string"),

            ("ingest_day", "string", "ingest_day", "string"),

            ("ingest_hour", "string", "ingest_hour", "string"),

        ],

        transformation_ctx="ChangeSchema_node16986872679326",)

        )
```

Sink

In this section, the incoming data set from the streaming source are stored at a target location. In this example we will write the data to an Amazon S3 location. The `AmazonS3_node_path` property details is pre-populated as determined by the settings you used during job creation from the canvas. You can set the `updateBehavior` based on your use case and decide to either Not update the data catalog table, or Create data catalog and update data catalog schema on subsequent runs, or create a catalog table and not update the schema definition on subsequent runs.

The `partitionKeys` property defines the storage partition option. The default behavior is to partition the data per the `ingestion_time_columns` that was made available in the source section. The `compression` property allows you to set the compression algorithm to be applied during target write. You have options to set Snappy, LZO, or GZIP as the compression technique. The `enableUpdateCatalog` property controls whether the AWS Glue catalog table needs to be updated. Available options for this property are `True` or `False`.

```
    #Script generated for node Amazon S3
    AmazonS3_node1696872743449 = glueContext.getSink(
        path =  AmazonS3_node1696872743449_path,
        connection_type = "s3",
        updateBehavior = "UPDATE_IN_DATABASE",
        partitionKeys = ["ingest_year", "ingest_month",
"ingest_day", "ingest_hour"],
        compression = "snappy",
        enableUpdateCatalog = True,
        transformation_ctx = "AmazonS3_node1696872743449",
    )
```

AWS Glue Catalog sink

This section of the job controls the AWS Glue catalog table update behavior. Set `catalogDatabase` and `catalogTableName` property per your AWS Glue Catalog database name and the table name associated with the AWS Glue job that you are designing. You can define the file format of the target data via the `setFormat` property. For this example we will store the data in parquet format.

Once you set up and run the AWS Glue streaming job referring this tutorial, the streaming data produced at Amazon Kinesis Data Streams will be stored at the Amazon S3 location in a parquet format with snappy compression. On successful runs of the streaming job you will able to query the data through Amazon Athena.

```
    AmazonS3_node1696872743449 = setCatalogInfo(

        catalogDatabase =  "demo", catalogTableName =
"demo_stream_transform_result"

    )

    AmazonS3_node1696872743449.setFormat("glueparquet")


AmazonS3_node1696872743449.writeFormat("ChangeSchema_node16986872679
326")

    )
```

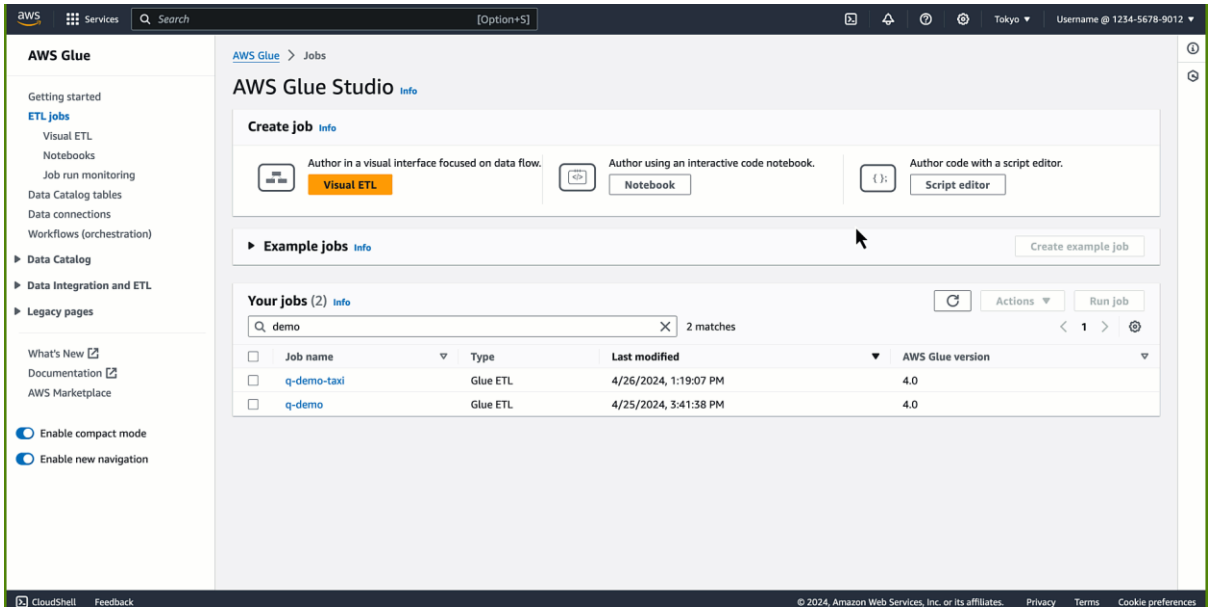
17. Amazon Q data integration in AWS Glue:

Amazon Q is a generative artificial intelligence (AI) powered conversational assistant that can help you understand, build, extend, and operate AWS applications.

17.1 Working with Amazon Q data integration in AWS Glue?

In the Amazon Q panel you can request Amazon Q generate code for an AWS Glue ETL script, or answer a question on AWS Glue features or troubleshooting an error. The response is an ETL script in PySpark with step-by-step instructions to customize the script, review and execute it. For questions, the response is generated based on the data integration knowledge base with a summary and source URL for references.

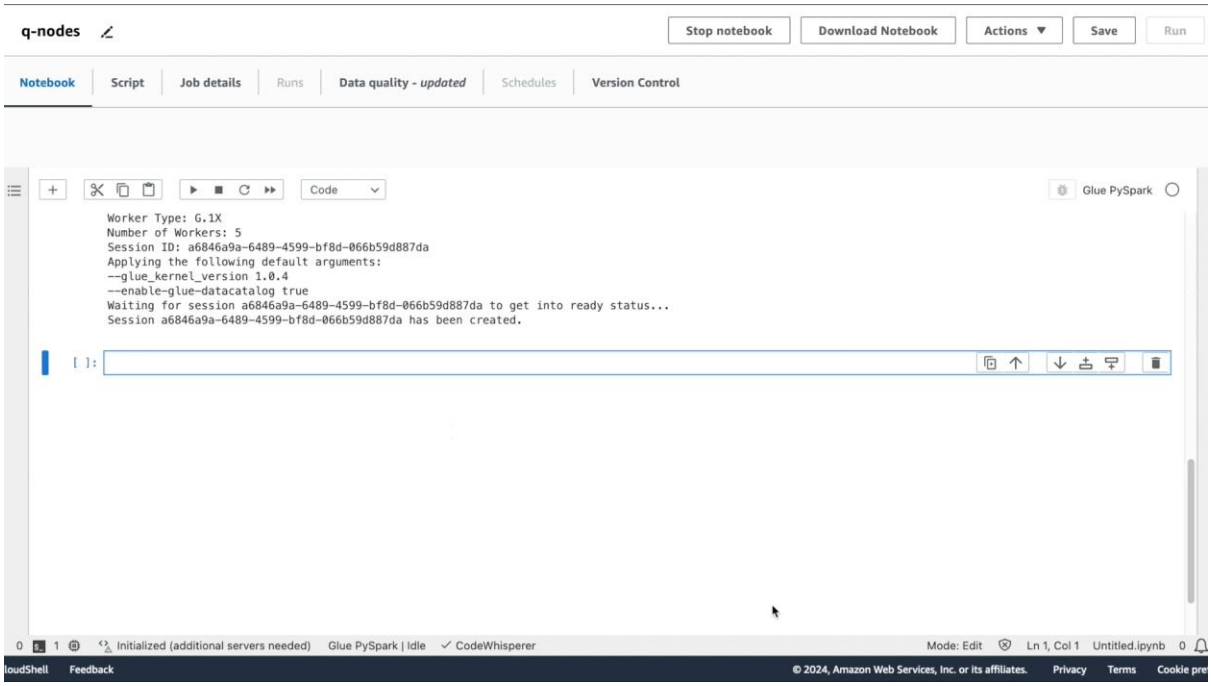
For example, you can ask Amazon Q to "*Please provide a Glue script that reads from Snowflake, renames the fields, and writes to Redshift*" and in response, Amazon Q data integration in AWS Glue will return an AWS Glue job script that can perform the requested action. You can review the generated code to ensure that it fulfills the requested intent. If satisfied, you can deploy it as an AWS Glue job in production. You can troubleshoot jobs by asking the integration to explain errors and failures, and to propose solutions. Amazon Q can answer questions about AWS Glue or data integration best practices.



About

17.2 AWS Glue Studio notebook interactions

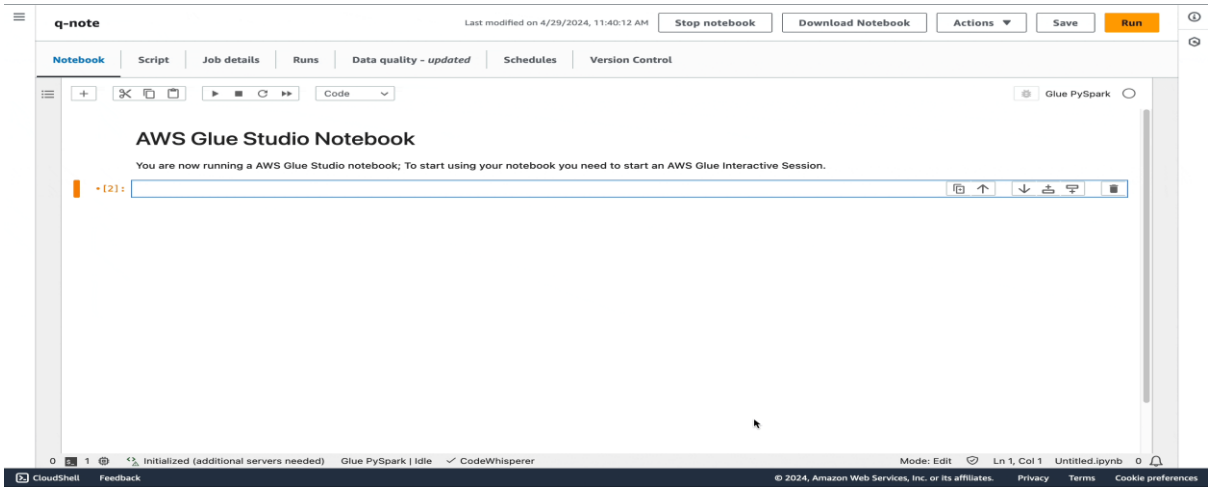
Add a new cell and enter your comment to describe what you want to achieve. After you press **Tab** and **Enter**, the recommended code is shown.First intent is to extract the data: *"Give me code that reads a Glue Data Catalog table"*, followed by *"Give me code to apply a filter transform with star_rating>3"* and *"Give me code that writes the frame into S3 as Parquet"*.



About

Complex prompts

You can generate a full script with a single complex prompt. *"I have JSON data in S3 and data in Oracle that needs combining. Please provide a Glue script that reads from both sources, does a join, and then writes results to Redshift."*



About

18. AWS Glue triggers:

There are three types of triggers:

Scheduled

A time-based trigger based on `cron`.

You can create a trigger for a set of jobs or crawlers based on a schedule. You can specify constraints, such as the frequency that the jobs or crawlers run, which days of the week they run, and at what time. These constraints are based on `cron`. When you're setting up a schedule for a trigger, consider the features and limitations of cron. For example, if you choose to run your crawler on day 31 each month, keep in mind that some months don't have 31 days. For more information about cron, see Time-based schedules for jobs and crawlers.

Conditional

A trigger that fires when a previous job or crawler or multiple jobs or crawlers satisfy a list of conditions.

When you create a conditional trigger, you specify a list of jobs and a list of crawlers to watch. For each watched job or crawler, you specify a status to watch for, such as succeeded, failed, timed out, and so on. The trigger fires if the watched jobs or crawlers end with the specified statuses. You can configure the trigger to fire when any or all of the watched events occur.

For example, you could configure a trigger T1 to start job J3 when both job J1 and job J2 successfully complete, and another trigger T2 to start job J4 if either job J1 or job J2 fails.

The following table lists the job and crawler completion states (events) that triggers watch for.

Job completion states	Crawler completion states
• `SUCCEEDED` • `STOPPED` • `FAILED` • `TIMEOUT`	• `SUCCEEDED` • `FAILED` • `CANCELLED`

On-demand

A trigger that fires when you activate it. On-demand triggers never enter the `ACTIVATED` or `DEACTIVATED` state. They always remain in the `CREATED` state.

18.1 Passing job parameters with triggers

A trigger can pass parameters to the jobs that it starts. Parameters include job arguments, timeout value, security configuration, and more. If the trigger starts multiple jobs, the parameters are passed to each job.

The following are the rules for job arguments passed by a trigger:

- If the key in the key-value pair matches a default job argument, the passed argument overrides the default argument. If the key doesn't match a default argument, then the argument is passed as an additional argument to the job.
- If the key in the key-value pair matches a non-overridable argument, the passed argument is ignored.

18.2 To add a trigger (console)

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, under **ETL**, choose **Triggers**. Then choose **Add trigger**.
3. Provide the following properties:

   Name

   Give your trigger a unique name.

   Trigger type

   Specify one of the following:

   - **Schedule:** The trigger fires at a specific frequency and time.
   - **Job events:** A conditional trigger. The trigger fires when any or all jobs in the list match their designated statuses. For the trigger to fire, the watched jobs must have been started by triggers. For any job you choose, you can only watch one job event (completion status).
   - **On-demand:** The trigger fires when it is activated.
4. Complete the trigger wizard. On the **Review** page, you can activate **Schedule** and **Job events** (conditional) triggers immediately by selecting **Enable trigger on creation**.

18.3 To add a trigger (AWS CLI)

- ```
  aws glue create-trigger --name MyTrigger --type SCHEDULED --schedule "cron(0 12 * * ? *)" --actions CrawlerName=MyCrawler --start-on-creation
  ```

  This command creates a schedule trigger named `MyTrigger`, which runs every day at 12:00pm UTC and starts a crawler named `MyCrawler`. The trigger is created in the activated state.

18.4 Cron expressions

Cron expressions have six required fields, which are separated by white space.

Syntax

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

Fields	Values	Wildcards
Minutes	0–59	, - * /
Hours	0–23	, - * /
Day-of-month	1–31	, - * ? / L W
Month	1–12 or JAN-DEC	, - * /
Day-of-week	1–7 or SUN-SAT	, - * ? / L
Year	1970–2199	, - * /

WILDCARDS

- The **,** (comma) wildcard includes additional values. In the `Month` field, `JAN,FEB,MAR` would include January, February, and March.
- The **-** (dash) wildcard specifies ranges. In the `Day` field, 1–15 would include days 1 through 15 of the specified month.
- The * (asterisk) wildcard includes all values in the field. In the `Hours` field, * would include every hour.
- The */* (forward slash) wildcard specifies increments. In the `Minutes` field, you could enter `1/10` to specify every 10th minute, starting from the first minute of the hour (for example, the 11th, 21st, and 31st minute).
```

- The **?** (question mark) wildcard specifies one or another. In the `Day-of-month` field you could enter **7**, and if you didn't care what day of the week the seventh was, you could enter **?** in the Day-of-week field.
- The **L** wildcard in the `Day-of-month` or `Day-of-week` fields specifies the last day of the month or week.
- The **W** wildcard in the `Day-of-month` field specifies a weekday. In the `Day-of-month` field, `3W` specifies the day closest to the third weekday of the month.

LIMITS

- You can't specify the `Day-of-month` and `Day-of-week` fields in the same cron expression. If you specify a value in one of the fields, you must use a **?** (question mark) in the other.
- Cron expressions that lead to rates faster than 5 minutes are not supported.

EXAMPLES

When creating a schedule, you can use the following sample cron strings.

