AWS Big Data Blog

Orchestrate Amazon Redshift-Based ETL workflows with AWS Step Functions and AWS Glue

by Ben Romano | on 11 OCT 2019 | in Amazon Redshift, Amazon Redshift, Analytics, AWS Big Data, AWS Glue, AWS Step Functions, Database | Permalink | Comments | Share

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud that offers fast query performance using the same SQL-based tools and business intelligence applications that you use today. Many customers also like to use Amazon Redshift as an extract, transform, and load (ETL) engine to use existing SQL developer skillsets, to quickly migrate pre-existing SQL-based ETL scripts, and—because Amazon Redshift is fully ACID-compliant—as an efficient mechanism to merge change data from source data systems.

In this post, I show how to use AWS Step Functions and AWS Glue Python Shell to orchestrate tasks for those Amazon Redshift-based ETL workflows in a completely serverless fashion. AWS Glue Python Shell is a Python runtime environment for running small to medium-sized ETL tasks, such as submitting SQL queries and waiting for a response. Step Functions lets you coordinate multiple AWS services into workflows so you can easily run and monitor a series of ETL tasks. Both AWS Glue Python Shell and Step Functions are serverless, allowing you to automatically run and scale them in response to events you define, rather than requiring you to provision, scale, and manage servers.

While many traditional SQL-based workflows use internal database constructs like triggers and stored procedures, separating workflow orchestration, task, and compute engine components into standalone services allows you to develop, optimize, and even reuse each component independently. So, while this post uses Amazon Redshift as an example, my aim is to more generally show you how to orchestrate any SQL-based ETL.

Prerequisites

If you want to follow along with the examples in this post using your own AWS account, you need a Virtual Private Cloud (VPC) with at least two private subnets that have routes to an S3 VPC endpoint.

If you don't have a VPC, or are unsure if yours meets these requirements, I provide an AWS CloudFormation template stack you can launch by selecting the following button. Provide a stack name on the first page and leave the default settings for everything else. Wait for the stack to display **Create Complete** (this should only take a few minutes) before moving on to the other sections.

Launch Stack

Scenario

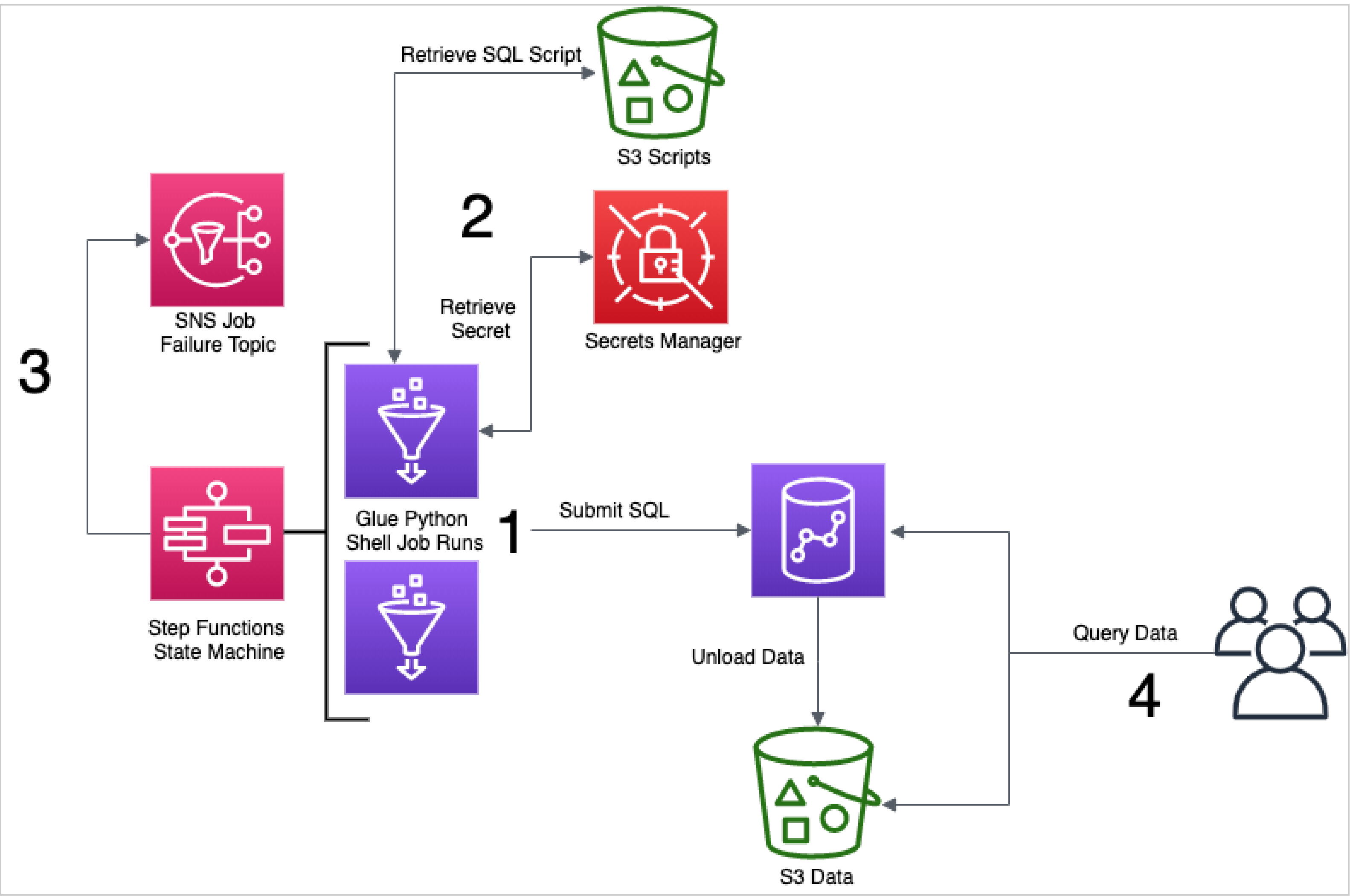
For the examples in this post, I use the Amazon Customer Reviews Dataset to build an ETL workflow that completes the following two tasks which represent a simple ETL process.

