AWS Whitepaper

AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline

AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline: AWS Whitepaper

Table of Contents

AWS Glue Best Practices: Building a Secure and Reliable Data Pipeline

Publication date: **August 26, 2022** (*Document revisions*)

Abstract

Data integration is a critical element in building a data lake and a data warehouse. Data integration enables data from different sources to be cleaned, harmonized, transformed, and finally loaded. When building a data warehouse, the bulk of development efforts are needed for building a data integration pipeline. Data integration is one of the most critical pillars in data analytics ecosystems. An efficient and well-designed data integration pipeline is critical for making the data available, and being trusted among the analytics consumers.

In this whitepaper, we show you some of the consideration and best practices for security and reliability of data pipelines built with AWS Glue.

To get the most out of reading this whitepaper, it helps to be familiar with AWS Glue, AWS Glue DataBrew, Amazon Simple Storage Service (Amazon S3), AWS Lambda, and AWS Step Functions.

- For best practices around Operational Excellence for your data pipelines, refer AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline.
- For best practices around Performance Efficiency and Cost Optimization for your data pipelines, refer AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Introduction

Data volumes and complexities are increasing at an unprecedented rate, exploding from terabytes to petabytes or even exabytes of data. Traditional on-premises based approaches for bundling a data pipeline do not work well with a cloud-based strategy, and most of the time, do not provide the elasticity and cost effectiveness of cloud native approaches.

AWS hears from customers that they want to extract more value from their data, but struggle to capture, store, and analyze all the data generated by today's modern and digital businesses. Data is growing exponentially, coming from new sources. It is increasingly diverse, and needs to be securely accessed and analyzed by any number of applications and people.

With changing data and business needs, the focus on building a high performing, cost effective, and low maintenance data pipeline is paramount. Introduced in 2017, AWS Glue is a fully managed, serverless data integration service that allows customers to scale based on their workload, with no infrastructures to manage.

The next section discusses common best practices for building and efficiently operating your data pipeline with AWS Glue. This document is intended for advanced users, data engineers and architects.

You may want to refer to AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline to understand more about the AWS Glue product family before proceeding to the next sections.

Using the AWS Well-Architected framework for building a data pipeline

Building a well-architected data pipeline is critical for the success of a data engineering project. When designing a well-architected data pipeline, use the guidelines of the AWS Well-Architected Framework. This helps you understand the pros and cons of decisions you make while building applications on AWS.

The Well-Architected Framework guides the architecture considerations in operating reliable, secure, efficient, and cost-effective systems in the cloud. It provides a way for you to consistently measure your architectures against best practices, and identify areas for improvement. AWS believes that having a well-architected data pipeline using the AWS Well-Architected pillars greatly increases the likelihood of success. The AWS Well-Architected Framework is based on six pillars:

- **Operational Excellence** — The Operational Excellence pillar includes the ability to support development and run workloads effectively, gain insight into their operations, and to continuously improve supporting processes and procedures to deliver business value. You can find prescriptive guidance on implementation in the *Operational Excellence Pillar* whitepaper.
- **Security** — The Security pillar encompasses the ability to protect data, systems, and assets to take advantage of cloud technologies to improve your security. You can find prescriptive guidance on implementation in the *Security Pillar* whitepaper.
- **Reliability** — The Reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test the workload through its total lifecycle. You can find prescriptive guidance on implementation in the *Reliability Pillar* whitepaper.
- **Performance Efficiency** —The Performance Efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve. You can find prescriptive guidance on implementation in the *Performance Efficiency Pillar* whitepaper.
- **Cost Optimization** — The Cost Optimization pillar includes the ability to run systems to deliver business value at the lowest price point. You can find prescriptive guidance on implementation in the *Cost Optimization Pillar* whitepaper.
- **Sustainability** — The Sustainability pillar focuses on environmental impacts, especially energy consumption and efficiency, since they are important levers for architects to inform direct

action to reduce resource usage. You can find prescriptive guidance on implementation in the *Sustainability Pillar* whitepaper.

This whitepaper covers best practices around Security and Reliability of data.

- For best practices around Operational Excellence for your data pipelines, refer AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline.
- For best practices around Performance Efficiency and Cost Optimization for your data pipelines, refer AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline.

Building a reliable data pipeline

The Well-Architected Reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test the workload through its total lifecycle. This section goes in depth on best practice guidance for implementing a reliable data pipeline on AWS Glue.

Error handling

Error handling is the response and recovery mechanism from errors in an application. Application developers should follow standard error handling practices according to the language (Scala/Python) they are using to code their AWS Glue jobs.

This AWS Glue developer guide lists some of the common exceptions – causes and solutions that can be used to resolve AWS Glue-related errors.

When setting up AWS Glue jobs, it is recommended that the application team use Amazon CloudWatch to log and monitor the application. AWS Glue also supports near real-time continuous logging for jobs.

There are options to set standard filters into these logs so that standard Spark and Hadoop-related log messages are pruned out from the application log. Apart from filters, there is also an option to use a custom script logger to log application-specific messages. For more details on how to set this up, refer to Enabling Continuous Logging for AWS Glue Jobs. AWS Glue jobs can also be configured to send the Spark UI logs to an Amazon S3 bucket, that can be visualized on the Spark UI. Refer to Monitoring Jobs Using the Apache Spark Web UI for instructions on how to monitor AWS Glue jobs and configure the Spark UI.

AWS Glue also provides a feature called job run insights. This feature simplifies job debugging and optimization for your AWS Glue jobs and, provides details on your job such as line number of errors in the application, last run Spark action (before failure), root cause of failure and recommended action, and so on. The feature also allows you to supply a custom rule file to customize the error messages and recommendations.