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
0	10	*	*	?	*	Run at 10:00 am (UTC) every day
15	12	*	*	?	*	Run at 12:15 pm (UTC) every day
0	18	?	*	MON-FRI	*	Run at 6:00 pm (UTC) every

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
						Monday through Friday
0	8	1	*	?	*	Run at 8:00 am (UTC) every first day of the month
0/15	*	*	*	?	*	Run every 15 minutes
0/10	*	?	*	MON-FRI	*	Run every 10 minutes Monday through Friday
0/5	8–17	?	*	MON-FRI	*	Run every 5 minutes Monday through Friday between 8:00 am and

Minutes	Hours	Day of month	Month	Day of week	Year	Meaning
						5:55 pm (UTC)

For example to run on a schedule of every day at 12:15 UTC, specify:

```
cron(15 12 * * ? *)
```

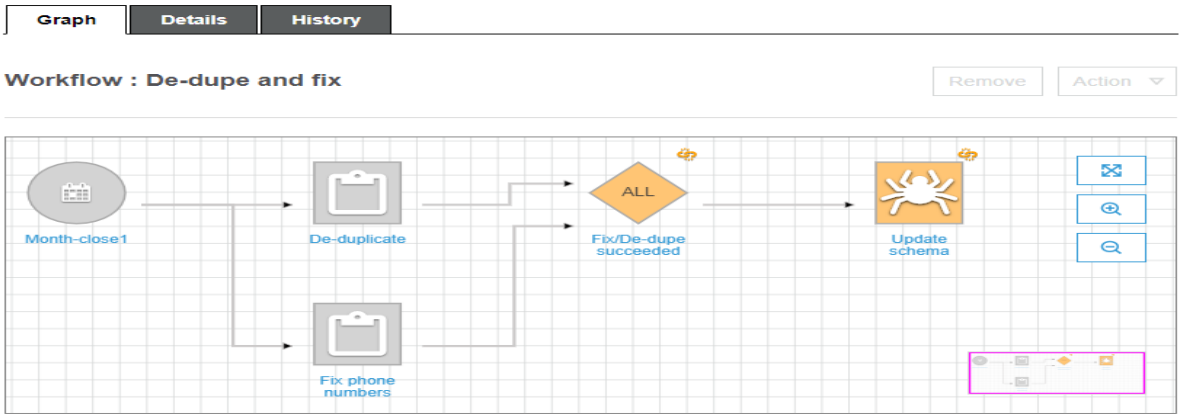
TO ACTIVATE OR DEACTIVATE A TRIGGER (CONSOLE)

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, under **ETL**, choose **Triggers**.
3. Select the check box next to the desired trigger, and on the **Action** menu choose **Enable trigger** to activate the trigger or **Disable trigger** to deactivate the trigger.

---

19. Creating and building out a workflow manually in AWS Glue:

In AWS Glue, you can use workflows to create and visualize complex extract, transform, and load (ETL) activities involving multiple crawlers, jobs, and triggers. Each workflow manages the execution and monitoring of all its jobs and crawlers. As a workflow runs each component, it records execution progress and status. This provides you with an overview of the larger task and the details of each step. The AWS Glue console provides a visual representation of a workflow as a graph.



Steps for creating Workflow:

Step 1: Create the workflow

1. Sign in to the AWS Management Console and open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, under **ETL**, choose **Workflows**.
3. Choose **Add workflow** and complete the **Add a new ETL workflow** form.

   Any optional default run properties that you add are made available as arguments to all jobs in the workflow. For more information, see Getting and setting workflow run properties in AWS Glue.
4. Choose **Add workflow**.

   The new workflow appears in the list on the **Workflows** page.

Step 2: Add a start trigger

1. On the **Workflows** page, select your new workflow. Then, at the bottom of the page, ensure that the **Graph** tab is selected.
2. Choose **Add trigger**, and in the **Add trigger** dialog box, do one of the following:
   - Choose **Clone existing**, and choose a trigger to clone. Then choose **Add**.

     The trigger appears on the graph, along with the jobs and crawlers that it watches and the jobs and crawlers that it starts.

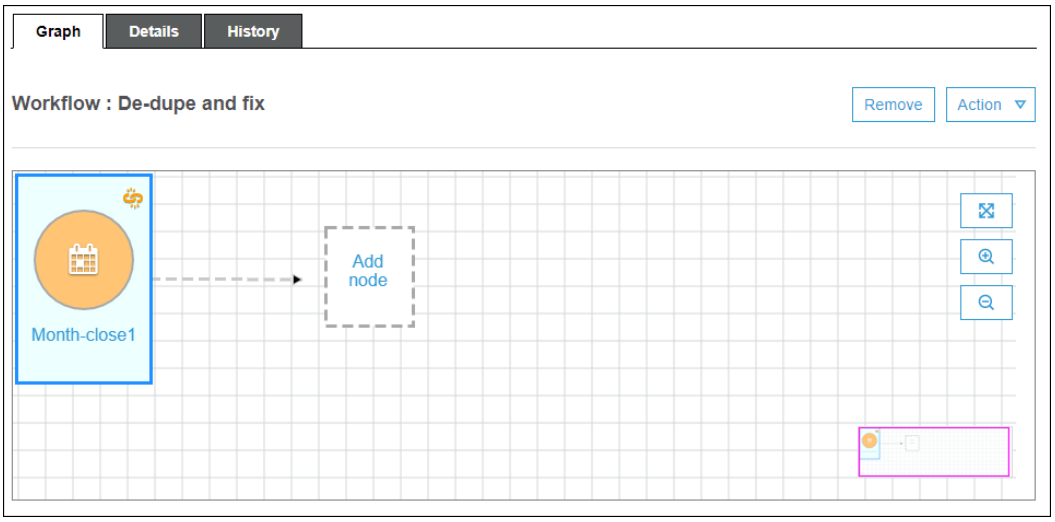
     If you mistakenly selected the wrong trigger, select the trigger on the graph, and then choose **Remove**.
   - Choose **Add new**, and complete the **Add trigger** form.
     a. For **Trigger type**, select **Schedule**, **On demand**, or **EventBridge event**.

For trigger type **Schedule**, choose one of the **Frequency** options. Choose **Custom** to enter a `cron` expression.

b. Choose **Add**.

The trigger appears on the graph, along with a placeholder node (labeled **Add node**). In the example below, the start trigger is a schedule trigger named `Month-close1`.

At this point, the trigger isn't saved yet.



3. If you added a new trigger, complete these steps:

a. Do one of the following:

- Choose the placeholder node (**Add node**).
- Ensure that the start trigger is selected, and on the **Action** menu above the graph, choose **Add jobs/crawlers to trigger**.

b. In the **Add jobs(s) and crawler(s) to trigger** dialog box, select one or more jobs or crawlers, and then choose **Add**.

The trigger is saved, and the selected jobs or crawlers appear on the graph with connectors from the trigger.

If you mistakenly added the wrong jobs or crawlers, you can select either the trigger or a connector and choose **Remove**.

Step 3: Add more triggers

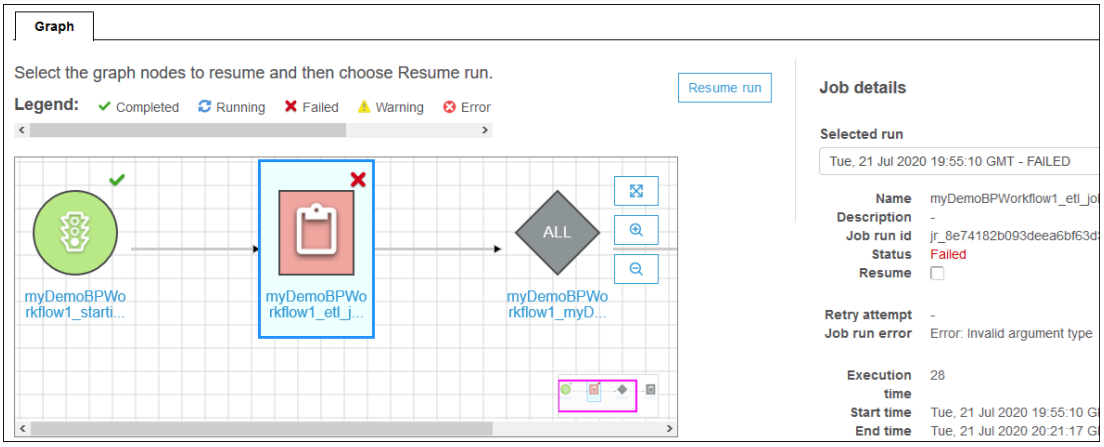
Step4:Run the workflow

TO RUN AND MONITOR A WORKFLOW (CONSOLE)

1. Open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. In the navigation pane, under **ETL**, choose **Workflows**.
3. Select a workflow. On the **Actions** menu, choose **Run**.

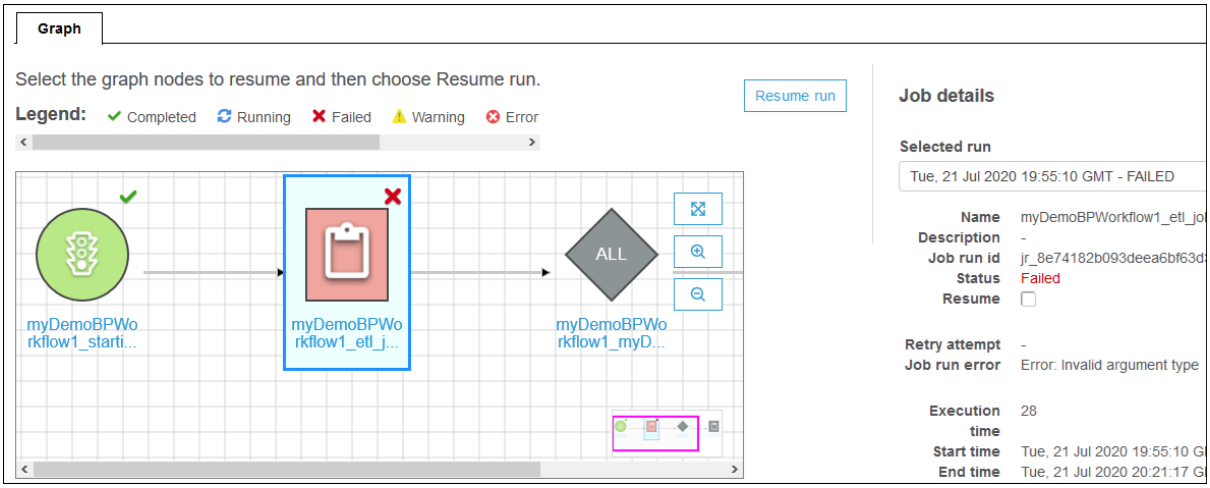
4. Check the **Last run status** column in the workflows list. Choose the refresh button to view ongoing workflow status.
5. While the workflow is running or after it has completed (or failed), view the run details by completing the following steps.
    a. Ensure that the workflow is selected, and choose the **History** tab.
    b. Choose the current or most recent workflow run, and then choose **View run details**.

       The workflow runtime graph shows the current run status.
    c. Choose any node in the graph to view details and status of the node.



To resume a workflow run (console):

1. In the navigation pane, choose **Workflows**.
2. Select a workflow, and then choose the **History** tab.
3. Select the workflow run that only partially ran, and then choose **View run details**.
4. In the run graph, select the first (or only) node that you want to restart and that you want to resume the workflow run from.
5. In the details pane to the right of the graph, select the **Resume** check box.



Blueprint restrictions

Keep the following blueprint restrictions in mind:

- The blueprint must be registered in the same AWS Region where the Amazon S3 bucket resides in.
- To share blueprints across AWS accounts you must give the read permissions on the blueprint ZIP archive in Amazon S3. Customers who have read permission on a blueprint ZIP archive can register the blueprint in their AWS account and use it.
- The set of blueprint parameters is stored as a single JSON object. The maximum length of this object is 128 KB.
- The maximum uncompressed size of the blueprint ZIP archive is 5 MB. The maximum compressed size is 1 MB.
- Limit the total number of jobs, crawlers, and triggers within a workflow to 100 or less. If you include more than 100, you might get errors when trying to resume or stop workflow runs.

20. Developing blueprints in AWS Glue:

A blueprint consists of a project that contains a blueprint parameter configuration file and a script that defines the *layout* of the workflow to generate. The layout defines the jobs and crawlers (or *entities* in blueprint script terminology) to create.

You do not directly specify any triggers in the layout script. Instead you write code to specify the dependencies between the jobs and crawlers that the script creates. AWS Glue generates the triggers based on your dependency specifications. The output of the layout script is a workflow object, which contains specifications for all workflow entities.

You build your workflow object using the following AWS Glue blueprint libraries:

- `awsglue.blueprint.base_resource` – A library of base resources used by the libraries.
- `awsglue.blueprint.workflow` – A library for defining a `Workflow` class.
- `awsglue.blueprint.job` – A library for defining a `Job` class.
- `awsglue.blueprint.crawler` – A library for defining a `Crawler` class.

The only other libraries that are supported for layout generation are those libraries that are available for the Python shell.

When you're ready to make the blueprint available to data analysts, you package the script, the parameter configuration file, and any supporting files, such as additional scripts and libraries, into a single deployable asset. You then upload the asset to Amazon S3 and ask an administrator to register it with AWS Glue.

20.1 Writing the blueprint code:

Each blueprint project that you create must contain at a minimum the following files:

- A Python layout script that defines the workflow. The script contains a function that defines the entities (jobs and crawlers) in a workflow, and the dependencies between them.
- A configuration file, `blueprint.cfg`, which defines:
    - The full path of the workflow layout definition function.
    - The parameters that the blueprint accepts.

20.1.1 Creating the blueprint layout script:

Here is a sample layout generator script in a file named `Layout.py`:

```
import argparse

import sys

import os

import json
```

```
from awsglue.blueprint.workflow import *

from awsglue.blueprint.job import *

from awsglue.blueprint.crawler import *

def generate_layout(user_params, system_params):

    etl_job =
Job(Name="{}_etl_job".format(user_params['WorkflowName']),

                  Command={

                      "Name": "glueetl",

                      "ScriptLocation":
user_params['ScriptLocation'],

                      "PythonVersion": "2"

                  },

                  Role=user_params['PassRole'])

    post_process_job =
Job(Name="{}_post_process".format(user_params['WorkflowName']),

                             Command={

                                 "Name": "pythonshell",

                                 "ScriptLocation":
user_params['ScriptLocation'],

                                 "PythonVersion": "2"

                             },

                             Role=user_params['PassRole'],

                             DependsOn={

                                 etl_job: "SUCCEEDED"

                             },
```

```
                              WaitForDependencies="AND")

    sample_workflow = Workflow(Name=user_params['WorkflowName'],

                              Entities=Entities(Jobs=[etl_job,
post_process_job]))

    return sample_workflow
```

`DependsOn` is a dictionary that specifies **which jobs must complete** before the current job can start, and **under what condition** (e.g., success or failure).

DependsOn={

etl_job: "SUCCEEDED"

}

This means:

- The `post_process_job` will **only start** after the `etl_job` has **finished successfully**.
- If `etl_job` fails or is skipped, `post_process_job` will **not run**.

While `"SUCCEEDED"` is the most common, AWS Glue also supports:

- `"FAILED"` – Run the job only if the dependency fails.
- `"STOPPED"` – Run if the dependency was stopped.
- `"TIMEOUT"` – Run if the dependency timed out.

`WaitForDependencies` is a parameter used in AWS Glue **Blueprint Job definitions** to control **how multiple dependencies are evaluated** before a job is triggered.

Job(

Name="job_B",

DependsOn={

job_A: "SUCCEEDED",

job_C: "SUCCEEDED"

},

WaitForDependencies="AND"

)

- **`AND"`** (default):

- The job will run **only if all** specified dependencies meet their condition.
- In the example above, `job_B` will run **only if both `job_A` and `job_C` succeed**.

