Migrating SSIS ETL jobs to AWS

# AWS Prescriptive Guidance

# AWS Prescriptive Guidance: Migrating SSIS ETL jobs to AWS

# Table of Contents

# Migrating SSIS ETL jobs to AWS

*Durga Prasad and Harpreet Singh, Amazon Web Services (AWS)*

*December 2021* (document history)

Amazon Web Services (AWS) provides services for developing and running extract, transform, and load (ETL) or extract, load, and transform (ELT) jobs and big data workloads. Microsoft SQL Server Integration Services (SSIS) is an on-premises ETL tool that has a graphical interface for building ETL jobs. Depending on the target state architecture, you can use the AWS Glue graphical interface or the Apache Spark framework with Python libraries to build ETL jobs in the AWS Cloud.

This guide describes the steps for migrating SSIS ETL/ELT jobs from on premises to AWS. It discusses migration phases, best practices, and recommendations to reduce the migration effort and improve the experience. The information is applicable to any target AWS architecture.

The guide is for program or project managers, product owners, solution architects, and developers who are:

- Migrating from SSIS to AWS for various reasons, such as modernization or cost savings.
- Primarily using AWS services for ETL and big data workloads.
- Looking for a cloud-based solution to take advantage of a serverless model and flexible pricing.

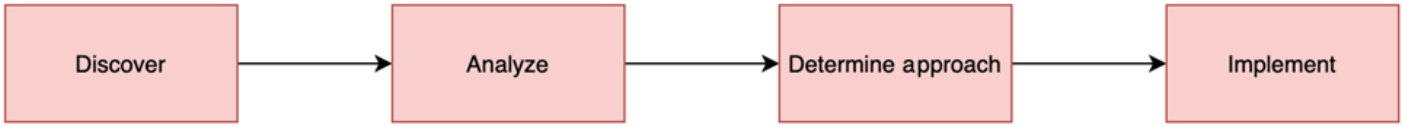
## Targeted business outcomes

Migrating your SSIS jobs to the AWS Cloud helps you achieve these primary outcomes:

- Modernize existing on-premises ETL processes in a cost-effective way
- Respond to the information needs of the organization faster
- Merge existing processes and retire unused processes to reduce maintenance and operational overhead
- Build a foundation to meet your organization's future data requirements

# Migration phases

Migrating SSIS ETL processes to the AWS Cloud consists of four major phases: discover, analyze, determine approach, and implement.




These phases are discussed in the following sections:

- Discovery phase
- Analysis phase
- Determining the migration approach
- Implementation phase

## Discovery phase

In the discovery phase, you create a list of the SSIS packages that you want to migrate to AWS. Different development teams follow different styles, standards, and patterns for developing ETL jobs. We recommend that you review your organization's existing documents to understand these patterns. However, the documentation is often incomplete. You can automate the extraction of important information from the ETL scripts. This saves manual effort and time, reduces human errors, and standardizes the migration approach. Here are some of the important details you'll want to extract:

- Total number of control flow tasks
- Details of control flow tasks
- Total number of data flow tasks
- Data flow transformations used
- Event handlers
- Connection managers

Use this information to understand the ETL patterns used at your organization, to evaluate their complexity, and to identify the appropriate AWS service to migrate this information to.

Migrating these ETL details from SSIS forms the bulk of the migration effort. However, additional properties can provide insights into design and architectural decisions. Some of these SSIS properties are:

- Checkpoints, which are used in SSIS to restart jobs from points of failure
- Propagate variables, which help an SSIS package succeed in specific use cases, even when there is an error
- Transaction isolation levels, which control the quality of data being read from databases
- Logging, to understand the types of logs being captured by the current design and their storage locations

The outcome of the discovery phase can be an inventory, as the following table shows.

