

AWS Glue vs Spark on EMR for UPSERT Operations

Technical Architecture Decision Document

Nathan Lunceford

2025-01-21

Table of contents

1	Summary	2
2	Understanding the Options	2
	AWS Glue	2
	Spark on EMR	3
3	Cost Analysis	3
3.1	AWS Glue Costs	3
3.1.1	Pricing Structure	3
3.1.2	Hidden Savings	3
3.1.3	Considerations	3
3.2	EMR Costs	3
3.2.1	Direct Costs	3
3.2.2	Optimization Options	4
3.2.3	Hidden Costs	4
4	Performance Comparison	4
4.1	AWS Glue Performance	4
4.2	EMR Performance	5
5	Development Experience	6
5.1	Key Differences	6
5.2	Integration Capabilities	6
5.2.1	AWS Glue	6
5.2.2	Spark on EMR	6

6	Monitoring and Operations	7
6.1	Monitoring Options	7
6.1.1	AWS Glue	7
6.1.2	Spark on EMR	7
6.2	Operational Requirements	7
7	Recommendation	7
7.1	Short Term	7
7.2	Long Term	8
8	Implementation Plan	8
8.1	Phase 1: Initial Setup	8
8.2	Phase 2: Production Migration	8
8.3	Phase 3: Optimization	8

1 Summary

This document compares two AWS-based Apache Spark solutions for handling UPSERT operations from S3 to multiple target databases including MSSQL and Apache Iceberg:

1. AWS Glue: A managed Apache Spark service
2. Apache Spark on Amazon EMR: A more configurable Spark deployment

Note

Both solutions use Apache Spark as their processing engine. The key differences lie in management, configuration, and operational aspects rather than core processing capabilities.

2 Understanding the Options

AWS Glue

AWS Glue provides a managed Apache Spark environment with:

- Built-in Apache Spark engine (same as EMR)
- AWS-specific optimizations and tooling
- Both Spark SQL and PySpark interfaces
- Additional features like DynamicFrames
- Managed infrastructure and scaling

Spark on EMR

Amazon EMR provides a more traditional Spark deployment with:

- Full Apache Spark ecosystem
- Complete configuration control
- Custom cluster management
- Direct access to Spark internals
- Infrastructure flexibility

3 Cost Analysis

3.1 AWS Glue Costs

3.1.1 Pricing Structure

- \$0.44 per DPU-Hour (1 DPU = 4 vCPU, 16GB memory)
- Minimum 10-minute billing
- Development endpoints additional cost

3.1.2 Hidden Savings

- No cluster management costs
- Includes Spark optimization
- Less operational overhead

3.1.3 Considerations

- More expensive per compute hour
- Less granular scaling
- Simplified cost model

3.2 EMR Costs

3.2.1 Direct Costs

- EC2 instance costs
- EMR service charges
- Storage and data transfer

3.2.2 Optimization Options

- Spot instance usage
- More granular scaling
- Resource optimization

3.2.3 Hidden Costs

- Operational overhead
- Management complexity
- Required expertise

4 Performance Comparison

4.1 AWS Glue Performance

```
# Example Glue Spark UPSERT implementation
from awsglue.context import GlueContext
from pyspark.context import SparkContext

# Initialize Glue Spark context
glueContext = GlueContext(SparkContext.getOrCreate())
spark = glueContext.spark_session

# Read from S3 (using standard Spark)
source_df = spark.read.parquet("s3://bucket/path")

# MSSQL UPSERT
def perform_mssql_upsert(df):
    # Write to staging table using Spark JDBC
    df.write \
        .format("jdbc") \
        .option("url", jdbc_url) \
        .option("dbtable", "staging_table") \
        .mode("overwrite") \
        .save()

    # Execute MERGE using Spark SQL
    spark.sql("""
MERGE INTO target_table t
```

```

USING staging_table s
ON t.key = s.key
WHEN MATCHED THEN UPDATE...
WHEN NOT MATCHED THEN INSERT...
""")

```

💡 Glue Performance Strengths

- Pre-configured Spark optimizations
- AWS service-specific tuning
- Auto-scaling built in
- Warm pools reduce startup time

⚠️ Glue Performance Limitations

- Less Spark configuration flexibility
- Fixed worker configurations
- Limited Spark version control

4.2 EMR Performance

```

# Example EMR Spark UPSERT implementation
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder \
    .appName("EMR UPSERT") \
    .getOrCreate()

# Read from S3 (identical to Glue)
source_df = spark.read.parquet("s3://bucket/path")

# MSSQL UPSERT (identical to Glue)
def perform_mssql_upsert(df):
    df.write \
        .format("jdbc") \
        .option("url", jdbc_url) \
        .option("dbtable", "staging_table") \
        .mode("overwrite") \
        .save()

```

💡 EMR Performance Strengths

- Full Spark configuration control
- Custom Spark properties
- Better performance for large jobs
- Fine-grained optimization

⚠️ EMR Performance Limitations

- Requires Spark expertise
- Infrastructure management overhead
- Cluster startup time

5 Development Experience

5.1 Key Differences

Aspect	AWS Glue	Spark on EMR
Setup Complexity	Low	High
Configuration Options	Limited	Extensive
Development Tools	AWS Console + IDE	Any IDE
Local Testing	Limited	Full Support
Debugging	Basic	Advanced

5.2 Integration Capabilities

5.2.1 AWS Glue

- Native AWS integration
- Pre-configured connectors
- Standard Spark JDBC
- Basic Iceberg support

5.2.2 Spark on EMR

- Full connector ecosystem
- Custom connectors
- All Spark data sources
- Complete Iceberg support

6 Monitoring and Operations

6.1 Monitoring Options

6.1.1 AWS Glue

- CloudWatch integration
- Built-in dashboards
- Auto-retry capability
- AWS-native alerting

6.1.2 Spark on EMR

- Full Spark metrics
- Custom monitoring
- Detailed job tracking
- Third-party tools

6.2 Operational Requirements

Requirement	AWS Glue	Spark on EMR
Spark Expertise	Basic	Advanced
DevOps Support	Minimal	Substantial
Maintenance	AWS Managed	Self Managed
Scaling	Automatic	Manual/Custom

7 Recommendation

7.1 Short Term

Recommend starting with AWS Glue due to:

- Faster implementation
- Managed environment
- Sufficient for current scale
- Lower operational overhead

7.2 Long Term

Consider migration to EMR if:

- Approaching cost crossover point
- Requiring more performance optimization
- Team has built Spark expertise
- Need more control over infrastructure

8 Implementation Plan

8.1 Phase 1: Initial Setup

1. Set up development environment
2. Create test jobs
3. Establish monitoring
4. Document procedures

8.2 Phase 2: Production Migration

1. Migrate simple jobs
2. Add error handling
3. Implement monitoring
4. Document operations

8.3 Phase 3: Optimization

1. Performance tuning
2. Cost optimization
3. Process refinement
4. Team training

Next Steps

1. Prototype both solutions
2. Test with production data volumes
3. Calculate actual costs