- **`"OR"`**:

  - The job will run **if any one** of the dependencies meets its condition.
  - So if either `job_A` or `job_C` succeeds, `job_B` will be triggered.

20.1.2 Creating the configuration file:

```
"layoutGenerator": "DemoBlueprintProject.Layout.generate_layout",
    "parameterSpec" : {
            "WorkflowName" : {
                 "type": "String",
                 "collection": false
            },
            "WorkerType" : {
                 "type": "String",
                 "collection": false,
                 "allowedValues": ["G1.X", "G2.X"],
                 "defaultValue": "G1.X"
            },
            "Dpu" : {
                 "type" : "Integer",
                 "allowedValues" : [2, 4, 6],
                 "defaultValue" : 2
            },
            "DynamoDBTableName": {
                 "type": "String",
                 "collection" : false
            },
            "ScriptLocation" : {
                 "type": "String",
                 "collection": false
     }
    }
```

Purpose of this JSON

It tells AWS Glue:

- What parameters the blueprint expects.
- What types of values are allowed.
- What default values to use.
- How to validate user input when the blueprint is deployed.

1. `layoutGenerator`

JSON

"layoutGenerator": "DemoBlueprintProject.Layout.generate_layout"

Show more lines

- Specifies the **Python function** that generates the workflow layout.
- This function (`generate_layout`) is defined in the module `DemoBlueprintProject.Layout`.

2. `parameterSpec`

Defines the **input parameters** required to run the blueprint.

Each parameter includes:

- `type`: Data type (e.g., String, Integer).
- `collection`: Whether it accepts multiple values (`true`) or just one (`false`).
- `allowedValues`: Optional list of valid values.
- `defaultValue`: Optional default value if the user doesn't provide one.

Parameter data type	Notes
`String`	-
`Integer`	-
`Double`	-
`Boolean`	Possible values are `true` and `false`. Generates a check box on the **Create a workflow from <blueprint>** page on the AWS Glue console.
`S3Uri`	Complete Amazon S3 path, beginning with `s3://`. Generates a text field and **Browse** button on the **Create a workflow from <blueprint>** page.
`S3Bucket`	Amazon S3 bucket name only. Generates a bucket picker on the **Create a workflow from <blueprint>** page.
`IAMRoleArn`	Amazon Resource Name (ARN) of an AWS Identity and Access Management (IAM) role. Generates a role picker on the **Create a workflow from <blueprint>** page.
`IAMRoleName`	Name of an IAM role. Generates a role picker on the **Create a workflow from <blueprint>** page.

TO PUBLISH A BLUEPRINT

1. Create the necessary scripts, resources, and blueprint configuration file.

2. Add all files to a ZIP archive and upload the ZIP file to Amazon S3. Use an S3 bucket that is in the same Region as the Region in which users will register and run the blueprint.

   You can create a ZIP file from the command line using the following command.

```
zip -r folder.zip folder
```

3. Add a bucket policy that grants read permissions to the AWS desired account. The following is a sample policy.
   - JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::111122223333:root"
         },
         "Action": "s3:GetObject",
         "Resource": "arn:aws:s3:::my-blueprints/*"
      }
   ]
}
```

   4.Grant the IAM `s3:GetObject` permission on the Amazon S3 bucket to the AWS Glue administrator or to whoever will be registering blueprints. For a sample policy to grant to administrators, see AWS Glue administrator permissions for blueprints.

Examples Folders:

aws-glue-blueprint-libs/samples at master · awslabs/aws-glue-blueprint-libs · GitHub

The titles of the sample projects are:

- Compaction: this blueprint creates a job that compacts input files into larger chunks based on desired file size.
- Conversion: this blueprint converts input files in various standard file formats into Apache Parquet format, which is optimized for analytic workloads.
- Crawling Amazon S3 locations: this blueprint crawls multiple Amazon S3 locations to add metadata tables to the Data Catalog.
- Custom connection to Data Catalog: this blueprint accesses data stores using AWS Glue custom connectors, reads the records, and populates the table definitions in the AWS Glue Data Catalog based on the record schema.
- Encoding: this blueprint converts your non-UTF files into UTF encoded files.
- Partitioning: this blueprint creates a partitioning job that places output files into partitions based on specific partition keys.
- Importing Amazon S3 data into a DynamoDB table: this blueprint imports data from Amazon S3 into a DynamoDB table.
- Standard table to governed: this blueprint imports an AWS Glue Data Catalog table into a Lake Formation table.

TO REGISTER A BLUEPRINT (CONSOLE)

1. Ensure that you have read permissions (`s3:GetObject`) on the blueprint ZIP archive in Amazon S3.
2. Open the AWS Glue console at https://console.aws.amazon.com/glue/.

   Sign in as a user that has permissions to register a blueprint. Switch to the same AWS Region as the Amazon S3 bucket that contains the blueprint ZIP archive.
3. In the navigation pane, choose **blueprints**. Then on the **blueprints** page, choose **Add blueprint**.
4. Enter a blueprint name and optional description.
5. For **ZIP archive location (S3)**, enter the Amazon S3 path of the uploaded blueprint ZIP archive. Include the archive file name in the path and begin the path with `s3://`.
6. (Optional) Add tag one or more tags.
7. Choose **Add blueprint**.

   The **blueprints** page returns and shows that the blueprint status is `CREATING`. Choose the refresh button until the status changes to `ACTIVE` or `FAILED`.
8. If the status is `FAILED`, select the blueprint, and on the **Actions** menu, choose **View**.

The detail page shows the reason for the failure. If the error message is `"Unable to access object at location..."` or `"Access denied on object at location..."`, review the following requirements:

- The user that you are signed in as must have read permission on the blueprint ZIP archive in Amazon S3.
- The Amazon S3 bucket that contains the ZIP archive must have a bucket policy that grants read permission on the object to your AWS account ID. For more information, see Developing blueprints in AWS Glue.
- The Amazon S3 bucket that you're using must be in the same Region as the Region that you're signed into on the console.

9. Ensure that data analysts have permissions on the blueprint.

   The suggested IAM policy for data analysts is shown in Data analyst permissions for blueprints. This policy grants `glue:GetBlueprint` on any resource. If your policy is more fine-grained at the resource level, then grant data analysts permissions on this newly created resource.

21. Creating a workflow from a blueprint in AWS Glue:

AWS Glue creates a workflow from a blueprint by *running* the blueprint. The blueprint run saves the parameter values that you supplied, and is used to track the progress and outcome of the creation of the workflow and its components. When troubleshooting a workflow, you can view the blueprint run to determine the blueprint parameter values that were used to create a workflow.

To create and view workflows, you require certain IAM permissions. For a suggested IAM policy, see Data analyst permissions for blueprints.

You can create a workflow from a blueprint by using the AWS Glue console, AWS Glue API, or AWS Command Line Interface (AWS CLI).

TO CREATE A WORKFLOW FROM A BLUEPRINT (CONSOLE)

1. Open the AWS Glue console at https://console.aws.amazon.com/glue/.

   Sign in as a user that has permissions to create a workflow.
2. In the navigation pane, choose **blueprints**.
3. Select a blueprint, and on the **Actions** menu, choose **Create workflow**.
4. On the **Create a workflow from <blueprint-name>** page, enter the following information:

   Blueprint parameters

These vary depending on the blueprint design. For questions about the parameters, see the developer. blueprints typically include a parameter for the workflow name.

IAM role

The role that AWS Glue assumes to create the workflow and its components. The role must have permissions to create and delete workflows, jobs, crawlers, and triggers. For a suggested policy for the role, see Permissions for blueprint roles.

5. Choose **Submit**.

   The **Blueprint Details** page appears, showing a list of blueprint runs at the bottom.

6. In the blueprint runs list, check the topmost blueprint run for workflow creation status.

   The initial status is `RUNNING`. Choose the refresh button until the status goes to `SUCCEEDED` or `FAILED`.

7. Do one of the following:
   - If the completion status is `SUCCEEDED`, you can go to the **Workflows** page, select the newly created workflow, and run it. Before running the workflow, you can review the design graph.
   - If the completion status is `FAILED`, select the blueprint run, and on the **Actions** menu, choose **View** to see the error message.

---

22. Writing an AWS Glue for Spark script:

Step 1. Create a job and paste your script

TO CREATE A JOB

1. In the AWS Management Console, navigate to the AWS Glue landing page.
2. In the side navigation pane, choose **Jobs**.
3. Choose **Spark script editor** in **Create job**, and then choose **Create**.
4. **Optional** - Paste the full text of your script into the **Script** pane. Alternatively, you can follow along with the tutorial.

Step 2. Import AWS Glue libraries

In this step, you perform the following actions.

- Import and initialize a `GlueContext` object. This is the most important import, from the script writing perspective. This exposes standard methods for

defining source and target datasets, which is the starting point for any ETL script. To learn more about the `GlueContext` class, see GlueContext class.

- Initialize a `SparkContext` and `SparkSession`. These allow you to configure the Spark engine available inside the AWS Glue job. You won't need to use them directly within introductory AWS Glue scripts.
- Call `getResolvedOptions` to prepare your job arguments for use within the script. For more information about resolving job parameters, see Accessing parameters using getResolvedOptions.
- Initialize a `Job`. The `Job` object sets configuration and tracks the state of various optional AWS Glue features. Your script can run without a `Job` object, but the best practice is to initialize it so that you don't encounter confusion if those features are later integrated.

  One of these features is job bookmarks, which you can optionally configure in this tutorial. You can learn about job bookmarks in the following section, Optional - Enable job bookmarks.

```
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

args = getResolvedOptions(sys.argv, ["JOB_NAME"])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args["JOB_NAME"], args)
```

Step 3. Extract data from a source

In this step, you provide the `create_dynamic_frame.from_catalog` method a `database` and `table_name` to extract data from a source configured in the AWS Glue Data Catalog.

In the previous step, you initialized a `GlueContext` object. You use this object to find methods that are used to configure sources, such as `create_dynamic_frame.from_catalog`.

```
S3bucket_node1 = glueContext.create_dynamic_frame.from_catalog(
    database="yyz-tickets", table_name="tickets",
transformation_ctx="S3bucket_node1"
    )
```

Step 4. Transform data with AWS Glue

After extracting source data in an ETL process, you need to describe how you want to change your data. You provide this information by creating a **Transform** node in the AWS Glue Studio visual editor.

In this step, you provide the `ApplyMapping` method with a map of current and desired field names and types to transform your `DynamicFrame`.

You perform the following transformations.

- Drop the four `location` and `province` keys.
- Change the name of `officer` to `officer_name`.
- Change the type of `ticket_number` and `set_fine_amount` to `float`.

`create_dynamic_frame.from_catalog` provides you with a `DynamicFrame` object. A `DynamicFrame` represents a dataset in AWS Glue. AWS Glue transforms are operations that change `DynamicFrames`.

```
ApplyMapping_node2 = ApplyMapping.apply(
    frame=S3bucket_node1,
    mappings=[
        ("tag_number_masked", "string", "tag_number_masked",
"string"),
        ("date_of_infraction", "string", "date_of_infraction",
"string"),
        ("ticket_date", "string", "ticket_date", "string"),
        ("ticket_number", "decimal", "ticket_number", "float"),
        ("officer", "decimal", "officer_name", "decimal"),
        ("infraction_code", "decimal", "infraction_code",
"decimal"),
        ("infraction_description", "string",
"infraction_description", "string"),
        ("set_fine_amount", "decimal", "set_fine_amount", "float"),
        ("time_of_infraction", "decimal", "time_of_infraction",
"decimal"),
    ],
    transformation_ctx="ApplyMapping_node2",
)
```

Step 5. Load data into a target

After you transform your data, you typically store the transformed data in a different place from the source. You perform this operation by creating a **target** node in the AWS Glue Studio visual editor.

In this step, you provide the `write_dynamic_frame.from_options` method a `connection_type`, `connection_options`, `format`, and `format_options` to load data into a target bucket in Amazon S3.

In Step 1, you initialized a `GlueContext` object. In AWS Glue, this is where you will find methods that are used to configure targets, much like sources.

In this procedure, you write the following code using `write_dynamic_frame.from_options`. This code is a portion of the generated sample script.

```
S3bucket_node3 = glueContext.write_dynamic_frame.from_options(

    frame=ApplyMapping_node2,

    connection_type="s3",

    format="glueparquet",

    connection_options={"path": "s3://amzn-s3-demo-bucket",
"partitionKeys": []},

    format_options={"compression": "gzip"},

    transformation_ctx="S3bucket_node3",

    )
```

Step 6. Commit the `Job` object

You initialized a `Job` object in Step 1. You may need to manually conclude its lifecycle at the end of your script if certain optional features need this to function properly, such as when using Job Bookmarks. This work is done behind the scenes in AWS Glue Studio.

In this step, call the `commit` method on the `Job` object.

In this procedure, you write the following code. This code is a portion of the generated sample script.

```
job.commit()
```

TO COMMIT THE JOB OBJECT

1. If you have not yet done this, perform the optional steps outlined in previous sections to include `transformation_ctx`.
2. Call `commit`.

Step 7. Run your code as a job

In this step, you run your job to verify that you successfully completed this tutorial. This is done with the click of a button, as in the AWS Glue Studio visual editor.

TO RUN YOUR CODE AS A JOB

1. Choose **Untitled job** on the title bar to edit and set your job name.
2. Navigate to the **Job details** tab. Assign your job an **IAM Role**. You can use the one created by the AWS CloudFormation template in the prerequisites for the AWS Glue Studio tutorial. If you have completed that tutorial, it should be available as `AWS Glue StudioRole`.
3. Choose **Save** to save your script.
4. Choose **Run** to run your job.
5. Navigate to the **Runs** tab to verify that your job completes.
6. Navigate to *`amzn-s3-demo-bucket`*, the target for `write_dynamic_frame.from_options`. Confirm that the output matches your expectations.

---

23. ETL scripts in PySpark:

Setting up to use Python with AWS Glue:

TO SET UP YOUR SYSTEM FOR USING PYTHON WITH AWS GLUE

Follow these steps to install Python and to be able to invoke the AWS Glue APIs.

1. If you don't already have Python installed, download and install it from the Python.org download page.
2. Install the AWS Command Line Interface (AWS CLI) as documented in the AWS CLI documentation.

   The AWS CLI is not directly necessary for using Python. However, installing and configuring it is a convenient way to set up AWS with your account credentials and verify that they work.
3. Install the AWS SDK for Python (Boto 3), as documented in the Boto3 Quickstart .

   Boto 3 resource APIs are not yet available for AWS Glue. Currently, only the Boto 3 client APIs can be used.

For more information about Boto 3, see AWS SDK for Python (Boto3) Getting Started.

23.1 PySpark extensions:

Accessing parameters using `getResolvedOptions`:

The AWS Glue `getResolvedOptions(args, options)` utility function gives you access to the arguments that are passed to your script when you run a job. To use this function, start by importing it from the AWS Glue `utils` module, along with the `sys` module:

```
import sys

from awsglue.utils import getResolvedOptions
```

GETRESOLVEDOPTIONS(ARGS, OPTIONS)

- `args` – The list of arguments contained in `sys.argv`.
- `options` – A Python array of the argument names that you want to retrieve.

1. Using AWS Glue Console

✅ Steps:

1. **Go to AWS Glue Console** → https://console.aws.amazon.com/glue
2. Click on **Jobs** → Select or create a new job.
3. In the job configuration:
    - Scroll to **"Job parameters (optional)"**
    - Add parameters like:
    - `--input_path  s3://your-bucket/input`
    - `--output_path  s3://your-bucket/output`
      `--JOB_NAME  myGlueJob`

Example:

```
from awsglue.utils import getResolvedOptions
import sys

args = getResolvedOptions(sys.argv, ['JOB_NAME', 'input_path', 'output_path'])

print("Job Name:", args['JOB_NAME'])
print("Input Path:", args['input_path'])
print("Output Path:", args['output_path'])
```

2. Using AWS CLI

✅ Command to Run a Glue Job:

```
aws glue start-job-run \
  --job-name myGlueJob \
  --arguments '{"--input_path":"s3://your-bucket/input","--output_path":"s3://your-bucket/output"}'
```

- AWS passes these arguments to the script.
- `getResolvedOptions` extracts them from `sys.argv`.

23.1.1 DynamicFrame class:

Apache Spark DataFrame

- **Similar to R and Pandas DataFrames**: Tabular structure with rows and columns.
- **Supports functional-style operations**: `map`, `reduce`, `filter`, etc.
- **Supports SQL operations**: `select`, `project`, `aggregate`, etc.
- **Schema requirement**: Needs a predefined schema before loading data.
- **Schema inference**: SparkSQL makes two passes—first to infer schema, second to load data.
- **Limitations with messy data**:
  - Inconsistent field types across records can cause issues.
  - Spark may default to treating fields as strings when types conflict.
  - Limited control over schema resolution.
  - Extra pass over large datasets can be expensive.

AWS Glue DynamicFrame

- **Schema not required initially**: Each record is self-describing.
- **Schema computed on-the-fly**: Inferred only when needed.
- **Handles schema inconsistencies**: Uses **choice (union) types** to represent multiple possible types.
- **Better suited for messy or semi-structured data**.
- **Can be resolved for compatibility**: Schema inconsistencies can be explicitly resolved for fixed-schema data stores.
- **DynamicRecord**:
  - Represents a single logical record in a DynamicFrame.
  - Self-describing like a row in Spark DataFrame.
  - Typically not manipulated individually in PySpark; transformations are applied to the whole DynamicFrame.
- **Interoperability**: Can convert between DynamicFrame and DataFrame after resolving schema issues.

1. `fromDF`

`FROMDF(DATAFRAME, GLUE_CTX, NAME)`

Parameters

- **`dataframe`**: The Spark DataFrame you want to convert.

- **`glue_ctx`**: The `GlueContext` object, which provides the context for AWS Glue operations.
- **`name`** *(optional)*: A name for the resulting DynamicFrame. Optional in AWS Glue 3.0 and later.

🧠 What It Does

- Converts each row in the Spark DataFrame into a **DynamicRecord**.
- Each DynamicRecord is **self-describing**, meaning it carries its own schema.
- Useful when working with **inconsistent or evolving schemas**.
- Ensures compatibility with AWS Glue transformations and connectors.

Duplicate column names in the DataFrame must be resolved **before** using `fromDF()`.

```
data = [("Alice", 25), ("Bob", 30)]
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)
# Convert to DynamicFrame
dyf = DynamicFrame.fromDF(df, glue_ctx, "people")
# Show schema and data
dyf.printSchema()
dyf.show()
```

2. `toDF`

TODF(OPTIONS)

The `toDF()` function in **AWS Glue** is used to convert a **DynamicFrame** back into a **Spark DataFrame**. This is useful when you want to use Spark-native operations after performing schema resolution or transformations using AWS Glue.

`options`: A list of `ResolveOption` objects.

- These are used to **resolve choice types** (i.e., fields with multiple possible types).
- Not used for format options like CSV parsing.

Converts each **DynamicRecord** in the DynamicFrame into a row in a Spark DataFrame. Ensures that schema inconsistencies (e.g., fields with multiple types) are resolved using the provided `ResolveOption` rules.

1. **Do not use `toDF()` for format options** like CSV parsing.
2. Use `from_options()` when creating the DynamicFrame for that.
3. Use `ResolveOption` to handle fields with multiple types (e.g., string or int).

```
# Resolve choice types before converting
resolved_dyf = dyf.resolveChoice(specs=[("age", "cast:int")])
```

```
# Correct: Specify format options in from_options
csv_dyf = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
```

```
    connection_options={"paths": ["s3://my-bucket/path/to/csv/"]},
    format="csv",
    format_options={
        "withHeader": True,
        "separator": ",",
        "inferSchema": True
    }
)
# Convert to DataFrame (no format options needed here)
csv_df = csv_dyf.toDF()
```

3. `count()`

Returns the number of records (rows) in a DataFrame or DynamicFrame.

df.count() # Spark DataFrame
dyf.count() # AWS Glue DynamicFrame

4. `printSchema()`

Prints the schema (structure) of a DynamicFrame in a readable format.

dyf.printSchema()

For Spark DataFrames, use `df.printSchema()`.

5. `show`

`show(num_rows)` – Prints a specified number of rows from the underlying `DataFrame`.

6. `repartition(n)`

Redistributes the data into **`n` partitions**. Useful for parallel processing or writing large datasets.

df_repartitioned = df.repartition(4)

the data is now split into 4 partitions.

7. `coalesce(n)`

Reduces the number of partitions to **`n`**, combining existing ones. More efficient than `repartition()` when decreasing partitions.

df_coalesced = df.coalesce(1).the data is now in 1 partition

8. `apply_mapping`

`APPLY_MAPPING(MAPPINGS, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)`

Applies a declarative mapping to a `DynamicFrame` and returns a new `DynamicFrame` with those mappings applied to the fields that you specify. Unspecified fields are omitted from the new `DynamicFrame`.

`mappings` *(Required)*

- A **list of tuples** that define how each field should be transformed.
- Each tuple follows this format:
- ("source_field_name", "source_type", "target_field_name", "target_type")

`transformation_ctx` *(Optional)*

- A **string identifier** for the transformation.
- Useful for debugging and tracking in AWS Glue job logs.

`info` *(Optional)*

- A string for **additional metadata or notes** about the transformation.

`stageThreshold` *(Optional)*

- Integer value to **limit the number of errors** allowed in a single stage.
- If exceeded, the job may fail or log warnings.

`totalThreshold` *(Optional)*

- Integer value to **limit the total number of errors** allowed across all stages.

```
persons = glueContext.create_dynamic_frame.from_catalog(
    database="legislators", table_name="persons_json"

)

persons_mapped = persons.apply_mapping(

    [
        ("family_name", "String", "last_name", "String"),
        ("name", "String", "first_name", "String"),
        ("birth_date", "String", "date_of_birth", "Date"),
    ]
)

persons_mapped.printSchema()
```

9. drop_fields

DROP_FIELDS(PATHS, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

- `paths` – A list of strings. Each contains the full path to a field node that you want to drop. You can use dot notation to specify nested fields. For example, if field `first` is a child of field `name` in the tree, you specify `"name.first"` for the path.

  If a field node has a literal `.` in the name, you must enclose the name in backticks (`` ` ``).

```
friends = friends.drop_fields(paths=["age", "location.county",
"friends.age"])
```

10. filter

FILTER(F, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Returns a new `DynamicFrame` that contains all `DynamicRecords` within the input `DynamicFrame` that satisfy the specified predicate function `f`.

- `f` – The predicate function to apply to the `DynamicFrame`. The function must take a `DynamicRecord` as an argument and return True if the `DynamicRecord` meets the filter requirements, or False if not (required).

Example:

```
sac_or_mon = medicare.filter(
    f=lambda x: x["Provider State"] in ["CA", "AL"]
    and x["Provider City"] in ["SACRAMENTO", "MONTGOMERY"]
)
```

11. join

JOIN(PATHS1, PATHS2, FRAME2, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Performs an equality join with another `DynamicFrame` and returns the resulting `DynamicFrame`.

- `paths1` – A list of the keys in this frame to join.
- `paths2` – A list of the keys in the other frame to join.
- `frame2` – The other `DynamicFrame` to join.

```
persons_memberships = persons.join(
    paths1=["id"], paths2=["person_id"], frame2=memberships
)
```

12. `map`

MAP(F, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Returns a new `DynamicFrame` that results from applying the specified mapping function to all records in the original `DynamicFrame`.

- `f` – The mapping function to apply to all records in the `DynamicFrame`. The function must take a `DynamicRecord` as an argument and return a new `DynamicRecord` (required).

```
def add_is_adult(record):
    age = record["age"]
    record["is_adult"] = age >= 18 if age is not None else False
    return record
# Apply map
mapped_dyf = dyf.map(
    f=add_is_adult,
    transformation_ctx="add_is_adult_flag"
)
```

13. `mergeDynamicFrame`

MERGEDYNAMICFRAME(STAGE_DYNAMIC_FRAME, PRIMARY_KEYS, TRANSFORMATION_CTX = "", OPTIONS = {}, INFO = "", STAGETHRESHOLD = 0, TOTALTHRESHOLD = 0)

- `stage_dynamic_frame` – The staging `DynamicFrame` to merge.
- `primary_keys` – The list of primary key fields to match records from the source and staging dynamic frames.

This function compares both DynamicFrames using the primary key.
If a matching key is found:

- The record from `stage_dynamic_frame` **replaces** the one in `mergeDynamicFrame`.

- If no match is found:
  - The new record is **inserted**.

```
merged_dyf = base_dyf.mergeDynamicFrame(
    stage_dynamic_frame=new_data_dyf,
    primary_keys=["customer_id"])
```

14. `relationalize`

RELATIONALIZE(ROOT_TABLE_NAME, STAGING_PATH, OPTIONS, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

The **`relationalize()`** function in **AWS Glue** is used to **flatten nested data structures** (like JSON) in a **DynamicFrame** into **multiple related tables**. This is especially useful when dealing

with semi-structured data such as deeply nested JSON files, where you want to convert the data into a relational format for easier querying or storage in databases.

- `root_table_name` – The name for the root table.
- `staging_path` – The path where the method can store partitions of pivoted tables in CSV format (optional). Pivoted tables are read back from this path.
- `options` *(Optional)*
  A dictionary of options to control relationalization behavior. Common options include:
  `"topLevelDir"`: Directory name for output
  `"includePath"`: Whether to include path info in the output

```
{
 "order_id": 101,
 "customer": {
  "name": "Alice",
  "address": {
   "city": "Chennai",
   "zip": "600001"
  }
 },
 "items": [
  {"product": "Book", "qty": 2},
  {"product": "Pen", "qty": 5}
 ]
}
```

The result is a **dictionary of DynamicFrames**, each representing a flattened table:

1. orders (root table)

order_id	customer	items
101	orders_customer_0	orders_items_0

2. orders.customer

id	name	address
orders_customer_0	Alice	orders_address_0

3. orders.customer.address

id	city	zip
orders_address_0	Chennai	600001

4. orders.items

id	product	qty
orders_items_0	Book	2
orders_items_1	Pen	5

15. `rename_field`

RENAME_FIELD(OLDNAME, NEWNAME, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Renames a field in this `DynamicFrame` and returns a new `DynamicFrame` with the field renamed.

`oldName` – The full path to the node you want to rename.If the old name has dots in it, `RenameField` doesn't work unless you place backticks around it (`).

`newName` – The new name, as a full path.

```
orgs = orgs.rename_field("id", "org_id").rename_field("name", "org_name")
```

16. `resolveChoice`

RESOLVECHOICE(SPECS = NONE, CHOICE = "" , DATABASE = NONE , TABLE_NAME = NONE , TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0, CATALOG_ID = NONE)

Resolves a choice type within this `DynamicFrame` and returns the new `DynamicFrame`.

There are two ways to use `resolveChoice`. The first is to use the `specs` argument to specify a sequence of specific fields and how to resolve them. The other mode for `resolveChoice` is to use the `choice` argument to specify a single resolution for all `ChoiceTypes`.

Use `specs` when you **know exactly which fields** have ambiguous types and want to resolve them **individually**.

- `specs` – A list of specific ambiguities to resolve, each in the form of a tuple: `(field_path, action)`.

```
resolved_df = dynamic_frame.resolveChoice(
   specs=[
      ("price", "cast:double"),
      ("discount", "project:string"),
```

```
        ("customer.age", "cast:int")
    ],
    transformation_ctx="resolve_specific_fields"
)
```

When a field contains **mixed types** (e.g., integers, floats, booleans, or strings), `project:string` ensures **uniformity** by converting **everything to a string**.

The `choice` parameter in AWS Glue's `resolveChoice` function is used to **apply a single resolution strategy to all ambiguous fields** (i.e., fields with multiple possible data types). This is useful when you **don't know all the ambiguous fields in advance** or want a **uniform resolution** across the entire `DynamicFrame`.

Supported Actions for `choice`:

Action	Description
`cast:type`	Casts all ambiguous fields to the specified type (e.g., `cast:string`, `cast:int`)
`make_cols`	Splits each ambiguous field into multiple columns based on type
`make_struct`	Converts ambiguous fields into structs containing all possible types
`project:type`	Projects all ambiguous fields to one type (e.g., `project:string`)
`match_catalog`	Matches field types to the schema defined in the AWS Glue Data Catalog

Examples

1. choice="cast:string"

Converts all ambiguous fields to strings.

Input:

JSON

```
{ "value": 100 }, { "value": "100" }
```

Show more lines

Output:

JSON

```
{ "value": "100" }, { "value": "100" }
```

Show more lines

---

2. `choice="make_cols"`

Creates separate columns for each type.

Input:

JSON

```
{ "status": "active" }, { "status": 1 }
```

Show more lines

Output:

JSON

```
{ "status_string": "active", "status_int": null }, { "status_string": null, "status_int": 1 }
```

Show more lines

---

3. `choice="make_struct"`

Creates a struct with all possible types.

Input:

JSON

```
{ "status": "active" }, { "status": 1 }
```

Show more lines

Output:

JSON

```
{ "status": { "string": "active", "int": null } }, { "status": { "string": null, "int": 1 } }
```

Show more lines

---

4. `choice="project:string"`

Projects all ambiguous fields to string.

Input:

JSON

{ "status": "active" }, { "status": 1 }, { "status": true }

Show more lines

Output:

JSON

{ "status": "active" }, { "status": "1" }, { "status": "true" }

Show more lines

---

5. `choice="match_catalog"`

Uses schema from Glue Data Catalog to resolve types.

17. `select_fields`

SELECT_FIELDS(PATHS, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Returns a new `DynamicFrame` that contains the selected fields.

- `paths` – A list of strings. Each string is a path to a top-level node that you want to select.

```
names = persons.select_fields(paths=["family_name", "given_name"])
```

18. `simplify_ddb_json`

`simplify_ddb_json(): DynamicFrame`

Simplifies nested columns in a `DynamicFrame` that are specifically in the DynamoDB JSON structure, and returns a new simplified `DynamicFrame`. If there're multiple types or Map type in a List type, the elements in the List will not be simplified. Note that this is a specific type of transform that behaves differently from the regular `unnest` transform and requires the data to already be in the DynamoDB JSON structure.

```
dynamicFrame = glueContext.create_dynamic_frame.from_options(
    connection_type = "dynamodb",
    connection_options = {
        'dynamodb.export': 'ddb',
        'dynamodb.tableArn': '<table arn>',
        'dynamodb.s3.bucket': '<bucket name>',
        'dynamodb.s3.prefix': '<bucket prefix>',
```

```
        'dynamodb.s3.bucketOwner': '<account_id of bucket>'
    }
)

simplified = dynamicFrame.simplify_ddb_json()
```

19. `spigot`

`SPIGOT(PATH, OPTIONS={})`

Writes sample records to a specified destination to help you verify the transformations performed by your job.

- `path` – The path of the destination to write to (required).
- `options` – Key-value pairs that specify options (optional). The `"topk"` option specifies that the first `k` records should be written. The `"prob"` option specifies the probability (as a decimal) of choosing any given record. You can use it in selecting records to write.