inventory

| count | Package | Flow | Task |
|---|---|---|---|
| 1 | SSIS_Package_1 | control_flow | Microsoft.ExecuteSQLTask |
| 1 | SSIS_Package_1 | data_flow | Microsoft.BDD |
| 1 | SSIS_Package_1 | data_flow | Microsoft.ConditionalSplit |
| 2 | SSIS_Package_1 | data_flow | Microsoft.OLEDBDestination |
| 1 | SSIS_Package_1 | data_flow | Microsoft.OLEDBSource |
| 1 | SSIS_Package_2 | control_flow | Microsoft.DbMaintenanceFileCleanupTask |
| 1 | SSIS_Package_2 | control_flow | Microsoft.ExecuteSQLTask |
| 1 | SSIS_Package_2 | control_flow | Microsoft.ScriptTask |
| 1 | SSIS_Package_2 | control_flow | STOCK:FOREACHLOOP |
| 1 | SSIS_Package_2 | control_flow | STOCK:FORLOOP |

This inventory might include the following information:

- Package: Name of the SSIS package to migrate

- Flow: Control flow or data flow
- Task: Name of the control flow task or data flow component
- Count: Number of times a task was used in the SSIS package

## Analysis phase

Analyzing the information from the discovery phase helps you understand the patterns and design the target architecture better. Follow these pointers to improve the outcome of analysis:

- The logging option at the package level logs events and captures runtime information for each component. Use custom logging so that you can configure log files to include additional information, such as execution IDs and other runtime values.
- Redirect bad records to a different storage location for analysis, correction, and reprocessing.
- Understand the data archival and purging strategies implemented in your on-premises SSIS environment.
- Send alerts and notifications by using tasks that are included with SSIS (such as the Send Mail task) or custom scripts (such as the Script task).
- Understand the behavior of transformations in scope to avoid corrupted data. For example, the Lookup transformation is an equi-join between two datasets, and its comparison is case-sensitive.

You can map each entry in the inventory to an estimated complexity, either in terms of duration (minutes or hours) or level (simple, medium, or complex). This expanded inventory, shown in the following table, is an outcome of the analysis phase.

inventory

| count | Package | Flow | Task | Effort | Complexity |
|---|---|---|---|---|---|
| 1 | SSIS_Package_1 | control_flow | Microsoft.ExecuteSQLTask | 30 | S |
| 1 | SSIS_Package_1 | data_flow | Microsoft.BDD | 0 | N/A |
| 1 | SSIS_Package_1 | data_flow | Microsoft.ConditionalSplit | 30 | S |
| 2 | SSIS_Package_1 | data_flow | Microsoft.OLEDBDestination | 60 | M |
| 1 | SSIS_Package_1 | data_flow | Microsoft.OLEDBSource | 30 | S |
| 1 | SSIS_Package_2 | control_flow | Microsoft.DbMaintenanceFileCleanupTask | 60 | M |
| 1 | SSIS_Package_2 | control_flow | Microsoft.ExecuteSQLTask | 30 | S |
| 1 | SSIS_Package_2 | control_flow | Microsoft.ScriptTask | 120 | C |
| 1 | SSIS_Package_2 | control_flow | STOCK:FOREACHLOOP | 30 | S |
| 1 | SSIS_Package_2 | control_flow | STOCK:FORLOOP | 30 | S |

## Determining the migration approach

To decide on a migration approach, you use the analysis you performed on existing patterns in the previous phase. Your organization's future data and analytics needs are equally important considerations. Traditional on-premises ETL tools deal with relational data models and structured data. If you have semi-structured and unstructured data to process, you can use AWS services such as AWS Glue or Amazon EMR for the migration. Other factors that can influence the migration approach include:

- Whether you want to use a graphical interface (such as AWS Glue Studio) or a custom framework (such as Spark/Python libraries)
- Whether you have secure access to on-premises sources and AWS targets
- Skills and training required for the team
- Audit and compliance requirements

You can select from three migration approaches: big bang, phased, and lift and shift. The following table compares these three approaches.