- *Task 1*: Move a copy of the dataset containing reviews from the year 2015 and later from S3 to an Amazon Redshift table.
- *Task 2*: Generate a set of output files to another Amazon S3 location which identifies the "most helpful" reviews by market and product category, allowing an analytics team to glean information about high quality reviews.

This dataset is publicly available via an Amazon Simple Storage Service (Amazon S3) bucket. Complete the following tasks to get set up.

Solution overview

The following diagram highlights the solution architecture from end to end:



The steps in this process are as follows:

1. The state machine launches a series of runs of an AWS Glue Python Shell job (more on how and why I use a single job later in this post!) with parameters for retrieving database connection information from AWS Secrets Manager and an .sql file from S3.
2. Each run of the AWS Glue Python Shell job uses the database connection information to connect to the Amazon Redshift cluster and submit the queries contained in the .sql file.
    1. *For Task 1*: The cluster utilizes Amazon Redshift Spectrum to read data from S3 and load it into an Amazon Redshift table. Amazon Redshift Spectrum is commonly used as an means for loading data to Amazon Redshift. (See Step 7 of *Twelve Best Practices for Amazon Redshift Spectrum* for more information.)
    2. *For Task 2*: The cluster executes an aggregation query and exports the results to another Amazon S3 location via UNLOAD.
3. The state machine may send a notification to an Amazon Simple Notification Service (SNS) topic in the case of pipeline failure.
4. Users can query the data from the cluster and/or retrieve report output files directly from S3.

I include an AWS CloudFormation template to jumpstart the ETL environment so that I can focus this post on the steps dedicated to building the task and orchestration components. The template launches the following resources:

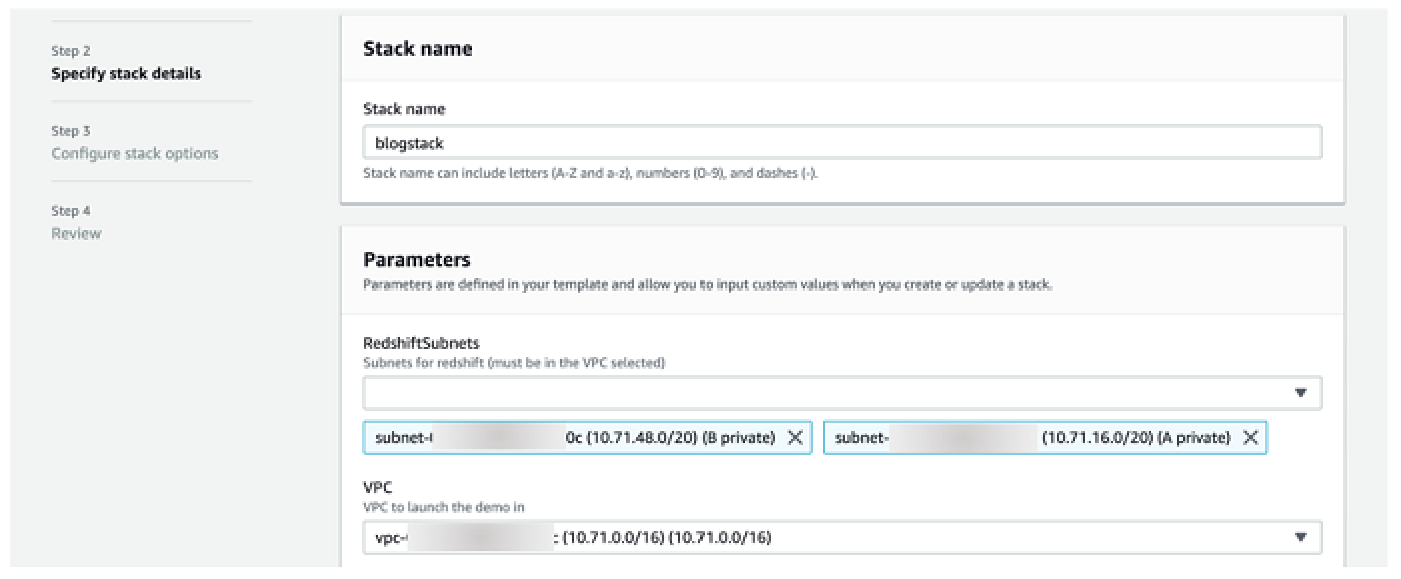
- Amazon Redshift Cluster
- Secrets Manager secret for storing Amazon Redshift cluster information and credentials
- S3 Bucket preloaded with Python scripts and .sql files
- Identity and Access Management (IAM) Role for AWS Glue Python Shell jobs

See the following resources for how to complete these steps manually:

- Create a Redshift Cluster in a VPC
- Creating and Managing Secrets with AWS Secrets Manager
- Getting Started with Simple Storage Service (S3)
- Create IAM Roles

Launch Stack

Be sure to select at least two private subnets and the corresponding VPC, as shown in the following screenshot. If you are using the VPC template from above, the VPC appears as **10.71.0.0/16** and the subnet names are **A private** and **B private**.



Stack name

Stack name: blogstack

Stack name can include letters (A-Z and a-z), numbers (0-9), and dashes (-).