```
# Perform the select_fields on the DynamicFrame
persons = persons.select_fields(paths=["family_name", "given_name",
"birth_date"])
# Use spigot to write a sample of the transformed data
# (the first 10 records)
spigot_output = persons.spigot(
    path="s3://DOC-EXAMPLE-BUCKET", options={"topk": 10}
)
```

20. `split_fields`

`SPLIT_FIELDS(PATHS, NAME1, NAME2, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)`

Returns a new `DynamicFrameCollection` that contains two `DynamicFrames`. The first `DynamicFrame` contains all the nodes that have been split off, and the second contains the nodes that remain.

- `paths` – A list of strings, each of which is a full path to a node that you want to split into a new `DynamicFrame`.
- `name1` – A name string for the `DynamicFrame` that is split off.
- `name2` – A name string for the `DynamicFrame` that remains after the specified nodes have been split off.

```
split_fields_collection = frame_to_split.split_fields(["id",
"index"], "left", "right")
```

```
Inspect the input DynamicFrame's schema:
```

```
root
|-- id: long
|-- index: int
|-- contact_details.val.type: string
|-- contact_details.val.value: string

Inspect the schemas of the DynamicFrames created with split_fields:
root
|-- id: long
|-- index: int

root
|-- contact_details.val.type: string
|-- contact_details.val.value: string
```

21. split_rows

SPLIT_ROWS(COMPARISON_DICT, NAME1, NAME2, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0)

Splits one or more rows in a `DynamicFrame` off into a new `DynamicFrame`.

The method returns a new `DynamicFrameCollection` that contains two `DynamicFrames`. The first `DynamicFrame` contains all the rows that have been split off, and the second contains the rows that remain.

- `comparison_dict` – A dictionary where the key is a path to a column, and the value is another dictionary for mapping comparators to values that the column values are compared to. For example, `{"age": {">": 10, "<": 20}}` splits off all rows whose value in the age column is greater than 10 and less than 20.
- `name1` – A name string for the `DynamicFrame` that is split off.
- `name2` – A name string for the `DynamicFrame` that remains after the specified nodes have been split off.

```
# Split up rows by ID
split_rows_collection = frame_to_split.split_rows({"id": {">": 10}}, "high", "low")
```

22. unbox

*UNBOX(PATH, FORMAT, TRANSFORMATION_CTX="", INFO="", STAGETHRESHOLD=0, TOTALTHRESHOLD=0, **OPTIONS)*

Unboxes (reformats) a string field in a `DynamicFrame` and returns a new `DynamicFrame` that contains the unboxed `DynamicRecords`.

A `DynamicRecord` represents a logical record in a `DynamicFrame`. It's similar to a row in an Apache Spark `DataFrame`, except that it is self-describing and can be used for data that doesn't conform to a fixed schema.

- `path` – A full path to the string node you want to unbox.
- `format` – A format specification (optional). You use this for an Amazon S3 or AWS Glue connection that supports multiple formats. For the formats that are supported, see Data format options for inputs and outputs in AWS Glue for Spark.
- `options` – One or more of the following:
    - `separator` – A string that contains the separator character.
    - `escaper` – A string that contains the escape character.
    - `skipFirst` – A Boolean value that indicates whether to skip the first instance.
    - `withSchema` – A string containing a JSON representation of the node's schema. The format of a schema's JSON representation is defined by the output of `StructType.json()`.
    - `withHeader` – A Boolean value that indicates whether a header is included.

```
unboxed = mapped_with_string.unbox("AddressString", "json")
|-- AddressString: struct
|    |-- Street: string
|    |-- City: string
|    |-- State: string
|    |-- Zip.Code: string
|    |-- Array: array
|    |    |-- element: string
If we choose as path as csv we need to use options.
```

23. `union`

`UNION(FRAME1, FRAME2, TRANSFORMATION_CTX = "", INFO = "", STAGETHRESHOLD = 0, TOTALTHRESHOLD = 0)`

Union two DynamicFrames. Returns DynamicFrame containing all records from both input DynamicFrames. This transform may return different results from the union of two DataFrames with equivalent data. If you need the Spark DataFrame union behavior, consider using `toDF`.

- `frame1` – First DynamicFrame to union.
- `frame2` – Second DynamicFrame to union.

24. unnest_ddb_json

Unnests nested columns in a `DynamicFrame` that are specifically in the DynamoDB JSON structure, and returns a new unnested `DynamicFrame`. Columns that are of an array of struct types will not be unnested. Note that this is a specific type of unnesting transform that behaves differently from the regular `unnest` transform and requires the data to already be in the DynamoDB JSON structure.

unnest_ddb_json(transformation_ctx="", info="", stageThreshold=0, totalThreshold=0)

```
unnested = dynamicFrame.unnest_ddb_json()
```

25. write

WRITE(CONNECTION_TYPE, CONNECTION_OPTIONS, FORMAT, FORMAT_OPTIONS, ACCUMULATOR_SIZE)

Syntax:

```
write(
    connection_type,        # e.g., "s3", "jdbc", "redshift"
    connection_options,     # Dictionary with destination details
    format,                 # e.g., "json", "csv", "parquet"
    format_options,         # Optional format-specific settings
    accumulator_size        # Optional buffer size for writing
)
```

Example:

```
glueContext.write_dynamic_frame.from_options(
    frame=dynamic_frame,
    connection_type="s3",
    connection_options={
        "path": "s3://my-bucket/output/",
        "partitionKeys": ["year", "month"]
    },
    format="csv",
    format_options={
        "separator": ",",
        "quoteChar": "\""
    },
    accumulator_size=1024
)
```

---

26. EvaluateDataQuality class

(frame, ruleset, publishing_options = {})

- `frame` – The `DynamicFrame` that you want evaluate the data quality of.

- `ruleset` – A Data Quality Definition Language (DQDL) ruleset in string format. To learn more about DQDL, see the Data Quality Definition Language (DQDL) reference guide.
- `publishing_options` – A dictionary that specifies the following options for publishing evaluation results and metrics:
    - `dataQualityEvaluationContext` – A string that specifies the namespace under which AWS Glue should publish Amazon CloudWatch metrics and the data quality results. The aggregated metrics appear in CloudWatch, while the full results appear in the AWS Glue Studio interface.
        - Required: No
        - Default value: `default_context`
    - `enableDataQualityCloudWatchMetrics` – Specifies whether the results of the data quality evaluation should be published to CloudWatch. You specify a namespace for the metrics using the `dataQualityEvaluationContext` option.
        - Required: No
        - Default value: False
    - `enableDataQualityResultsPublishing` – Specifies whether the data quality results should be visible on the **Data Quality** tab in the AWS Glue Studio interface.
        - Required: No
        - Default value: True
    - `resultsS3Prefix` – Specifies the Amazon S3 location where AWS Glue can write the data quality evaluation results.
        - Required: No
        - Default value: "" (empty string)

```
# Evaluate data quality
dqResults = EvaluateDataQuality.apply(
    frame=legislatorsAreas,
    ruleset=ruleset,
    publishing_options={
        "dataQualityEvaluationContext": "legislatorsAreas",
        "enableDataQualityCloudWatchMetrics": True,
        "enableDataQualityResultsPublishing": True,
        "resultsS3Prefix": "amzn-s3-demo-bucket1",
    },
)
```

assertErrorThreshold

`assertErrorThreshold( )` – An assert for errors in the transformations that created this `DynamicFrame`. Returns an `Exception` from the underlying `DataFrame`.

errorsAsDynamicFrame

`errorsAsDynamicFrame( )` – Returns a `DynamicFrame` that has error records nested inside.

errorsCount

`errorsCount( )` – Returns the total number of errors in a `DynamicFrame`.

stageErrorsCount

`stageErrorsCount` – Returns the number of errors that occurred in the process of generating this `DynamicFrame`.

Loading from PostgreSQL with SQL SELECT query:

```
# Load only specific columns from a large table
selected_columns_dyf =
glueContext.create_dynamic_frame.from_options(
    connection_type="mysql",
    connection_options={
        "url": "jdbc:mysql://your-mysql-host:3306/your-database",
        "user": "your-username",
        "password": "your-password",
        "dbtable": "(SELECT order_id, customer_id FROM
large_orders_table) AS selected_data"
    }
)

# Alternative approach using column selection in query
efficient_load_dyf = glueContext.create_dynamic_frame.from_options(
    connection_type="postgresql",
    connection_options={
        "url": "jdbc:postgresql://your-postgres-host:5432/your-
database",
        "user": "your-username",
        "password": "your-password",
        "query": "SELECT product_id, product_name FROM products
WHERE category = 'electronics'"
    }
)
```

24. DynamicFrameCollection class:

Keys

`keys( )` – Returns a list of the keys in this collection, which generally consists of the names of the corresponding `DynamicFrame` values.

Values

`values(key)` – Returns a list of the `DynamicFrame` values in this collection.

Select

`SELECT(KEY)`
Returns the `DynamicFrame` that corresponds to the specfied key (which is generally the name of the `DynamicFrame`).

- `key` – A key in the `DynamicFrameCollection`, which usually represents the name of a `DynamicFrame`.

Map

`MAP(CALLABLE, TRANSFORMATION_CTX="")`
Uses a passed-in function to create and return a new `DynamicFrameCollection` based on the `DynamicFrames` in this collection.

- `callable` – A function that takes a `DynamicFrame` and the specified transformation context as parameters and returns a `DynamicFrame`.
- `transformation_ctx` – A transformation context to be used by the callable (optional).

Flatmap

`FLATMAP(F, TRANSFORMATION_CTX="")`
Uses a passed-in function to create and return a new `DynamicFrameCollection` based on the `DynamicFrames` in this collection.

- `f` – A function that takes a `DynamicFrame` as a parameter and returns a `DynamicFrame` or `DynamicFrameCollection`.

25. DynamicFrameWriter class:

from_options

FROM_OPTIONS(FRAME, CONNECTION_TYPE, CONNECTION_OPTIONS={}, FORMAT=NONE, FORMAT_OPTIONS={}, TRANSFORMATION_CTX="")

Writes a `DynamicFrame` using the specified connection and format.

- `frame` – The `DynamicFrame` to write.
- `connection_type` – The connection type. Valid values include `s3`, `mysql`, `postgresql`, `redshift`, `sqlserver`, and `oracle`.
- `connection_options` – Connection options, such as path and database table (required). For a `connection_type` of `s3`, an Amazon S3 path is defined. `connection_options = {"path": "s3://aws-glue-target/temp"}`
- For JDBC connections, several properties must be defined. Note that the database name must be part of the URL. `connection_options = {"url": "jdbc-url/database", "user": "username", "password": passwordVariable,"dbtable": "table-name", "redshiftTmpDir": "s3-tempdir-path"}`
- `format` – A format specification (optional). This is used for an Amazon Simple Storage Service (Amazon S3) or an AWS Glue connection that supports multiple formats. See Data format options for inputs and outputs in AWS Glue for Spark for the formats that are supported.
- `format_options` – Format options for the specified format. See Data format options for inputs and outputs in AWS Glue for Spark for the formats that are supported.

from_catalog

FROM_CATALOG(FRAME, NAME_SPACE, TABLE_NAME, REDSHIFT_TMP_DIR="", TRANSFORMATION_CTX="")

Writes a `DynamicFrame` using the specified catalog database and table name.

- `frame` – The `DynamicFrame` to write.
- `name_space` – The database to use.
- `table_name` – The `table_name` to use.
- `redshift_tmp_dir` – An Amazon Redshift temporary directory to use (optional).
- `transformation_ctx` – A transformation context to use (optional).

- `additional_options` – Additional options provided to AWS Glue.

  To write to Lake Formation governed tables, you can use these additional options:

  - `transactionId` – (String) The transaction ID at which to do the write to the Governed table. This transaction can not be already committed or aborted, or the write will fail.
  - `callDeleteObjectsOnCancel` – (Boolean, optional) If set to `true` (default), AWS Glue automatically calls the `DeleteObjectsOnCancel` API after the object is written to Amazon S3. For more information, see DeleteObjectsOnCancel in the *AWS Lake Formation Developer Guide*.

  EXAMPLE: WRITING TO A GOVERNED TABLE IN LAKE FORMATION

```
txId = glueContext.start_transaction(read_only=False)

glueContext.write_dynamic_frame.from_catalog(

    frame=dyf,

    database = db,

    table_name = tbl,

    transformation_ctx = "datasource0",

    additional_options={"transactionId":txId})

...

glueContext.commit_transaction(txId)
```

26.DynamicFrameReader class

from_options

FROM_OPTIONS(CONNECTION_TYPE, CONNECTION_OPTIONS={}, FORMAT=NONE, FORMAT_OPTIONS={}, TRANSFORMATION_CTX="")

Reads a `DynamicFrame` using the specified connection and format.

- `connection_type` – The connection type. Valid values include `s3`, `mysql`, `postgresql`, `redshift`, `sqlserver`, `oracle`, `dynamodb`, and `snowflake`.

- `connection_options` – Connection options, such as path and database table (optional). For more information, see Connection types and options for ETL in AWS Glue for Spark . For a `connection_type` of `s3`, Amazon S3 paths are defined in an array.
  `connection_options = {"paths": [ "s3://amzn-s3-demo-bucket/object_a", "s3://amzn-s3-demo-bucket/object_b"]}`
- For JDBC connections, several properties must be defined. Note that the database name must be part of the URL. It can optionally be included in the connection options.
- `format` – A format specification (optional). This is used for an Amazon Simple Storage Service (Amazon S3) or an AWS Glue connection that supports multiple formats. See Data format options for inputs and outputs in AWS Glue for Spark for the formats that are supported.
- `format_options` – Format options for the specified format. See Data format options for inputs and outputs in AWS Glue for Spark for the formats that are supported.

from_catalog

FROM_CATALOG(DATABASE, TABLE_NAME, REDSHIFT_TMP_DIR="", TRANSFORMATION_CTX="", PUSH_DOWN_PREDICATE="", ADDITIONAL_OPTIONS={})

Reads a `DynamicFrame` using the specified catalog namespace and table name.

- `database` – The database to read from.
- `table_name` – The name of the table to read from.
- `redshift_tmp_dir` – An Amazon Redshift temporary directory to use (optional if not reading data from Redshift).
- `transformation_ctx` – The transformation context to use (optional).
- `push_down_predicate` – Filters partitions without having to list and read all the files in your dataset. For more information, see Pre-filtering using pushdown predicates.
- `additional_options` – Additional options provided to AWS Glue.
  - To use a JDBC connection that performs parallel reads, you can set the `hashfield`, `hashexpression`, or `hashpartitions` options. For example:

```
additional_options = {"hashfield": "month"}
```

27. GlueContext class

create_dynamic_frame_from_catalog

CREATE_DYNAMIC_FRAME_FROM_CATALOG(DATABASE, TABLE_NAME, REDSHIFT_TMP_DIR, TRANSFORMATION_CTX = "", PUSH_DOWN_PREDICATE= "", ADDITIONAL_OPTIONS = {}, CATALOG_ID = NONE)

- **database**: Name of the Glue Data Catalog database.
- **table_name**: Name of the table in the catalog.
- **redshift_tmp_dir**: Temporary directory in Amazon S3 for intermediate data (used especially when reading from Redshift).
- **transformation_ctx**: Optional context name for debugging and logging.
- **push_down_predicate**: Optional SQL-like filter to reduce the amount of data read.
- **additional_options**: Dictionary of extra options (e.g., partitioning, reading modes).
- **catalog_id**: Optional AWS account ID if accessing a catalog in another account.