The AWS Glue job run insights feature is enabled by default when building jobs using AWS Glue Studio. Alternatively, it can be enabled setting the `--enable-job-insights` flag to true. Job run insights works with AWS Glue jobs version 2.0 and above.

Enforcing schema

Sometimes the format of the incoming files or dataset may have variations from the existing data schema, and may result in the failure of downstream jobs and workflows. Applying schema can help you conform the data to the schema using the schema validation mechanism, as explained next.

In AWS Glue, `DynamicFrame` infers the schema of datasets by default. You can use the `with_frame_schema()` to supply schema to `DynamicFrame`. This is a performance optimization that excepts `DynamicFrame` from having to do any schema inference.

However, in special cases where you need to enforce a schema or do some casting or mappings such as reading numeric columns as string, the recommended approach is to use the `withSchema` format option.

Consider a case where one of the columns in your dataset is numeric in nature. For example, zip codes that may have a leading zero. The leading zero gets dropped when read as a number. To preserve the leading zero, you can force the column to string.

Following is an example code snippet that shows how to pass the `withSchema` format option.

1. Create the schema that matches your data.

```
from pyspark.sql.types import *
schema = StructType([
Field("first_name", StringType())
,Field("last_name", StringType())
,Field("email", StringType())
,Field("code", StringType()),
])
```

2. Pass the JSON value of the previous schema to `DynamicFrame`. The `withSchema` accepts a JSON string equivalent of the schema.

```
read_datasource_s3_with_schema =
glueContext.create_dynamic_frame_from_options(
connection_type = "s3",
connection_options = {"paths": input_datasource_s3}, format="xml",
format_options= {"rowTag":"record","withSchema":json.dumps(schema.jsonValue())},
transformation_ctx ="read_datasource_s3_with_schema")
```

In the previous step, we have passed the JSON string equivalent of the schema using `json.dumps(schema.jsonValue())`. Alternatively, when creating the schema, you can always create the JSON string equivalent.

To test this, we can use the following XML data set:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
<record>
<first_name>Clayson</first_name>
<last_name>Stollsteiner</last_name>
<email>cstollsteiner@example.com</email>
<code>1</code>
</record>
<record>
<first_name>Arnoldo</first_name>
<last_name>Presdie</last_name>
<email>apresdie@example.com</email>
<code>2</code>
</record>
<record>
<first_name>Allie</first_name>
<last_name>Asling</last_name>
<email>aasling@example.com</email>
<code>3</code>
</record>
</dataset>
```

Alternatively, you can supply the JSON string of schema, as follows:

```
jsonStringSchema="{ \"dataType\": \"struct\", \"fields\": [{
"name\": \"first_name\", \"container\": { \"dataType\": \"string\"
} }, { \"name\": \"last_name\", \"container\": { \"dataType\":
\"string\" } }, { \"name\": \"email\", \"container\": {
\"dataType\": \"string\" } }, { \"name\": \"code\", \"container\":
{ \"dataType\": \"string\" } }] }"

read_datasource_s3_with_schema =
glueContext.create_dynamic_frame_from_options(
connection_type = "s3",
connection_options = {"paths": input_datasource_s3}, format="xml",
format_options= {"rowTag":"record","withSchema":jsonStringSchema},
```

```
transformation_ctx ="read_datasource_s3_with_schema")
```

Sharing state using AWS Glue workflow properties

When you are building complex data pipelines where one job depends on another job, it's important to share the state information between different AWS Glue jobs. This section describes how to share state between chained jobs in an AWS Glue workflow.

There can be several reasons to share states between AWS Glue jobs in a workflow. A simple example is a job that needs a computed value from a predecessor job. This can be achieved in an AWS Glue workflow.

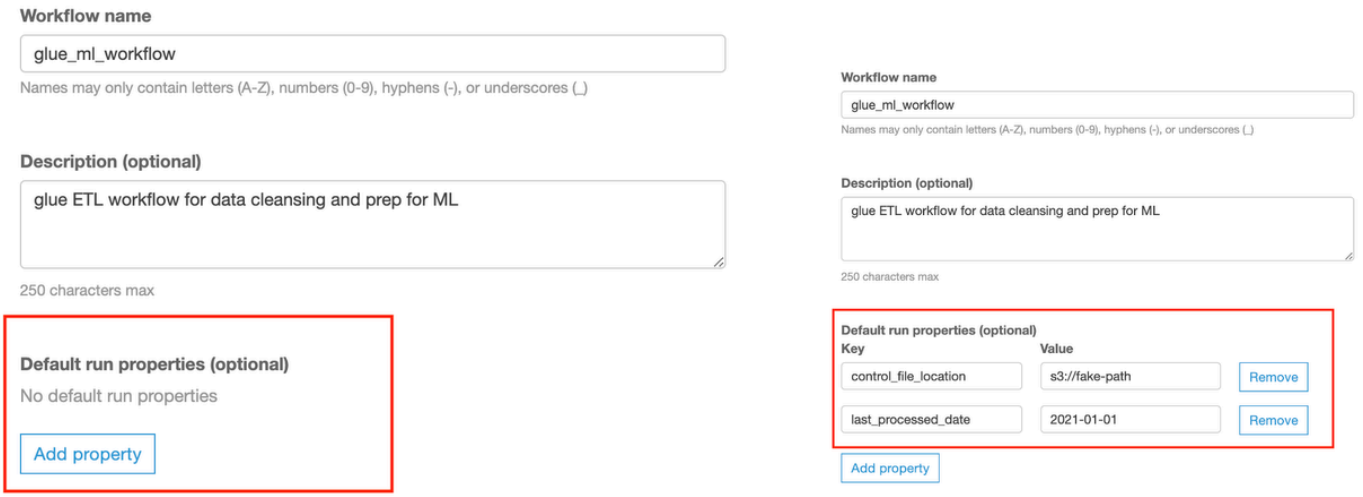
Workflows in AWS Glue support key-value pairs that are called run properties. When creating a workflow, you can create these workflow properties and initialize them with default values. As the job runs, you can overwrite these default values with computed values that other jobs can read.

Creating workflow properties

This section assumes that you are already familiar with the creation of an AWS Glue workflow. If you are not, refer to Creating and Building Out a Workflow Manually in AWS Glue.

Setting the properties using the AWS Glue console

If you are creating a workflow via the AWS Glue console, scroll down to the **Default run properties** section, choose **Add property,** and add your properties and default values in the prompted key value fields.



Creating an AWS Glue workflow property

Setting the properties using the AWS Command Line Interface (AWS CLI)

If you are creating the workflow using the AWS CLI, pass the key-value pair using the flag `--default-run-properties`.

The flag accepts a map of key values that act as the default job run properties. You can use the shorthand syntax or JSON as input:

```
--default-run-properties {"key1":"value1","key2":"value2"}
or
--default-run-properties key1=value1,key2=value2
```

For additional details on the AWS CLI for AWS Glue workflow creation, refer to create-workflow.

Saving state from an AWS Glue job

Now that the workflow run properties are defined, you can access these properties from within an AWS Glue job and set the values that other jobs can read.

```
import sys
import boto3
from awsglue.utils import getResolvedOptions

glue_client = boto3.client("glue")

#access the workflow name and run id
args = getResolvedOptions(sys.argv, ['JOB_NAME','WORKFLOW_NAME', 'WORKFLOW_RUN_ID'])
workflow_name = args['WORKFLOW_NAME']
workflow_run_id = args['WORKFLOW_RUN_ID']
'''
the same glue job could be a part of multiple workflows
so it is good to access the workflow name and run_id to avoid undesired results
'''
run_properties={}
run_properties["control_file_path"]="fake path"
run_properties["last_processed_date"]="fake date"
glue_client.put_workflow_run_properties(
Name=workflow_name,
RunId=workflow_run_id,
```

```
RunProperties=run_properties
)
```

Accessing state from an AWS Glue job

You can read the properties from within an AWS Glue job:

```
#minimum required imports
import sys
import boto3
from awsglue.utils import getResolvedOptions

glue_client = boto3.client("glue")

#access the workflow name and run id
args = getResolvedOptions(sys.argv, ['JOB_NAME','WORKFLOW_NAME', 'WORKFLOW_RUN_ID'])
workflow_name = args['WORKFLOW_NAME']
workflow_run_id = args['WORKFLOW_RUN_ID']
'''
the same glue job could be a part of multiple workflows
so it is good to access the workflow name and run_id to avoid undesired results
'''
run_properties = glue_client.get_workflow_run_properties(
Name=workflow_name,
RunId=workflow_run_id
)["RunProperties"]
# let us access 2 keys - control_file_path & last_processed_date
control_file_path=run_properties["control_file_path"]
last_processed_date=run_properties["last_processed_date"]
```

Glue streaming ETL for near real-time jobs

A streaming extract, transform, load (ETL) job in AWS Glue is based on Apache Spark's Structured Streaming engine, which provides a fault-tolerant, scalable, and easy way to achieve end-to-end stream processing. Streaming job running in AWS Glue inherit most or all of its best practices from an equivalent Apache Spark structured streaming job.

Jobs and scalability

Streaming jobs in AWS Glue can consume data from streaming sources such as Amazon Kinesis Data Streams, Apache Kafka, and Amazon Managed Streaming for Apache Kafka (Amazon MSK).

Each shard in a Kinesis data stream or each partition in a Kafka topic is treated as an input partition in Spark. For each partition, Spark generates one or more tasks and runs them on available workers.

In Spark, a *task* is the smallest individual unit of job run. Each Spark task by default maps to a single core and works on a single Spark DataFrame partition of data. If there are equal number of workers available as number of partitions, Spark runs each task on a separate worker. Otherwise, it runs more than one task on a single worker (or data processing unit (DPU)).

For example, G.1X instances can support one Spark executor with four cores. Assuming there is enough capacity, the worker can have four tasks and can process four partitions in parallel. Increasing the Kafka topic partitions (on Kinesis shards), in general, is a good recommendation to improve AWS Glue streaming ETL job performance. Having more partitions (preferably multiples of cores) than cores avoids having idle executors, and is a good way to scale up.

However, increasing the partitions or shards may introduce more management and operational overheads. You should follow the Kafka/Kinesis recommendations on correctly sizing the partitions or shards, based on your read/write throughput requirements. Alternatively, use the following two options to increase parallelism.

- **Programmatically increase spark partitions** — Spark API does support the `repartition(numPartitions)` method. This method is used to increase or decrease the number of partitions of a Spark data frame. To see the current number of partitions on a data frame, you can use the `rdd.getNumPartitions()` method. You can pass a bigger number of partitions as argument to the repartition method on the micro-batch data frame. It will create partitions of more or less equal size by performing a full shuffle of data across all the nodes. It is recommended to try this option on your actual workload to ensure that the overhead (of shuffle) is small and the benefits are higher.
- **Add more workers** — You can configure AWS Glue ETL jobs with a higher number of workers by setting the `NumberOfWorkers`. This is a good option if the current number of workers doesn't have enough capacity to allocate Spark tasks for all the partitions/shards.

In summary, for maximum parallelism, allocate the number of AWS Glue workers proportionate to the number of input partitions/shards. However, based on the workload, you can experiment with different workers (type and count) that suit your application's processing requirements, and pick the one that best fits your needs. At the time of this writing, AWS Glue is previewing autoscaling for AWS Glue streaming jobs. You can find more details on the feature in Using Auto Scaling for AWS Glue.

Job recovery

AWS Glue streaming is based on Apache Spark's Structured Streaming engine, which provides a fault-tolerant, scalable, and easy way to achieve end-to-end stream processing. Applications/jobs can use Spark's checkpoint mechanism for maintaining intermediate state and recovery from failure. AWS Glue streaming ETL jobs let you configure an S3 location for storing the checkpoint data. Unlike batch ETL, streaming ETL doesn't depend on bookmarks. Therefore, for reprocessing a dataset, you will need to delete the checkpoint directory. Note that if the application is configured to read the latest records from source, deleting the checkpoint will still fetch only the latest records, and not the previously processed records.

Schema handling

AWS Glue streaming jobs can infer the schema of incoming data streams. However, if you have a fixed schema data and you are already aware of the data structure and type, predefining the schema in a catalog table or AWS Glue schema registry is recommended. This is because automatic schema detection restricts the ability to perform operations such as joins on the stream.

AWS Glue DataBrew for data quality

Organizations are generating more and more data these days, and it is increasingly important for them to maintain the quality of data to make it usable by downstream applications and analytics services to make important business decisions. Data comes from many different data streams or applications that may develop slight changes in data schema and data profile over a period of time. Data is then stored in a data lake for further consumption. If the quality of data is not maintained and data is not properly curated, the data in the data lake may become unusable by consumers. The data lake may turned into a data swamp, resulting in frustration among consumers. Continuously monitoring data quality to filter out bad data by comparing it with predefined target metrics helps you keep data clean and usable.

AWS Glue DataBrew is a new visual data preparation tool that makes it easy for data analysts and data scientists to clean and normalize data to prepare it for analytics and machine learning (ML). You can choose from over 250 pre-built transformations to automate data preparation tasks, all without needing to write any code.

You can also use AWS Glue DataBrew to evaluate the quality of your data by profiling it to understand data patterns, and detect anomalies by connecting directly to your data lake, data warehouses, and databases. To ensure data is always of high quality, you need to consistently

profile new data, evaluate that it meets your business rules, and alert for problems in the data to fix any issues. Refer to the AWS blog post Setting up automated data quality workflows and alerts using AWS Glue DataBrew and AWS Lambda for guidance on using AWS Glue DataBrew to set up a recurring profile job to determine data quality metrics; and using your defined business rules to report on the validity of the data.

Memory management for large ETL workloads

AWS Glue uses Apache Spark in its core, which provides several approaches for memory management to optimize the use of memory for workloads with large data volume. This section shows how you can combine AWS Glue capabilities and Spark best practices for handling large jobs.

Optimization with Spark Driver

Push down predicates — AWS Glue jobs let you use pushdown predicates to prune the partitions not required from the table before data is read. This comes handy for tables with large numbers of partitions, of which the job requires only a subset for processing.

However, note that the `push_down_predicate` option is applied only after the partitions are listed, and it alone doesn't help with faster data fetch. You can use server-side partition pruning using `catalogPartitionPredicate`, which uses the partition index.

Following is an example usage of `push_down_predicate` alongside `catalogPartitionPredicate`.