| Approach | Description | Use case | Advantages and disadvantages |
| --- | --- | --- | --- |
| Big bang | Migrate all SSIS packages within a specific time period. | • Complexity, scope, and target architecture are clear.<br>• Team has the required skills, or the learning curve is shallow. | • High risk.<br>• Takes less time than the phased approach.<br>• You can use AWS Glue, Amazon EMR, or custom frameworks. |
| Phased | Identify one SSIS package for each distinct pattern and complexity. Migrate the package to AWS, test, and compare results with existing architecture. | • Time is not a constraint.<br>• You want different designs for different ETL patterns. | • Less risky than the big bang approach but takes more time and effort.<br>• You can use AWS Glue, Amazon EMR, or custom frameworks. |
| Lift and shift | Migrate the current architecture as is to AWS. | • Your on-premises hardware is no longer supported.<br>• You don't have the resources to plan a migration immediately. | • Least amount of migration effort and time required.<br>• The problems with the existing solution remain on AWS.<br>• SSIS packages are run as is. No other ETL tools or frameworks are needed. |

A comparison of data on the source and target systems is fundamental for a successful migration. Because the existing production system gets regular updates from source systems, this comparison might become confusing. For this reason, when you're determining your migration approach, we recommend that you also decide on your data validation strategy.

- Take backups of all applicable databases and files from the production environment on the source system at a specific date and time.
- Take backups of all databases from the production environment on the target system after all jobs have successfully loaded data from backed up source data.
- Restore the source data in a testing environment, and run the new jobs.
- Agree on a percentage of valid differences between the source and target (old and new) databases. For example, you might decide that a difference of less than 1% is acceptable.
- List all the validation rules to be covered.
- Automate the comparison as much as possible, and cover all the rules.

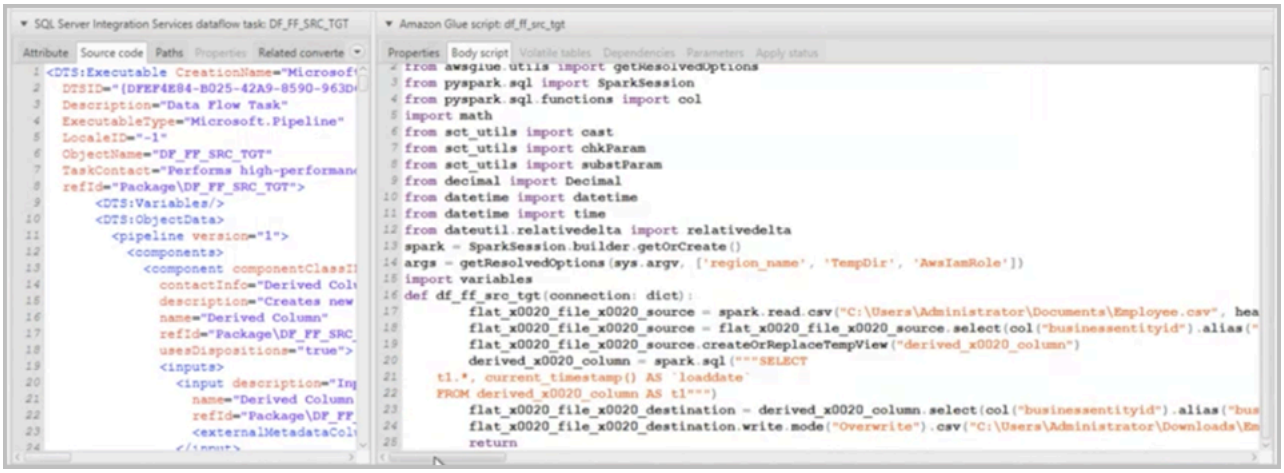
# Implementation phase

Migration that follows the big bang or phased approach requires new development and testing. The AWS Schema Conversion Tool (AWS SCT) can automatically generate AWS Glue jobs from SSIS packages. This reduces the migration time and effort significantly. Or, you can use AWS Glue Studio for graphical interface-based development, or build Spark libraries that you can run on either AWS Glue or Amazon EMR.

The following sections provide useful pointers for using AWS SCT, AWS Glue, and Amazon EMR.

## AWS SCT

The following screen illustration shows an AWS Glue job script that was converted by AWS SCT.