```
# Create DynamicFrame from catalog
orders_df = glueContext.create_dynamic_frame_from_catalog(
    database="sales_db",
    table_name="orders_2023",
    redshift_tmp_dir="s3://my-temp-dir/redshift/",
    transformation_ctx="orders_ctx",
    push_down_predicate="order_date >= '2023-01-01' AND order_date < '2023-02-01'",
    additional_options={"useCatalog": True, "readThroughputPercent": 0.5}
)
```

create_dynamic_frame_from_options

CREATE_DYNAMIC_FRAME_FROM_OPTIONS(CONNECTION_TYPE, CONNECTION_OPTIONS={}, FORMAT=NONE, FORMAT_OPTIONS={}, TRANSFORMATION_CTX = "")

Returns a `DynamicFrame` created with the specified connection and format.

create_sample_dynamic_frame_from_catalog

CREATE_SAMPLE_DYNAMIC_FRAME_FROM_CATALOG(DATABASE, TABLE_NAME, NUM, REDSHIFT_TMP_DIR, TRANSFORMATION_CTX = "", PUSH_DOWN_PREDICATE= "", ADDITIONAL_OPTIONS = {}, SAMPLE_OPTIONS = {}, CATALOG_ID = NONE)

Returns a sample `DynamicFrame` that is created using a Data Catalog database and table name. The `DynamicFrame` only contains first `num` records from a datasource.

create_sample_dynamic_frame_from_options

CREATE_SAMPLE_DYNAMIC_FRAME_FROM_OPTIONS(CONNECTION_TYPE, CONNECTION_OPTIONS={}, NUM, SAMPLE_OPTIONS={}, FORMAT=NONE, FORMAT_OPTIONS={}, TRANSFORMATION_CTX = "")

Returns a sample `DynamicFrame` created with the specified connection and format. The `DynamicFrame` only contains first `num` records from a datasource.

purge_table

PURGE_TABLE(CATALOG_ID=NONE, DATABASE="", TABLE_NAME="", OPTIONS={}, TRANSFORMATION_CTX="")

Deletes files from Amazon S3 for the specified catalog's database and table. If all files in a partition are deleted, that partition is also deleted from the catalog. We don't support purge_table action on tables registered with Lake Formation.

If you want to be able to recover deleted objects, you can turn on object versioning on the Amazon S3 bucket. When an object is deleted from a bucket that doesn't have object versioning enabled, the object can't be recovered. For more information about how to recover deleted objects in a version-enabled bucket, see How can I retrieve an Amazon S3 object that was deleted? in the AWS Support Knowledge Center.

- `catalog_id` – The catalog ID of the Data Catalog being accessed (the account ID of the Data Catalog). Set to `None` by default. `None` defaults to the catalog ID of the calling account in the service.
- `database` – The database to use.
- `table_name` – The name of the table to use.
- `options` – Options to filter files to be deleted and for manifest file generation.
    - `retentionPeriod` – Specifies a period in number of hours to retain files. Files newer than the retention period are retained. Set to 168 hours (7 days) by default.
    - `partitionPredicate` – Partitions satisfying this predicate are deleted. Files within the retention period in these partitions are not deleted. Set to `""` – empty by default.
    - `excludeStorageClasses` – Files with storage class in the `excludeStorageClasses` set are not deleted. The default is `Set()` – an empty set.

- `manifestFilePath` – An optional path for manifest file generation. All files that were successfully purged are recorded in `Success.csv`, and those that failed in `Failed.csv`

```
glueContext.purge_table("database", "table", {"partitionPredicate":
"(month=='march')", "retentionPeriod": 1, "excludeStorageClasses":
["STANDARD_IA"], "manifestFilePath": "s3://bucketmanifest/"})
```

purge_s3_path

PURGE_S3_PATH(S3_PATH, OPTIONS={}, TRANSFORMATION_CTX="")

Deletes files from the specified Amazon S3 path recursively.

28.Connections

28.1 DynamoDB connections

You can use AWS Glue for Spark to read from and write to tables in DynamoDB in AWS Glue. You connect to DynamoDB using IAM permissions attached to your AWS Glue job. AWS Glue supports writing data into another AWS account's DynamoDB table.

DynamoDB export connector:

```
dyf = glue_context.create_dynamic_frame.from_options(
    connection_type="dynamodb",
    connection_options={
        "dynamodb.export": "ddb",
        "dynamodb.tableArn": test_source,
        "dynamodb.s3.bucket": bucket_name,
        "dynamodb.s3.prefix": bucket_prefix,
        "dynamodb.s3.bucketOwner": account_id_of_bucket,
    }
)
```

- `"dynamodb.export"`: (Required) A string value:
  - If set to `ddb` enables the AWS Glue DynamoDB export connector where a new `ExportTableToPointInTimeRequest` will be invoked during the AWS Glue job. A new export will be generated with the location passed from `dynamodb.s3.bucket` and `dynamodb.s3.prefix`.
  - If set to `s3` enables the AWS Glue DynamoDB export connector but skips the creation of a new DynamoDB export and instead uses the `dynamodb.s3.bucket` and `dynamodb.s3.prefix` as the Amazon S3 location of a past export of that table.
- `"dynamodb.tableArn"`: (Required) The DynamoDB table to read from.
- `"dynamodb.s3.bucket"`: (Optional) Indicates the Amazon S3 bucket location in which the DynamoDB `ExportTableToPointInTime` process is to be conducted. The file format for the export is DynamoDB JSON.

    - `"dynamodb.s3.prefix"`: (Optional) Indicates the Amazon S3 prefix location inside the Amazon S3 bucket in which the DynamoDB `ExportTableToPointInTime` loads are to be stored. If neither `dynamodb.s3.prefix` nor `dynamodb.s3.bucket` are specified, these values will default to the Temporary Directory location specified in the AWS Glue job configuration. For more information, see Special Parameters Used by AWS Glue.
    - `"dynamodb.s3.bucketOwner"`: Indicates the bucket owner needed for cross-account Amazon S3 access.
- `"dynamodb.sts.roleArn"`: (Optional) The IAM role ARN to be assumed for cross-account access and/or cross-Region access for the DynamoDB table. Note: The same IAM role ARN will be used to access the Amazon S3 location specified for the `ExportTableToPointInTime` request.

For Cross-Region:

```
dyf = glue_context.create_dynamic_frame_from_options(
    connection_type="dynamodb",
    connection_options={
    "dynamodb.region": "us-east-1",
    "dynamodb.input.tableName": "test_source",
    "dynamodb.sts.roleArn": "<DynamoDBCrossAccessRole's ARN>"
    }
)
```

28.2 Kinesis connections

You can use a Kinesis connection to read and write to Amazon Kinesis data streams using information stored in a Data Catalog table, or by providing information to directly access the data stream. You can read information from Kinesis into a Spark DataFrame, then convert it to a AWS Glue DynamicFrame. You can write DynamicFrames to Kinesis in a JSON format. If you directly access the data stream, use these options to provide the information about how to access the data stream.

To connect to a Kinesis data stream in an AWS Glue Spark job, you will need some prerequisites:

- If reading, the AWS Glue job must have Read access level IAM permissions to the Kinesis data stream.
- If writing, the AWS Glue job must have Write access level IAM permissions to the Kinesis data stream.

Reading from Kinesis streams

```
kinesis_options =
   { "streamARN": "arn:aws:kinesis:us-east-
2:777788889999:stream/fromOptionsStream",
     "startingPosition": "TRIM_HORIZON",
```

```
    "inferSchema": "true",
    "classification": "json"
  }
data_frame_datasource0 =
glueContext.create_data_frame.from_options(connection_type="kinesis"
, connection_options=kinesis_options)
```

Designates connection options for Amazon Kinesis Data Streams.

Use the following connection options for Kinesis streaming data sources:

- `"streamARN"` (Required) Used for Read/Write. The ARN of the Kinesis data stream.
- `"classification"` (Required for read) Used for Read. The file format used by the data in the record. Required unless provided through the Data Catalog.
- `"streamName"` – (Optional) Used for Read. The name of a Kinesis data stream to read from. Used with `endpointUrl`.
- `"endpointUrl"` – (Optional) Used for Read. Default: "https://kinesis.us-east-1.amazonaws.com". The AWS endpoint of the Kinesis stream. You do not need to change this unless you are connecting to a special region.
- `"delimiter"` (Optional) Used for Read. The value separator used when `classification` is CSV. Default is "`,`."
- `"startingPosition"`: (Optional) Used for Read. The starting position in the Kinesis data stream to read data from. The possible values are `"latest"`, `"trim_horizon"`, `"earliest"`, or a Timestamp string in UTC format in the pattern `yyyy-mm-ddTHH:MM:SSZ` (where `Z` represents a UTC timezone offset with a +/-. For example "2023-04-04T08:00:00-04:00"). The default value is `"latest"`. Note: the Timestamp string in UTC Format for `"startingPosition"` is supported only for AWS Glue version 4.0 or later.
- **`TRIM_HORIZON`**:
    - Starts reading from the **oldest available record** in the stream.
    - Useful when you want to process **all historical data**.
- **`LATEST`**:
    - Starts reading from the **most recent record**.
    - Useful for **real-time processing**, ignoring older data.

28.3 Redshift connections

You can use AWS Glue for Spark to read from and write to tables in Amazon Redshift databases. When connecting to Amazon Redshift databases, AWS Glue moves data through Amazon S3 to achieve maximum throughput, using the Amazon Redshift SQL `COPY` and `UNLOAD` commands.

Configuring Redshift connections

To use Amazon Redshift clusters in AWS Glue, you will need some prerequisites:

- An Amazon S3 directory to use for temporary storage when reading from and writing to the database.
- An Amazon VPC enabling communication between your Amazon Redshift cluster, your AWS Glue job and your Amazon S3 directory.
- Appropriate IAM permissions on the AWS Glue job and Amazon Redshift cluster.

Configuring IAM roles:

Set up the role for the Amazon Redshift cluster

set up the role for the AWS Glue job

Set up Amazon VPC

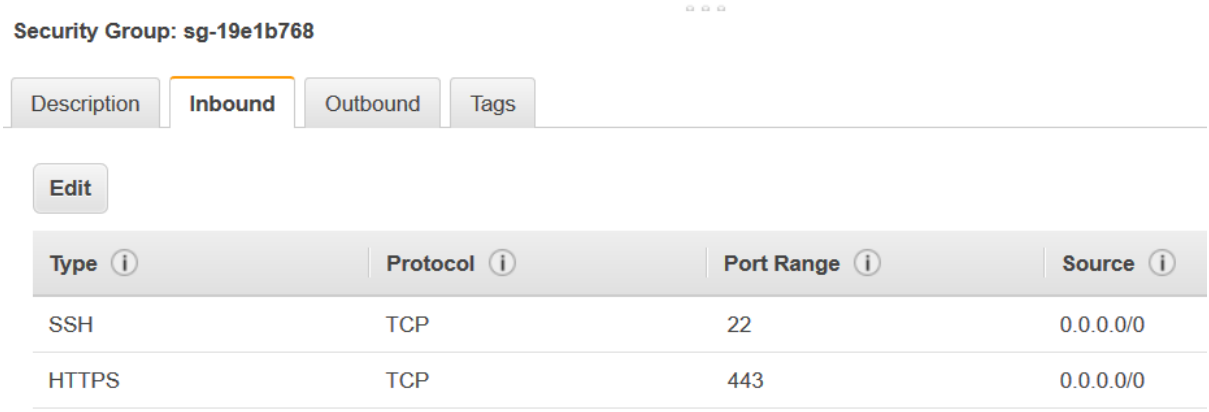
To set up access for Amazon Redshift data stores

1. Sign in to the AWS Management Console and open the Amazon Redshift console at https://console.aws.amazon.com/redshiftv2/.
2. In the left navigation pane, choose **Clusters**.
3. Choose the cluster name that you want to access from AWS Glue.
4. In the **Cluster Properties** section, choose a security group in **VPC security groups** to allow AWS Glue to use. Record the name of the security group that you chose for future reference. Choosing the security group opens the Amazon EC2 console **Security Groups** list.
5. Choose the security group to modify and navigate to the **Inbound** tab.
6. Add a self-referencing rule to allow AWS Glue components to communicate. Specifically, add or confirm that there is a rule of **Type** `All TCP`, **Protocol** is `TCP`, **Port Range** includes all ports, and whose **Source** is the same security group name as the **Group ID**.

   The inbound rule looks similar to the following:

Type	Protocol	Port range	Source
All TCP	TCP	0–65535	*database-security-group*

   For example:



7. Add a rule for outbound traffic also. Either open outbound traffic to all ports, for example:

Type	Protocol	Port range	Destination
All Traffic	ALL	ALL	0.0.0.0/0

8. Or create a self-referencing rule where **Type** `All TCP`, **Protocol** is `TCP`, **Port Range** includes all ports, and whose **Destination** is the same security group name as the **Group ID**. If using an Amazon S3 VPC endpoint, also add an HTTPS rule for Amazon S3 access. The *`s3-prefix-list-id`* is required in the security group rule to allow traffic from the VPC to the Amazon S3 VPC endpoint.
9. For example:

Type	Protocol	Port range	Destination
All TCP	TCP	0–65535	*`security-group`*
HTTPS	TCP	443	*`s3-prefix-list-id`*

Set up AWS Glue

You will need to create an AWS Glue Data Catalog connection that provides Amazon VPC connection information.

To configure Amazon Redshift Amazon VPC connectivity to AWS Glue in the console

1. Create a Data Catalog connection by following the steps in: Adding an AWS Glue connection. After creating the connection, keep the connection name, *connectionName*, for the next step.
    - When selecting a **Connection type**, select **Amazon Redshift**.
    - When selecting a **Redshift cluster**, select your cluster by name.
    - Provide default connection information for a Amazon Redshift user on your cluster.
    - Your Amazon VPC settings will be automatically configured.

Note

You will need to manually provide `PhysicalConnectionRequirements` for your Amazon VPC when creating an **Amazon Redshift** connection through the AWS SDK.

2. In your AWS Glue job configuration, provide *connectionName* as an **Additional network connection**.

Reading from Amazon Redshift tables:

Configuration: In your function options you will identify your Data Catalog Table with the `database` and `table_name` parameters. You will identify your Amazon S3 temporary directory with `redshift_tmp_dir`. You will also provide *rs-role-name* using the `aws_iam_role` key in the `additional_options` parameter.