```
dynamic_frame = glueContext.create_dynamic_frame.from_catalog(
    database=dbname,
    table_name=tablename,
    transformation_ctx="tx_context_0",
    push_down_predicate="col_1>=100 and col_2 <=10",
    additional_options={"catalogPartitionPredicate":"part_1='US' and part_2='2022'"}
)
```

AWS Glue S3 Lister — AWS Glue provides an optimized mechanism to list files on S3 while reading data into a DynamicFrame. The AWS Glue S3 Lister can be enabled by setting the DynamicFrame's `additional_options` parameter `useS3ListImplementation` to `True`. The AWS Glue S3 Lister offers advantage over the default S3 list implementation by strictly iterating over the final list of filtered files to be read.

```
datasource = glue_context.create_dynamic_frame.from_catalog(
```

```
database = "tpc",
table_name = "dl_web_sales",
push_down_predicate = partitionPredicate,
additional_options = {"useS3ListImplementation":True}
)
```

Grouping — AWS Glue allows you to consolidate multiple files per Spark task using the file grouping feature. Grouping files together reduces the memory footprint on the Spark driver, as well as simplifying file split orchestration. Without grouping, a Spark application must process each file using a different Spark task. Each task must then send a `mapStatus` object containing the location information to the Spark driver. In our testing using the AWS Glue standard worker type, we found that Spark applications processing more than roughly 650,000 files often cause the Spark driver to crash with an "out of memory: exception as shown in the following error message:

```
# java.lang.OutOfMemoryError: Java heap space
# -XX:OnOutOfMemoryError="kill -9 %p"
# Executing /bin/sh -c "kill -9 12039"...
```

`groupFiles` allows you to group files within a Hive-style S3 partition (`inPartition`) and across S3 partitions (`acrossPartition`). `groupSize` is an optional field that allows you to configure the amount of data to be read from each file and processed by individual Spark tasks.

```
dyf = glueContext.create_dynamic_frame_from_options("s3",{'paths':
 ["s3://input-s3-path/"],'recurse':True,'groupFiles':
'inPartition','groupSize': '1048576'},format="json")
```

Exclusions for S3 storage classes — AWS Glue offers the ability to exclude objects based on their underlying S3 storage class. As the lifecycle of data evolves, hot data becomes cold, and automatically moves to lower-cost storage based on the configured S3 bucket policy, it's important to make sure ETL jobs process the correct data. This is particularly useful when working with large datasets that span across multiple S3 storage classes using the Apache Parquet file format, where Spark will try to read the schema from the file footers in these storage classes.

Amazon S3 offers a range of storage classes:

- **S3 Standard** for frequently accessed data
- **S3 Standard-Infrequent Access (S3 Standard-IA)** and **S3 One Zone-Infrequent Access (S3 One Zone-IA)** for less frequently accessed data
- **S3 Glacier Instant Retrieval** for archive data that needs immediate access

- **S3 Glacier Flexible Retrieval (formerly S3 Glacier)** for rarely accessed long-term data that does not require immediate access
- **Amazon S3 Glacier Deep Archive (S3 Glacier Deep Archive)** for long-term archive and digital preservation with retrieval in hours at the lowest cost storage in the cloud

When reading data using `DynamicFrames`, you can specify a list of S3 storage classes you want to exclude. This feature uses the optimized AWS Glue S3 Lister. The following example shows how to exclude files stored in `GLACIER` and `DEEP_ARCHIVE` storage classes.