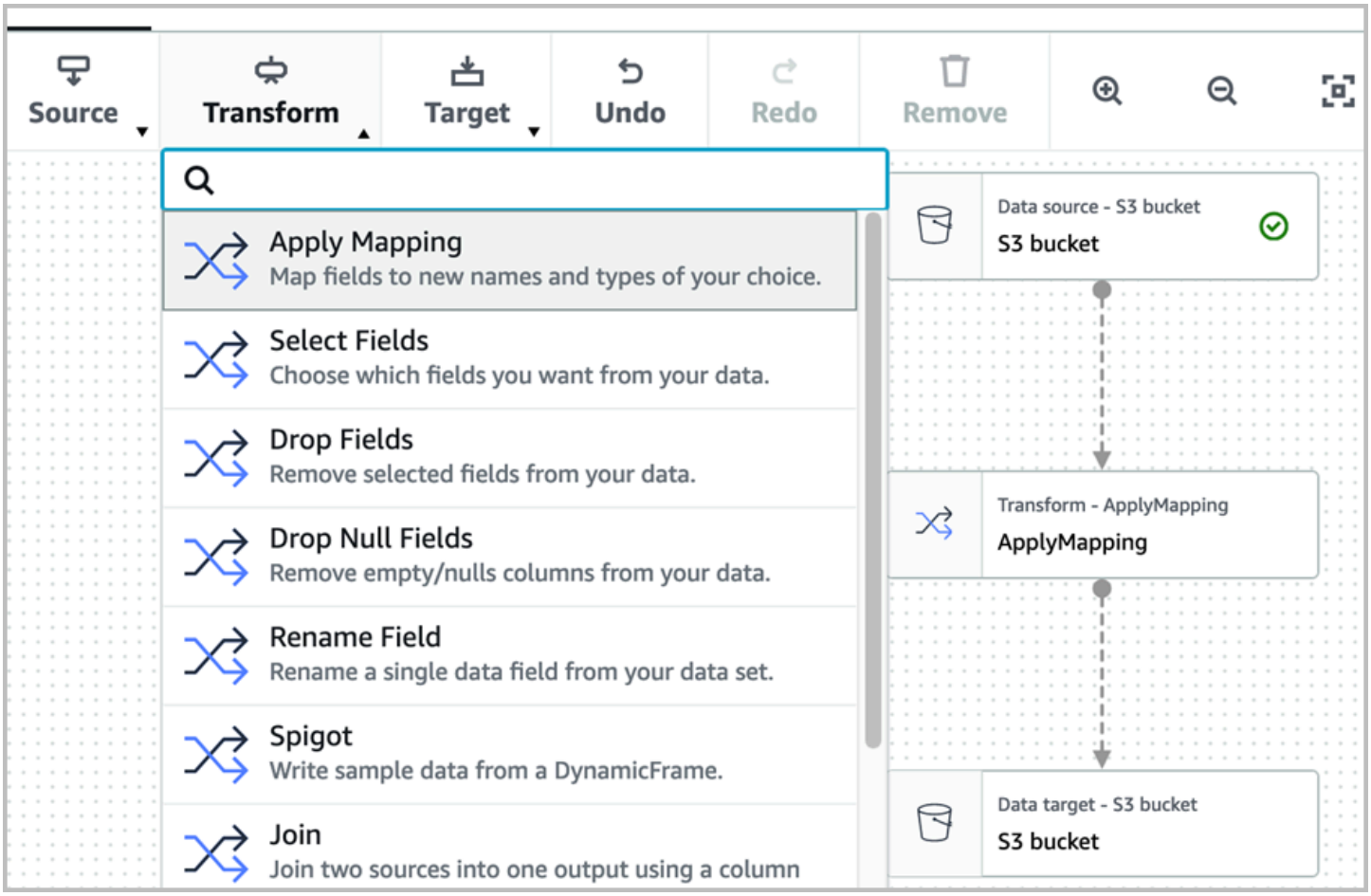



AWS SCT can convert SSIS packages to AWS Glue jobs in bulk. You can edit the script to update existing logic or to add new logic, based on your new design. We recommend that you follow the naming conventions in the AWS SCT converted scripts to customize the scripts.