```
glueContext.create_dynamic_frame.from_catalog(
    database = "redshift-dc-database-name",
    table_name = "redshift-table-name",
    redshift_tmp_dir = args["temp-s3-dir"],
    additional_options = {"aws_iam_role": "arn:aws:iam::role-account-id:role/rs-role-name"})
```

Writing to Amazon Redshift tables

```
glueContext.write_dynamic_frame.from_catalog(
```

```
    frame = input dynamic frame,
    database = "redshift-dc-database-name",
    table_name = "redshift-table-name",
    redshift_tmp_dir = args["temp-s3-dir"],
    additional_options = {"aws_iam_role": "arn:aws:iam::account-
id:role/rs-role-name"})
```

- `"redshiftTmpDir"`: (Required) The Amazon S3 path where temporary data can be staged when copying out of the database.

28.4 Kafka connections

You can use a Kafka connection to read and write to Kafka data streams using information stored in a Data Catalog table, or by providing information to directly access the data stream. The connection supports a Kafka cluster or an Amazon Managed Streaming for Apache Kafka cluster. You can read information from Kafka into a Spark DataFrame, then convert it to a AWS Glue DynamicFrame. You can write DynamicFrames to Kafka in a JSON format. If you directly access the data stream, use these options to provide the information about how to access the data stream.

Reading from Kafka streams

```
kafka_options =
    { "connectionName": "ConfluentKafka",
      "topicName": "kafka-auth-topic",
      "startingOffsets": "earliest",
      "inferSchema": "true",
      "classification": "json"
    }
data_frame_datasource0 =
glueContext.create_data_frame.from_options(connection_type="kafka",
connection_options=kafka_options)
```

Writing to Kafka streams

```
kafka_options =
    { "connectionName": "ConfluentKafka",
      "topicName": "kafka-auth-topic",
      "classification": "json"
    }
data_frame_datasource0 =
glueContext.write_dynamic_frame.from_options(connection_type="kafka"
, connection_options=kafka_options)
```

- `"bootstrap.servers"` (Required) A list of bootstrap server URLs, for example, as `b-1.vpc-test-2.o4q88o.c6.kafka.us-east-1.amazonaws.com:9094`. This option

must be specified in the API call or defined in the table metadata in the Data Catalog.

- `"security.protocol"` (Required) The protocol used to communicate with brokers. The possible values are `"SSL"` or `"PLAINTEXT"`.
- `"topicName"` (Required) A comma-separated list of topics to subscribe to. You must specify one and only one of `"topicName"`, `"assign"` or `"subscribePattern"`.
- `"assign"`: (Required) A JSON string specifying the specific `TopicPartitions` to consume. You must specify one and only one of `"topicName"`, `"assign"` or `"subscribePattern"`.

  Example: '{"topicA":[0,1],"topicB":[2,4]}'
- `"subscribePattern"`: (Required) A Java regex string that identifies the topic list to subscribe to. You must specify one and only one of `"topicName"`, `"assign"` or `"subscribePattern"`.

  Example: 'topic.*'
- `"classification"` (Required) The file format used by the data in the record. Required unless provided through the Data Catalog.
- `"delimiter"` (Optional) The value separator used when `classification` is CSV. Default is `","`.

28.5 JDBC connections

Certain, typically relational, database types support connecting through the JDBC standard.

The JDBC connectionType values include the following:

- `"connectionType": "sqlserver"`: Designates a connection to a Microsoft SQL Server database.
- `"connectionType": "mysql"`: Designates a connection to a MySQL database.
- `"connectionType": "oracle"`: Designates a connection to an Oracle database.
- `"connectionType": "postgresql"`: Designates a connection to a PostgreSQL database.
- `"connectionType": "redshift"`: Designates a connection to an Amazon Redshift database.

◆ Required Parameters

- **`url`**: Specifies the JDBC connection string used to connect to the database.
- **`dbtable`**: Indicates the table to read from or write to. If the database supports schemas, it can include the schema name.
- **`user`**: The username used for authenticating the JDBC connection.

- **`password`**: The password associated with the specified user.

- Optional Parameters for Custom JDBC Drivers

  - **`customJdbcDriverS3Path`**: Defines the Amazon S3 path to a custom JDBC driver JAR file.
  - **`customJdbcDriverClassName`**: Specifies the fully qualified class name of the custom JDBC driver.

- Performance Optimization Parameters

  - **`bulkSize`**: Sets the number of rows to insert in parallel during bulk writes to improve performance.

- Parallel Read Parameters

  - **`hashfield`**: Identifies a column in the table used to partition data for parallel reads.
  - **`hashexpression`**: A SQL expression that returns a whole number, used to partition data for parallel reads.
  - **`hashpartitions`**: Specifies the number of partitions to divide the data into for parallel reads.

- Sampling Parameters

  - **`sampleQuery`**: A custom SQL query used to retrieve a subset of data from the table.
  - **`enablePartitioningForSampleQuery`**: Enables partitioning when using `sampleQuery`, allowing parallel reads.
  - **`sampleSize`**: Limits the number of rows returned when partitioning is enabled with `sampleQuery`.

```
{
 "url": "jdbc:mysql://your-db-hostname:3306/your_database",
 "dbtable": "public.employee_data",
 "user": "admin_user",
 "password": "securePassword123",
 "customJdbcDriverS3Path": "s3://my-bucket/drivers/mysql-connector-java-8.0.28.jar",
 "customJdbcDriverClassName": "com.mysql.cj.jdbc.Driver",
 "bulkSize": 1000, # Write 1000 rows in parallel per batch


 "hashfield": "employee_id",
 "hashpartitions": 5,
 "sampleQuery": "SELECT * FROM employee_data WHERE department = 'Sales' AND",
 "enablePartitioningForSampleQuery": true,
 "sampleSize": 500
```

}

28.6 MongoDB connections

You can use AWS Glue for Spark to read from and write to tables in MongoDB and MongoDB Atlas in AWS Glue 4.0 and later versions. You can connect to MongoDB using username and password credentials credentials stored in AWS Secrets Manager through a AWS Glue connection.

To configure a connection to MongoDB:

1. Optionally, in AWS Secrets Manager, create a secret using your MongoDB credentials. To create a secret in Secrets Manager, follow the tutorial available in Create an AWS Secrets Manager secret in the AWS Secrets Manager documentation. After creating the secret, keep the Secret name, `secretName` for the next step.
    - When selecting **Key/value pairs**, create a pair for the key `username` with the value `mongodbUser`.

      When selecting **Key/value pairs**, create a pair for the key `password` with the value `mongodbPass`.
2. In the AWS Glue console, create a connection by following the steps in Adding an AWS Glue connection. After creating the connection, keep the connection name, `connectionName`, for future use in AWS Glue.
    - When selecting a **Connection type**, select **MongoDB** or **MongoDB Atlas**.
    - When selecting **MongoDB URL** or **MongoDB Atlas URL**, provide the hostname of your MongoDB instance.

      A MongoDB URL is provided in the format `mongodb://mongoHost:mongoPort/mongoDBname`.

      A MongoDB Atlas URL is provided in the format `mongodb+srv://mongoHost:mongoPort/mongoDBname`.

      Providing the default database for the connection, `mongoDBname` is optional.
    - If you chose to create an Secrets Manager secret, choose the AWS Secrets Manager **Credential type**.

      Then, in **AWS Secret** provide `secretName`.
    - If you choose to provide **Username and password**, provide `mongodbUser` and `mongodbPass`.
3. In the following situations, you may require additional configuration:
    - For MongoDB instances hosted on AWS in an Amazon VPC

    - You will need to provide Amazon VPC connection information to the AWS Glue connection that defines your MongoDB security credentials. When creating or updating your connection, set **VPC**, **Subnet** and **Security groups** in **Network options**.

After creating a AWS Glue MongoDB connection, you will need to perform the following actions before calling your connection method:

- If you chose to create an Secrets Manager secret, grant the IAM role associated with your AWS Glue job permission to read `secretName`.
- In your AWS Glue job configuration, provide `connectionName` as an **Additional network connection**.

Reading from MongoDB using a AWS Glue connection

```
mongodb_read = glueContext.create_dynamic_frame.from_options(
    connection_type="mongodb",
    connection_options={
        "connectionName": "connectionName",
        "database": "mongodbName",
        "collection": "mongodbCollection",
        "partitioner":
"com.mongodb.spark.sql.connector.read.partitioner.SinglePartitionPar
titioner",
        "partitionerOptions.partitionSizeMB": "10",
        "partitionerOptions.partitionKey": "_id",
        "disableUpdateUri": "false",
    }
)
```

Writing to MongoDB tables

```
glueContext.write_dynamic_frame.from_options(
    frame=dynamicFrame,
    connection_type="mongodb",
    connection_options={
        "connectionName": "connectionName",
        "database": "mongodbName",
        "collection": "mongodbCollection",
        "disableUpdateUri": "false",
        "retryWrites": "false",
    },
)
```

These connection properties are shared between source and sink connections:

- `connectionName` — Used for Read/Write. The name of a AWS Glue MongoDB connection configured to provide auth and networking information to your connection method. When a AWS Glue connection is configured as described

in the previous section, Configuring MongoDB connections, providing `connectionName` will replace the need to provide the `"uri"`, `"username"` and `"password"` connection options.

- `"uri"`: (Required) The MongoDB host to read from, formatted as `mongodb://<host>:<port>`. Used in AWS Glue versions prior to AWS Glue 4.0.
- `"connection.uri"`: (Required) The MongoDB host to read from, formatted as `mongodb://<host>:<port>`. Used in AWS Glue 4.0 and later versions.
- `"username"`: (Required) The MongoDB user name.
- `"password"`: (Required) The MongoDB password.
- `"database"`: (Required) The MongoDB database to read from. This option can also be passed in `additional_options` when calling `glue_context.create_dynamic_frame_from_catalog` in your job script.
- `"collection"`: (Required) The MongoDB collection to read from. This option can also be passed in `additional_options` when calling `glue_context.create_dynamic_frame_from_catalog` in your job script.

28.7 Snowflake connections

You can use AWS Glue for Spark to read from and write to tables in Snowflake in AWS Glue 4.0 and later versions. You can read from Snowflake with a SQL query. You can connect to Snowflake using a user and password.

TO MANAGE YOUR CONNECTION CREDENTIALS WITH AWS GLUE

1. In Snowflake, generate a user, *`snowflakeUser`* and password, *`snowflakePassword`*.
2. In AWS Secrets Manager, create a secret using your Snowflake credentials. To create a secret in Secrets Manager, follow the tutorial available in Create an AWS Secrets Manager secret in the AWS Secrets Manager documentation. After creating the secret, keep the Secret name, *`secretName`* for the next step.
    - When selecting **Key/value pairs**, create a pair for *`snowflakeUser`* with the key `USERNAME`.
    - When selecting **Key/value pairs**, create a pair for *`snowflakePassword`* with the key `PASSWORD`.
    - When selecting **Key/value pairs**, you can provide your Snowflake warehouse with the key `sfWarehouse`.
    - When selecting **Key/value pairs**, you can provide additional Snowflake connection properties using their corresponding Spark property names as keys. Supported properties include:

    - `sfDatabase` - Snowflake database name
    - `sfSchema` - Snowflake schema name
    - `sfRole` - Snowflake role name
    - `pem_private_key` - Private key for key-pair authentication
3. In the AWS Glue Data Catalog, create a connection by choosing **Connections**, then **Create connection**. Following the steps in the connection wizard to complete the process:
    - When selecting a **Data source**, select Snowflake, then choose **Next**.
    - Enter the connection details such as host and port. When entering the host **Snowflake URL**, provide the URL of your Snowflake instance. The URL will typically use a hostname in the form `account_identifier.snowflakecomputing.com`. However, the URL format may vary depending on your Snowflake account type (for example, AWS, Azure, or Snowflake-hosted).
    - When selecting the IAM service role, choose from the drop-down menu. This is the IAM role from your account that will be used to access AWS Secrets Manager and assign IP if VPC is specified.
    - When selecting an **AWS Secret**, provide `secretName`.
4. In the next step in the wizard, set properties for your Snowflake connection.
5. In the final step in the wizard, review your settings and then complete the process to create your connection.

Reading from Snowflake tables

Prerequisites: A Snowflake table you would like to read from. You will need the Snowflake table name, `tableName`. You will need your Snowflake url `snowflakeUrl`, username `snowflakeUser` and password `snowflakePassword`. If your Snowflake user does not have a default namespace set, you will need the Snowflake database name, `databaseName` and the schema name `schemaName`. Additionally, if your Snowflake user does not have a default warehouse set, you will need a warehouse name `warehouseName`.

```
snowflake_node = glueContext.create_dynamic_frame.from_options(
    connection_type="snowflake",
    connection_options={
        "autopushdown": "on",
        "query": "select * from sales where store='1' and
IsHoliday='TRUE'",
        "connectionName": "snowflake-glue-conn",
        "sfDatabase": "databaseName",
        "sfSchema": "schemaName",
        "sfWarehouse": "warehouseName",
```

```
    }

)
```

Additionally, you can use the `autopushdown` and `query` parameters to read a portion of a Snowflake table.

Writing to Snowflake tables

```
glueContext.write_dynamic_frame.from_options(
    connection_type="snowflake",
    connection_options={
        "connectionName": "connectionName",
        "dbtable": "tableName",
        "sfDatabase": "databaseName",
        "sfSchema": "schemaName",
        "sfWarehouse": "warehouseName",
    },

)
```

- `sfDatabase` — Required if a user default is not set in Snowflake. Used for Read/Write. The database to use for the session after connecting.
- `sfSchema` — Required if a user default is not set in Snowflake. Used for Read/Write. The schema to use for the session after connecting.
- `sfWarehouse` — Required if a user default is not set in Snowflake. Used for Read/Write. The default virtual warehouse to use for the session after connecting.
- `sfRole` — Required if a user default is not set in Snowflake. Used for Read/Write. The default security role to use for the session after connecting.
- `sfUrl` — (Required) Used for Read/Write. Specifies the hostname for your account in the following format: *`account_identifier`*`.snowflakecomputing.com`. For more information about account identifiers, see Account Identifiers in the Snowflake documentation.
- `sfUser` — (Required) Used for Read/Write. Login name for the Snowflake user.
- `sfPassword` — (Required unless `pem_private_key` provided) Used for Read/Write. Password for the Snowflake user.
- `dbtable` — Required when working with full tables. Used for Read/Write. The name of the table to be read or the table to which data is written. When reading, all columns and records are retrieved.
- `pem_private_key` — Used for Read/Write. An unencrypted b64-encoded private key string. The private key for the Snowflake user. It is common to copy this

out of a PEM file. For more information, see Key-pair authentication and key-pair rotation in the Snowflake documentation.

- `query` — Required when reading with a query. Used for Read. The exact query (`SELECT` statement) to run

29. Data format options

29.1 CSV format

You can use AWS Glue to read CSVs from Amazon S3 and from streaming sources as well as write CSVs to Amazon S3. You can read and write `bzip` and `gzip` archives containing CSV files from S3.

Read CSV files or folders from S3

```
dynamicFrame = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
    connection_options={"paths": ["s3://s3path"]},
    format="csv",
    format_options={
        "withHeader": True,
        # "optimizePerformance": True,
    },
)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
dataFrame = spark.read\

    .format("csv")\

    .option("header", "true")\

    .load("s3://s3path")
```

Write CSV files and folders to S3

```
glueContext.write_dynamic_frame.from_options(
    frame=dynamicFrame,
    connection_type="s3",
    connection_options={"path": "s3://s3path"},
    format="csv",
    format_options={
        "quoteChar": -1,
    },
)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
dataFrame.write\

    .format("csv")\

    .option("quote", None)\

    .mode("append")\

    .save("s3://s3path")
```

You can use the following `format_options` wherever AWS Glue libraries specify `format="csv"`:

- `separator` –Specifies the delimiter character. The default is a comma, but any other character can be specified.
  - **Type:** Text, **Default:** `","`
- `escaper` – Specifies a character to use for escaping. This option is used only when reading CSV files, not writing. If enabled, the character that immediately follows is used as-is, except for a small set of well-known escapes (`\n`, `\r`, `\t`, and `\0`).
  - **Type:** Text, **Default:** none
- `quoteChar` – Specifies the character to use for quoting. The default is a double quote. Set this to `-1` to turn off quoting entirely.
  - **Type:** Text, **Default:** `'"'`
- `multiLine` – Specifies whether a single record can span multiple lines. This can occur when a field contains a quoted new-line character. You must set this option to `True` if any record spans multiple lines.
  Enabling `multiLine` might decrease performance because it requires more cautious file-splitting while parsing.
  - **Type:** Boolean, **Default:** `false`
- `withHeader` – Specifies whether to treat the first line as a header. This option can be used in the `DynamicFrameReader` class.
  - **Type:** Boolean, **Default:** `false`
- `writeHeader` – Specifies whether to write the header to output. This option can be used in the `DynamicFrameWriter` class.
  - **Type:** Boolean, **Default:** `true`
- `skipFirst` – Specifies whether to skip the first data line.
  - **Type:** Boolean, **Default:** `false`

- `optimizePerformance` – Specifies whether to use the advanced SIMD CSV reader along with Apache Arrow–based columnar memory formats. Only available in AWS Glue 3.0+.
    - **Type:** Boolean, **Default:** `false`
- `strictCheckForQuoting` – When writing CSVs, Glue may add quotes to values it interprets as strings. This is done to prevent ambiguity in what is written out. To save time when deciding what to write, Glue may quote in certain situations where quotes are not necessary. Enabling a strict check will perform a more intensive computation and will only quote when strictly necessary. Only available in AWS Glue 3.0+.
    - **Type:** Boolean, **Default:** `false`

29.2 Parquet format

AWS Glue supports using the Parquet format. This format is a performance-oriented, column-based data format.

Read Parquet files or folders from S3