```
glueContext.create_dynamic_frame.from_catalog(
database = "my_database",
tableName = "my_table_name",
redshift_tmp_dir = "",
transformation_ctx = "my_transformation_context",
additional_options = {
"excludeStorageClasses" : ["GLACIER", "DEEP_ARCHIVE"]
)
)
```

`GLACIER` and `DEEP_ARCHIVE` storage classes allow only listing files and require an asynchronous S3 restore process to read the actual data. The following is the exception you will see when you try to access Glacier and Deep Archive storage classes from your AWS Glue ETL job:

```
java.io.IOException:
com.amazon.ws.emr.hadoop.fs.shaded.com.amazonaws.services.s3.model.
AmazonS3Exception:
The operation is not valid for the object's storage class (Service:
Amazon S3; Status Code: 403;
Error Code: InvalidObjectState; Request ID: ), S3 Extended Request
ID: (1)
```

Optimize Spark queries

Inefficient queries or transformations can have a significant impact on Apache Spark driver memory utilization. Common examples include:

- `Collect` is a Spark action that collects the results from workers and returns them back to the driver. In most cases, the results may be very large, overwhelming the driver. It is recommended to be careful while using `collect`, because it can frequently cause Spark driver out of memory

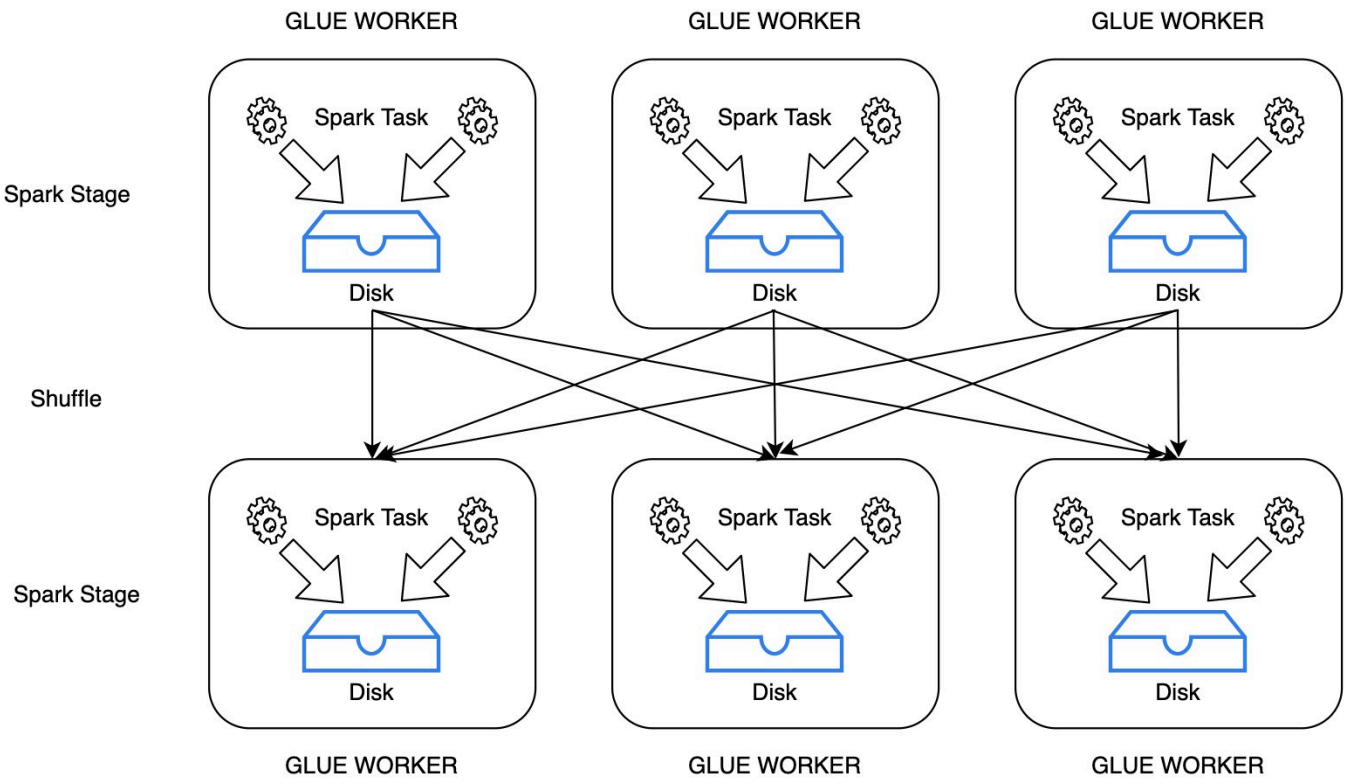
(OOM) exceptions. To preview the data from a previous transformation, you can rely on the `take` or `takeSample` actions.

- **Shared variables** —Apache Spark offers two different ways to share variables between Spark driver and executors: broadcast variables and accumulators. Broadcast variables are useful to provide a read-only copy of data or fact tables shared across Spark workers to improve map-side joins. Accumulators are useful to provide a writable copy to implement distributed counters across Spark executors. Both should be used carefully, and destroyed when no longer needed, as they can frequently result in Spark driver OOM exceptions.

Manage shuffle disk capacity and memory constraint

In Apache Spark, *Shuffle* is a mechanism for redistributing data so that it's grouped differently across partitions. Some of the transformations or operations which can cause a shuffle include `repartition`, `coalesce`, `distinct`, and `ByKey` (except for counting), `groupByKey` and `reduceByKey`, and `join` operations such as `cogroup` and `join`.

The Shuffle operation is triggered when the wide transformation needs information from other partitions to complete its processing. Spark gathers the required data from each partition and combines it into a new partition. During a Shuffle phase, when data does not fit in memory, the Spark map tasks write shuffle data to a local disk that is transferred across the network and fetched by Spark reduce tasks. With AWS Glue, workers write shuffle data on local disk volumes attached to the AWS Glue workers. The following diagram illustrates how the Spark map tasks write the shuffle and spill files to local disks.

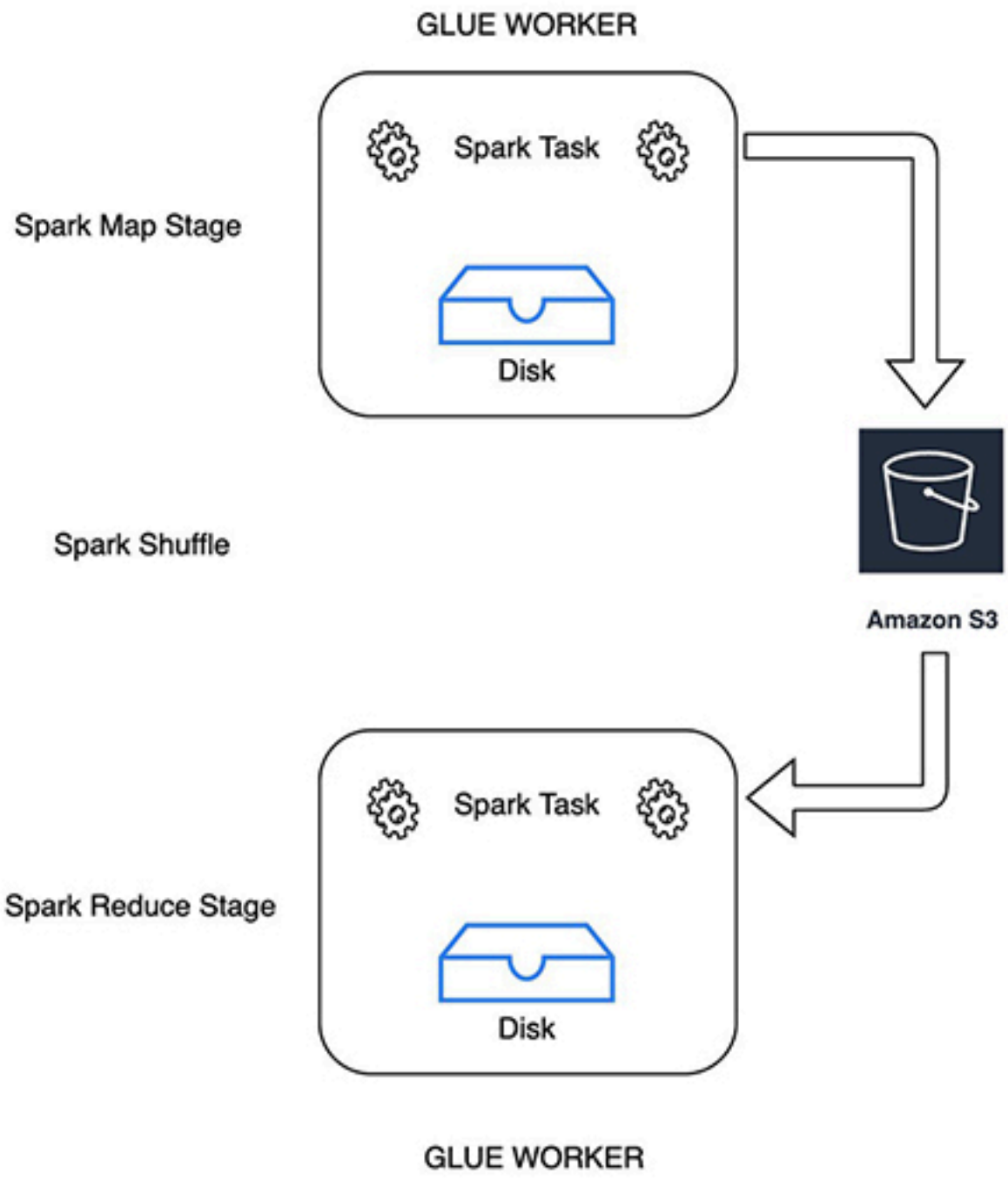


Shuffle operation with write to local disk

Challenges with the Shuffle operation — The process of shuffling data results in additional overhead of disk input/output (I/O), data serialization, network I/O, and increased garbage collection, making the Shuffle a complex and costly operation. It leads to the following challenges:

- **Hitting local storage limits** — Due to the overhead, the Shuffle operation is often constrained by the available local disk capacity, or data skew, which can cause straggling executors. Spark often throws a "No space left on device" or `MetadataFetchFailedException` error when there is not enough disk space left on the executor; and there is no recovery from the exception state. If you are using G.1X workers, a quick attempt to get around the issue would be to try G.2X worker which has twice the disk capacity.
- **Co-location of storage with executors** — If an executor is lost, then shuffle files are lost as well. This leads to several task and stage retries, as Spark tries to recompute stages in order to recover lost shuffle data. Spark natively provides an external shuffle service that lets it store shuffle data independent to the life of executors. But the shuffle service itself becomes a point of failure and must always be up in order to serve shuffle data. Additionally, shuffles are still stored on local disk, which might run out of space for a large job.

To overcome these challenges, a new Spark shuffle manager is available in AWS Glue that disaggregates Spark compute and shuffle storage by utilizing Amazon S3 to store Spark shuffle and spill files. Using Amazon S3 for Spark shuffle storage lets you run data-intensive workloads much more reliably, and scale elastically. The following figure illustrates how Spark map tasks write the shuffle and spill files to the given Amazon S3 shuffle bucket. Reducer tasks consider the shuffle blocks as remote blocks, and read them from the same shuffle bucket.



AWS Glue shuffle manager with write to S3

Enabling AWS Glue Spark shuffle manager – AWS Glue 2.0 allows users to configure the AWS Glue Spark shuffle manager. It can be enabled using the following job parameters.

Table 1 — Job parameters for AWS Glue shuffle manager

Key	Value	Explanation
`--write-shuffle-files-to-s3`	TRUE	This is the main flag, which tells Spark to use S3 buckets for writing and reading shuffle data.
`--write-shuffle-spills-to-s3`	TRUE	This is an optional flag that lets you offload spill files to S3 buckets, which provides additional resiliency to your Spark job. This is only required for large workloads that spill a lot of data to disk. This flag is disabled by default.
`--conf`	`spark.shuffle.glue.s3ShuffleBucket=s3://<shuffle-bucket>`	This is also optional, and it specifies the S3 bucket where we write the shuffle files.

Consider this solution in following situations:

- This feature is recommended when you want to ensure the reliable run of your data intensive workloads that create a large amount of shuffle or spill data. Writing and reading shuffle files from Amazon S3 is marginally slower when compared to local disk for our experiments with TPC-DS queries. S3 shuffle performance is impacted by the number and size of shuffle files. For example, S3 can be slower for reads as compared to local storage if you have a large number of small shuffle files or partitions in your Spark application.
- You can use this feature if your job frequently suffers from “No space left on device” issues.
- You can use this feature if your job frequently suffers fetch failure issues:

```
org.apache.spark.shuffle.MetadataFetchFailedException
```

- You can use this feature if your data is skewed.

- We recommend setting the S3 bucket lifecycle policies on the shuffle bucket (`spark.shuffle.glue.s3ShuffleBucket`) to clean up old shuffle data.
- This feature is available on AWS Glue 2.0 and Spark 2.4.3.

Building a secure data pipeline

The Well-Architected Security pillar focuses on protecting data and information. It describes how to take advantage of cloud technologies to protect data and information in a way that can improve your security posture. Following are some of the best practices to consider to improve the security of data and information with your data pipeline in AWS Glue.

Data encryption

Encryption of data at rest

AWS Glue supports encryption for authoring jobs in AWS Glue, and developing scripts using development endpoints. Encryption configurations can be provided by attaching a security configuration. Security configurations contain Amazon S3-managed server-side encryption keys (SSE-S3) or customer managed keys (CMKs) stored in AWS Key Management Services (AWS KMS) (SSE-KMS). It is also worth noting that AWS Glue, as of writing this document, supports only symmetric CMKs.

You can encrypt the metadata stored in the AWS Glue Data Catalog using AWS KMS. Additionally, you can use AWS KMS keys to encrypt job bookmarks, and the logs generated by AWS Glue crawlers and ETL jobs.

In AWS Glue, you control encryption settings in the following places:

- The settings of your Data Catalog.
- The security configurations that you create.
- The server-side encryption setting (SSE-S3 or SSE-KMS) that is passed as a parameter to your AWS Glue ETL job.

Encrypting your Data Catalog

You can turn on encryption of your AWS AWS Glue Data Catalog objects. You can turn on or turn off encryption settings for the entire Data Catalog. In the process, you specify an AWS KMS key that is automatically used when objects, such as tables, are written to the Data Catalog. The encrypted objects include the following:

- Databases

- Tables
- Partitions
- Table versions
- Connections
- User-defined functions

AWS AWS Glue Data Catalog also allows you to encrypt connection specific passwords using KMS keys.

Encryption of data in transit

AWS provides Transport Layer Security (TLS) encryption for data in motion between AWS Glue and S3.

An AWS Glue connection allows you to configure TLS certificates for database configurations as well.

Network security

AWS AWS Glue Data Catalog and AWS Glue ETL are serverless services, and can be accessed outside of VPCs by default using AWS Glue APIs. AWS Glue provides three AWS Identity and Access Management (AWS IAM) condition keys `glue:VpcIds`, `glue:SubnetIds`, and `glue:SecurityGroupIds`. You can use the condition keys in IAM policies when granting permissions to create and update jobs. You can use this setting to ensure that jobs are not created (or updated to) to run outside of a desired VPC environment.

To support certain use cases, AWS Glue ETL jobs may need to connect to services running inside a VPC - for example, a database running inside a private subnet. To support these use cases, AWS Glue provides an AWS Glue network connection feature. You can configure an AWS Glue network connection based on a VPC, VPC subnet, and security groups and attach it to an AWS Glue job. When the AWS Glue job runs, it creates an Elastic Network Interface (ENI) using the configuration defined in the network connection, and uses that ENI to access resources running within the VPC, making the connection secure and private without routing traffic through public networks.

Applications and services running inside a VPC without NAT Gateway or Internet Gateway cannot connect to AWS Glue out of the box, because there is no network path to AWS Glue API or AWS Glue services. To allow applications running in such environments to access AWS Glue, you can

create and attach AWS Glue Virtual Private Endpoints (VPCE) to the VPC subnets and route AWS Glue-specific connections through VPCE.

Managing Identity and Access in AWS Glue

Authentication

AWS provides two primary permission mechanisms to access AWS Glue: using an IAM user, or an IAM role.

- **IAM user** — A user is an identity within your AWS account that has specific custom permissions; for example, permissions to create a table in AWS Glue, or run an AWS Glue ETL job.
- **IAM role** — An IAM role is an IAM identity that you can create in your account that has specific permissions. An IAM role is similar to a user, however, instead of being uniquely associated with one person, a role is intended to be assumable by anyone who needs it. Also, a role does not have standard long-term credentials such as a password or access keys associated with it. Instead, when you assume a role, it provides you with temporary security credentials for your role session.

  IAM roles with temporary credentials are useful while using:

  - **Federated user access** from AWS Directory Service, your enterprise user directory, or a web identity provider.
  - **AWS service access** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. Example: When the AWS Glue service runs an AWS Glue ETL job on your behalf.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:

- Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
- (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Permission policy example:

The following policy grants all permissions on an AWS Glue table named books in database db1. This includes read and write permissions on the table itself, on archived versions of it, and on all its partitions. This policy can then be attached to a role to grant them these permissions.