For more information, see Converting SSIS to AWS Glue using AWS SCT in the AWS SCT documentation.

## AWS Glue

AWS Glue Studio provides a graphical interface and a development experience that's similar to SSIS, as illustrated in the following screen.




If you prefer not to use a graphical interface, you can also run your custom scripts with the required Python libraries from the AWS Glue console. For more information, see Providing your own custom scripts in the AWS Glue documentation.

AWS Glue provides a set of built-in transforms for processing your data. These are similar to SSIS data flow transformations. Follow these best practices when you migrate your SSIS ETL jobs by using AWS Glue:

- Prepare a mapping from AWS Glue transforms to the equivalent SSIS transformations.
- If your transformations cannot be mapped to AWS Glue transforms, build them by using a Python or Scala custom script.
- For custom logging (such as rows read, rows written, or bad records), use custom scripts in addition to Amazon CloudWatch.
- Add a development endpoint to develop and debug custom scripts locally.

## Amazon EMR

You can run custom scripts (written in Python or Scala) or compiled Python libraries in EMR clusters, as with AWS Glue. Follow these best practices:

- Start with memory optimized instance types while creating EMR clusters with the Spark framework. (SSIS uses memory buffers.)
- Build generic Python methods that are equivalent to each SSIS task or transformation. For example, in the following illustration, a method that takes two dataframes as input produces a third dataframe that has matching records from the two dataframes as output. This works as a merge join transformation.

```
AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def merge_join_spark(spark: SparkSession, df1: DataFrame, df2: DataFrame, join_type: str, join_column:str,
                     merge_fetch_columns: str):
    """Merge/Join directly
        spark: Current Spark session
        df1: First dataframe for merge join
        df2: Second dataframe for merge join
        join_type: Left/Inner/Outer join
        join_column : Column to be merge joined on
        merge_fetch_columns: Columns required in the output dataframe
    """
    try:

        spark.catalog.dropTempView("df1")
        spark.catalog.dropTempView("df2")
        df1.createOrReplaceTempView("df1")
        df2.createOrReplaceTempView("df2")
        merge_sql = f"select src.*, {merge_fetch_columns} from df1 {join_type} join df2 on df1.{join_column} = df2.{join_column}".format(
            merge_fetch_columns=merge_fetch_columns, join_type=join_type,join_column=join_column)
        logger.debug("Merge query is : " + merge_sql)
        output_df = spark.sql(merge_sql)
    except Exception as ex:
        logger.error("Merge Execution Failed for " + str(ex))
        raise Exception("Merge Execution Failed " + str(ex))
    finally:
        spark.catalog.dropTempView("df1")
        spark.catalog.dropTempView("df2")
    return output_df
```

## Testing

A testing framework is required to validate the completeness and correctness of data. This framework should cover all the existing scenarios and any improvements you made while migrating your jobs to AWS.

- Completeness validation:
  - All jobs are migrated to their target state.
  - All functionality is migrated in each job.

  - All types of logs are available, including job execution details, error messages, bad records, and row counts.
- Correctness validation:
  - The quality of data is consistent in the existing and new environments.
  - All columns of all tables match, or tables are improved on AWS.
  - All audit and logging information match.

You should also verify that the performance of your migrated jobs matches the performance of your existing jobs.

# Best practices

Follow these general best practices when migrating your SSIS jobs to AWS:

- If you're migrating your databases to an AWS managed service such as Amazon Relational Database Service (Amazon RDS), descope maintenance tasks (such as backup, restore, update statistics, or check database integrity) that are not applicable or managed by AWS.
- Build reusable scripts and methods to standardize development and reduce effort.
- Always test migrated jobs on infrastructure and data volumes that match production resources.
- SSIS packages are in XML format. Build XML parser utilities to automate migration activities where possible. The following example shows an SSIS package in XML format.

```
    DTS:LastModifiedProductVersion="11.0.2100.60"
    DTS:LocaleID="1033"
    DTS:ObjectName="Package"
    DTS:PackageType="5"
    DTS:VersionBuild="1"
    DTS:VersionGUID="{EC16490E-6783-4BE6-92CA-FDD5BC644F88}">
    <DTS:Property
      DTS:Name="PackageFormatVersion">6</DTS:Property>
    <DTS:ConnectionManagers>
      <DTS:ConnectionManager
```