```
dynamicFrame = glueContext.create_dynamic_frame.from_options(

    connection_type = "s3",

    connection_options = {"paths": ["s3://s3path/"]},

    format = "parquet"

)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
dataFrame = spark.read.parquet("s3://s3path/")
```

Write Parquet files and folders to S3

```
glueContext.write_dynamic_frame.from_options(

    frame=dynamicFrame,

    connection_type="s3",

    format="parquet",
```

```
    connection_options={

        "path": "s3://s3path",

    },

    format_options={

        # "useGlueParquetWriter": True,

    },

)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
df.write.parquet("s3://s3path/")
```

You can use the following `format_options` wherever AWS Glue libraries specify `format="parquet"`:

- `useGlueParquetWriter` – Specifies the use of a custom Parquet writer that has performance optimizations for DynamicFrame workflows. For usage details, see Glue Parquet Writer.
    - **Type:** Boolean, **Default:**`false`
- `compression` – Specifies the compression codec used. Values are fully compatible with `org.apache.parquet.hadoop.metadata.CompressionCodecName`.
    - **Type:** Enumerated Text, **Default:** `"snappy"`
    - Values: `"uncompressed"`, `"snappy"`, `"gzip"`, and `"lzo"`
- `blockSize` – Specifies the size in bytes of a row group being buffered in memory. You use this for tuning performance. Size should divide exactly into a number of megabytes.
    - **Type:** Numerical, **Default:**`134217728`
    - The default value is equal to 128 MB.
- `pageSize` – Specifies the size in bytes of a page. You use this for tuning performance. A page is the smallest unit that must be read fully to access a single record.
    - **Type:** Numerical, **Default:**`1048576`
    - The default value is equal to 1 MB.

29.3 JSON format

AWS Glue supports using the JSON format. This format represents data structures with consistent shape but flexible contents, that aren't row or column based.

Read JSON files or folders from S3

```
dynamicFrame = glueContext.create_dynamic_frame.from_options(

    connection_type="s3",

    connection_options={"paths": ["s3://s3path"]},

    format="json",

    format_options={

        "jsonPath": "$.id",

        "multiline": True,

        # "optimizePerformance": True, -> not compatible with jsonPath, multiline

    }

)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
dataFrame = spark.read\

    .option("multiline", "true")\

    .json("s3://s3path")
```

Write JSON files and folders to S3

```
glueContext.write_dynamic_frame.from_options(

    frame=dynamicFrame,

    connection_type="s3",

    connection_options={"path": "s3://s3path"},

    format="json"
```

```
)
```

You can also use DataFrames in a script (`pyspark.sql.DataFrame`).

```
df.write.json("s3://s3path/")
```

You can use the following `format_options` values with `format="json"`:

- `jsonPath` — A JsonPath expression that identifies an object to be read into records. This is particularly useful when a file contains records nested inside an outer array. For example, the following JsonPath expression targets the `id` field of a JSON object.

  ```
  format="json", format_options={"jsonPath": "$.id"}
  ```

- `multiline` — A Boolean value that specifies whether a single record can span multiple lines. This can occur when a field contains a quoted new-line character. You must set this option to `"true"` if any record spans multiple lines. The default value is `"false"`, which allows for more aggressive file-splitting during parsing.
- `optimizePerformance` — A Boolean value that specifies whether to use the advanced SIMD JSON reader along with Apache Arrow based columnar memory formats. Only available in AWS Glue 3.0. Not compatible with `multiline` or `jsonPath`. Providing either of those options will instruct AWS Glue to fall back to the standard reader.
- `withSchema` — A String value that specifies a table schema in the format described in Manually specify the XML schema. Only used with `optimizePerformance` when reading from non-Catalog connections.

```
format="json",
format_options={
  "optimizePerformance": "true",
  "withSchema": "id:int,name:string,details:struct<age:int,city:string>"
}
```

---

30. AWS Glue API code examples using AWS SDKs

The following code examples show how to:

- Create a crawler that crawls a public Amazon S3 bucket and generates a database of CSV-formatted metadata.
- List information about databases and tables in your AWS Glue Data Catalog.
- Create a job to extract CSV data from the S3 bucket, transform the data, and load JSON-formatted output into another S3 bucket.
- List information about job runs, view transformed data, and clean up resources.

```
class GlueWrapper:
  """Encapsulates AWS Glue actions."""
     def __init__(self, glue_client):
    """
    :param glue_client: A Boto3 Glue client.
    """
    self.glue_client = glue_client
   def get_crawler(self, name):
    """
    Gets information about a crawler.
    :param name: The name of the crawler to look up.
    :return: Data about the crawler.
    """
    crawler = None
    try:
      response = self.glue_client.get_crawler(Name=name)
      crawler = response["Crawler"]
    except ClientError as err:
      if err.response["Error"]["Code"] == "EntityNotFoundException":
        logger.info("Crawler %s doesn't exist.", name)
    return crawler
```

```
def create_crawler(self, name, role_arn, db_name, db_prefix, s3_target):
    """
    Creates a crawler that can crawl the specified target and populate a
    database in your AWS Glue Data Catalog with metadata that describes the data
    in the target.
    :param name: The name of the crawler.
    :param role_arn: The Amazon Resource Name (ARN) of an AWS Identity and Access
                     Management (IAM) role that grants permission to let AWS Glue
                     access the resources it needs.
    :param db_name: The name to give the database that is created by the crawler.
    :param db_prefix: The prefix to give any database tables that are created by
                      the crawler.
    :param s3_target: The URL to an S3 bucket that contains data that is
                      the target of the crawler.
    """
    try:
        self.glue_client.create_crawler(
            Name=name,
            Role=role_arn,
            DatabaseName=db_name,
            TablePrefix=db_prefix,
            Targets={"S3Targets": [{"Path": s3_target}]},
        )
    except ClientError as err:
        logger.error(
            "Couldn't create crawler. Here's why: %s: %s",
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

```
def start_crawler(self, name):
    """
    Starts a crawler. The crawler crawls its configured target and creates
    metadata that describes the data it finds in the target data source.
    :param name: The name of the crawler to start.
    """
    try:
        self.glue_client.start_crawler(Name=name)
    except ClientError as err:
        logger.error(
            "Couldn't start crawler %s. Here's why: %s: %s",
            name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
def get_database(self, name):
    """
    Gets information about a database in your Data Catalog.
    :param name: The name of the database to look up.
    :return: Information about the database.
    """
    try:
        response = self.glue_client.get_database(Name=name)
    except ClientError as err:
        logger.error(
            "Couldn't get database %s. Here's why: %s: %s",
            name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
```

```
def get_tables(self, db_name):
    """
    Gets a list of tables in a Data Catalog database.
    :param db_name: The name of the database to query.
    :return: The list of tables in the database.
    """
    try:
        response = self.glue_client.get_tables(DatabaseName=db_name)
    except ClientError as err:
        logger.error(
            "Couldn't get tables %s. Here's why: %s: %s",
            db_name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
def create_job(self, name, description, role_arn, script_location):
    """
    Creates a job definition for an extract, transform, and load (ETL) job that can
    be run by AWS Glue.
    :param name: The name of the job definition.
    :param description: The description of the job definition.
    :param role_arn: The ARN of an IAM role that grants AWS Glue the permissions
                     it requires to run the job.
    :param script_location: The Amazon S3 URL of a Python ETL script that is run as
                            part of the job. The script defines how the data is
                            transformed.
    """
    try:
        self.glue_client.create_job(
            Name=name,
            Description=description,
```

```
            Role=role_arn,
            Command={
                "Name": "glueetl",
                "ScriptLocation": script_location,
                "PythonVersion": "3",
            },
            GlueVersion="3.0",
        )
    except ClientError as err:
        logger.error(
            "Couldn't create job %s. Here's why: %s: %s",
            name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
def start_job_run(self, name, input_database, input_table, output_bucket_name):
    """
    Starts a job run. A job run extracts data from the source, transforms it,
    and loads it to the output bucket.
    :param name: The name of the job definition.
    :param input_database: The name of the metadata database that contains tables
                           that describe the source data. This is typically created
                           by a crawler.
    :param input_table: The name of the table in the metadata database that
                        describes the source data.
    :param output_bucket_name: The S3 bucket where the output is written.
    :return: The ID of the job run.
    """
    try:
        # The custom Arguments that are passed to this function are used by the
        # Python ETL script to determine the location of input and output data.
```

```
            response = self.glue_client.start_job_run(
                JobName=name,
                Arguments={
                    "--input_database": input_database,
                    "--input_table": input_table,
                    "--output_bucket_url": f"s3://{output_bucket_name}/",
                },
            )
        except ClientError as err:
            logger.error(
                "Couldn't start job run %s. Here's why: %s: %s",
                name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    def list_jobs(self):
        """
        Lists the names of job definitions in your account.
        :return: The list of job definition names.
        """
        try:
            response = self.glue_client.list_jobs()
        except ClientError as err:
            logger.error(
                "Couldn't list jobs. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["JobNames"]
```

```
def get_job_runs(self, job_name):
    """
    Gets information about runs that have been performed for a specific job
    definition.
    :param job_name: The name of the job definition to look up.
    :return: The list of job runs.
    """
    try:
        response = self.glue_client.get_job_runs(JobName=job_name)
    except ClientError as err:
        logger.error(
            "Couldn't get job runs for %s. Here's why: %s: %s",
            job_name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["JobRuns"]
def get_job_run(self, name, run_id):
    """
    Gets information about a single job run.
    :param name: The name of the job definition for the run.
    :param run_id: The ID of the run.
    :return: Information about the run.
    """
    try:
        response = self.glue_client.get_job_run(JobName=name, RunId=run_id)
    except ClientError as err:
        logger.error(
            "Couldn't get job run %s/%s. Here's why: %s: %s",
            name,
```

```
            run_id,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["JobRun"]
def delete_job(self, job_name):
    """
    Deletes a job definition. This also deletes data about all runs that are
    associated with this job definition.
    :param job_name: The name of the job definition to delete.
    """
    try:
        self.glue_client.delete_job(JobName=job_name)
    except ClientError as err:
        logger.error(
            "Couldn't delete job %s. Here's why: %s: %s",
            job_name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
def delete_table(self, db_name, table_name):
    """
    Deletes a table from a metadata database.
    :param db_name: The name of the database that contains the table.
    :param table_name: The name of the table to delete.
    """
    try:
        self.glue_client.delete_table(DatabaseName=db_name, Name=table_name)
    except ClientError as err:
```

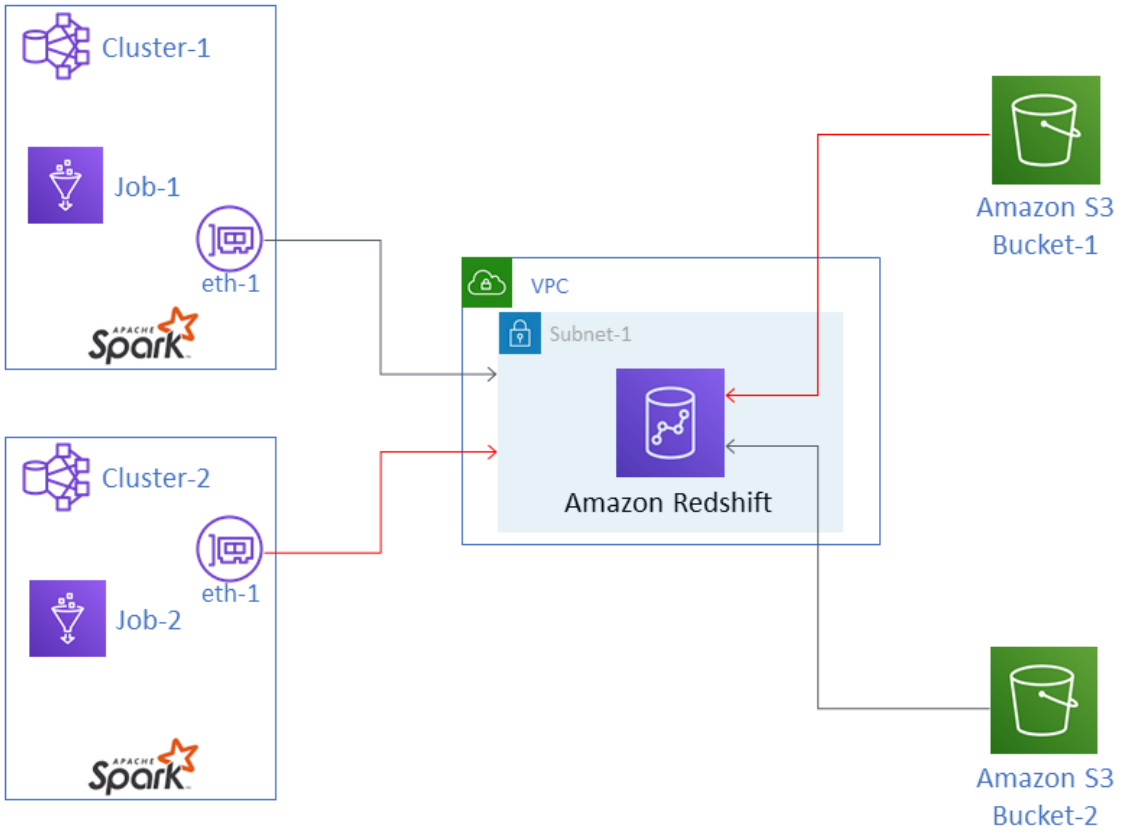
```
            logger.error(
                "Couldn't delete table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    def delete_database(self, name):
        """
        Deletes a metadata database from your Data Catalog.
        :param name: The name of the database to delete.
        """
        try:
            self.glue_client.delete_database(Name=name)
        except ClientError as err:
            logger.error(
                "Couldn't delete database %s. Here's why: %s: %s",
                name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    def delete_crawler(self, name):
        """
        Deletes a crawler.
        :param name: The name of the crawler to delete.
        """
        try:
            self.glue_client.delete_crawler(Name=name)
        except ClientError as err:
            logger.error(
                "Couldn't delete crawler %s. Here's why: %s: %s",
```

```
        name,
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise
```

31. Preventing cross-job data access

Consider the situation where you have two AWS Glue Spark jobs in a single AWS Account, each running in a separate AWS Glue Spark cluster. The jobs are using AWS Glue connections to access resources in the same virtual private cloud (VPC). In this situation, a job running in one cluster might be able to access the data from the job running in the other cluster.

The following diagram illustrates an example of this situation.



In the diagram, AWS Glue `Job-1` is running in `Cluster-1`, and Job-2 is running in `Cluster-2`. Both jobs are working with the same instance of Amazon Redshift, which resides in `Subnet-1` of a VPC. `Subnet-1` could be a public or private subnet.

`Job-1` is transforming data from Amazon Simple Storage Service (Amazon S3) `Bucket-1` and writing the data to Amazon Redshift. `Job-2` is doing the same with data in `Bucket-2`. `Job-1` uses the AWS Identity and Access Management (IAM) role `Role-1` (not shown), which gives access to `Bucket-1`. `Job-2` uses `Role-2` (not shown), which gives access to `Bucket-2`.

These jobs have network paths that enable them to communicate with each other's clusters and thus access each other's data. For example, `Job-2` could access data in `Bucket-1`. In the diagram, this is shown as the path in red.

To prevent this situation, we recommend that you attach different security configurations to `Job-1` and `Job-2`. By attaching the security configurations, cross-job access to data is blocked by virtue of certificates that AWS Glue creates. The security configurations can be *dummy* configurations. That is, you can create the security configurations without enabling encryption of Amazon S3 data, Amazon CloudWatch data, or job bookmarks. All three encryption options can be disabled.

To attach a security configuration to a job

1. Open the AWS Glue console at https://console.aws.amazon.com/glue/.
2. On the **Configure the job properties** page for the job, expand the **Security configuration, script libraries, and job parameters** section.
3. Select a security configuration in the list.