```
{
    "Version": "2012-10-17",
    "Statement":  [
        {
            "Sid": "FullAccessOnTable",
            "Effect": "Allow",
            "Action":  [
                "glue:CreateTable",
                "glue:GetTable",
                "glue:GetTables",
                "glue:UpdateTable",
                "glue:DeleteTable",
                "glue:BatchDeleteTable",
                "glue:GetTableVersion",
                "glue:GetTableVersions",
                "glue:DeleteTableVersion",
                "glue:BatchDeleteTableVersion",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:BatchGetPartition",
                "glue:UpdatePartition",
                "glue:DeletePartition",
                "glue:BatchDeletePartition"
            ],
            "Resource":  [
                "arn:aws:glue:us-west-2:123456789012:catalog",
                "arn:aws:glue:us-west-2:123456789012:database/db1",
                "arn:aws:glue:us-west-2:123456789012:table/db1/books"
```

```
            ]
        }
    ]
}
```

AWS Lake Formation

AWS Lake Formation makes it easy to set up a secure data lake by providing easy grant/revoke access to the AWS Glue Data Catalog and the underlying S3 locations without the need of managing granular permissions in IAM.

In the above permission example, access to the database db1 and the books are granted through an IAM policy. This permission model works well if you have small number of databases and tables to share among a handful of users. However, many of our customers manage hundreds of databases and thousands of tables using the AWS Glue Data Catalog. The data for those tables span many S3 bucket locations which need to be shared securely among multiple teams and users. In these scenarios, IAM policy-based access control becomes complex to manage.

AWS Lake Formation was built to simplify the permission management for data lakes that manage a large data catalog.

When you enable Lake Formation for a Region, all permission management for the AWS Glue Data Catalog is automatically governed by Lake Formation going forward. Any legacy IAM based permissions for AWS Glue Data Catalog may still work if you have enabled backward compatibility through `IAMAllowedPrincipals`. You can start managing the Lake Formation permission through the AWS Management Console or though Lake Formation API.

Lake Formation access management involves following constructs:

- Principal – A Principal is any one of an IAM role, user, Security Assertion Markup Language (SAML) group, or SAML user.
- Resources – A resource is any of the following elements: data location, AWS Glue Data Catalog, databases, tables, columns, cells, and LF tags.

Lake Formation permissions can be managed through two access control models:

Resource Based Access Control (RBAC) — In the RBAC model, you can grant or revoke permissions to resources such as database, table, and column for a principal such as an IAM role or user. Based

on the resource type, the available permissions may vary. For a detailed definition of resources and permissions, refer to Lake Formation Permissions Reference.

Tag Based Access Control (TBAC) — In the TBAC model, you can create LF-tags which are key-value pairs (Example: classification=confidential, pii=true) and attach them to Resources and Principals. You can then assign and revoke permissions on resources using these LF-tags. Lake Formation allows operations on those resources when the principal's tag matches the resource tag. This model allows you to decouple permissions from resource creation which helps govern large number of databases, tables, and columns by removing the need to update permissions every time a new resource is added to the data lake. For detailed information about TBAC, refer to Overview of Lake Formation Tag-Based Access Control.

Conclusion

In this whitepaper, we explained some of the best practices around security and reliability when building your data pipeline in AWS Glue while considering the guidance contained in the AWS Well-Architectured framework.

Contributors

Contributors to this document include:

- ***Durga Mishra,*** Sr Solution Architect, Amazon Web Services
- ***Arun A K,*** Solution Architect, Amazon Web Services
- ***Narendra Gupta,*** Sr Solution Architect, Amazon Web Services
- ***Jay Palaniappan,*** Sr Solution Architect, Amazon Web Services
- ***Rajesh Agarwalla,*** Data Architect, Amazon Web Services

Further reading

For additional information, see:

- Snappy compressed format description
- Parquet compression definitions
- LanguageManual ORC
- Amazon Athena: Bucketing vs partitioning
- Top 10 Performance Tuning Tips for Amazon Athena
- AWS Glue pricing
- Load data incrementally and optimized Parquet writer with AWS Glue
- AWS Glue Exceptions
- Enabling Continuous Logging for AWS Glue Jobs
- Monitoring Jobs Using the Apache Spark Web UI
- Monitoring with AWS Glue job run insights
- *AWS Glue Best Practices: Building an Operationally Efficient Data Pipeline* (AWS whitepaper)
- *AWS Glue Best Practices: Building a Performant and Cost Optimized Data Pipeline* (AWS whitepaper)

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Minor revision	Updated guide to align with the IAM best practices . For more information, see Security best practices in IAM.	February 9, 2023
Initial publication	Whitepaper published.	August 26, 2022

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.