The following illustration shows a sample XML parser.

```
def getlistofallexecutables(directory):
    """
        Iterate the root directory with all SSIS packages (.dtsx)
        Get the control flow tasks and data flow task components
    """
    try:
        lst=[]
        for fileName in os.listdir(directory):
                if fileName.endswith('.dtsx'):
                        cfgfilename=os.path.basename(fileName).split('.')[0]
                        tree = etree.parse(directory+fileName)
                        root = tree.getroot()
                        prefix = '{www.microsoft.com/'
                        exetag = prefix + 'SqlServer/Dts}Executable'
                        dataflow='Microsoft.Pipeline'
                        childtag=prefix+ 'SqlServer/Dts}ExecutableType'
                        objdata=prefix+ 'SqlServer/Dts}ObjectData'
                        pipeline='pipeline'
                        components= 'components'
                        componentclassid="componentClassID"
                        for cnt, ele in enumerate(root.xpath(".//*")):
                                if ele.tag==exetag:
                                        attr=ele.attrib[childtag]#controlflow
                                        flow=cfgfilename+',control_flow,'
                                        if attr!=dataflow:
                                                lst.append(flow+attr)
                                        if attr==dataflow:
                                                for child0 in ele:
                                                        if child0.tag==objdata:
                                                                for child1 in child0:
                                                                        if child1.tag==pipeline:
                                                                                for child2 in child1:
                                                                                        if child2.tag==components:
                                                                                                for child3 in child2:
                                                                                                        df_component=child3.attrib[componentclassid]
                                                                                                        flow=cfgfilename+',data_flow,'
                                                                                                        lst.append(flow+df_component)
        return lst
    except Exception as e:
```

- Use checksums and hash values to test correctness and completeness.
- If you're building custom libraries, implement threading to run tasks in parallel. This improves job performance.
- Decommission the existing environment only after all jobs are stable on AWS for a period of time.
- Use Amazon CloudWatch for logging, to reduce logging customization efforts.
- Use Amazon Simple Notification Service (Amazon SNS) to replace custom tasks for sending notifications. Use Amazon Simple Email Service (Amazon SES) to replace custom email tasks.

# FAQ

This section provides answers to commonly raised questions about migrating your SSIS ETL jobs to AWS.

## Should I enhance SSIS jobs during migration?

Yes. Implement any important enhancements or bug fixes in both existing and new jobs at the same time. Low-priority changes can be implemented after migration.

## How can I monitor jobs on AWS?

You can use Amazon CloudWatch to monitor jobs and to send rule-based alerts.

## When can I decommission my on-premises SSIS jobs?

We recommend that you maintain old and new jobs in parallel for a period of time until the performance, accuracy, and completeness of data are the same in both environments. You can also replicate reporting systems to connect and compare reports from both environments. You can decommission your SSIS jobs when the reports are identical.

# Next steps

As the next step, we recommend that you build a proof of concept to determine your deployment approach for the target architecture. For your proof of concept:

- Create AWS Glue jobs by using the AWS Glue Studio graphical interface.
- Use the AWS Schema Conversion Tool (AWS SCT) to generate AWS Glue scripts from SSIS packages.
- Build a custom migration framework by using Spark libraries.
- Build a testing framework.
- Identify tools and processes for continuous integration and continuous deployment (CI/CD).

# Related resources

- Converting SSIS to AWS Glue using AWS SCT
- Amazon RDS backup and restore
- AWS Glue documentation
- AWS Glue FAQs
- Continuous logging for AWS Glue jobs
- Amazon EMR documentation
- Amazon EMR FAQs
- Monitor metrics with CloudWatch
- Accelerate large-scale data migration validation using PyDeequ
- AWS Data and Analytics Competency Partners

# Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

| Change | Description | Date |
| --- | --- | --- |
| Initial publication | — | December 6, 2021 |