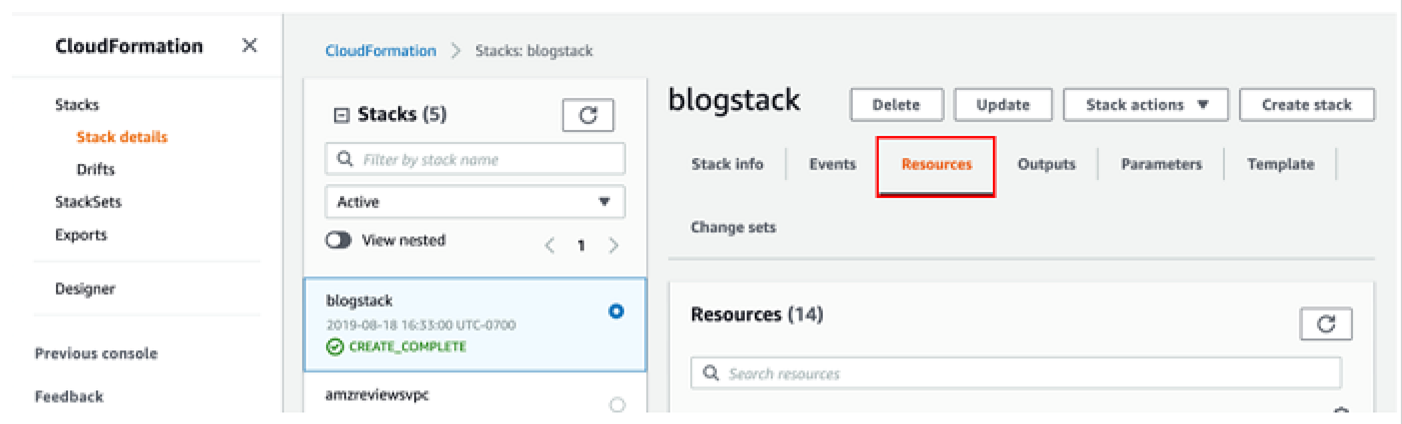
Parameters

Parameters are defined in your template and allow you to input custom values when you create or update a stack.

RedshiftSubnets
Subnets for redshift (must be in the VPC selected)
subnet-[redacted]0c (10.71.48.0/20) (B private) | subnet-[redacted] (10.71.16.0/20) (A private)

VPC
VPC to launch the demo in
vpc-[redacted]: (10.71.0.0/16) (10.71.0.0/16)

The stack should take 10-15 minutes to launch. Once it displays **Create Complete**, you can move on to the next section. Be sure to take note of the **Resources** tab in the AWS CloudFormation console, shown in the following screenshot, as I refer to these resources throughout the post.



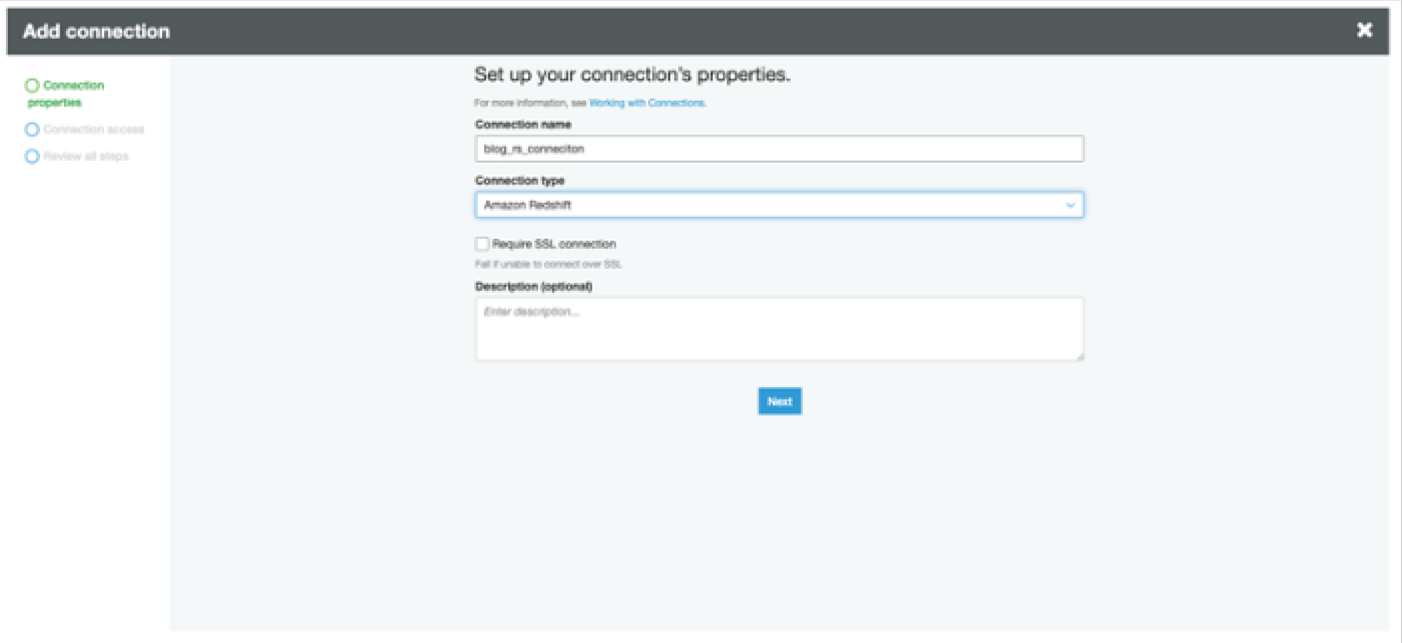
Building with AWS Glue Python Shell

Begin by navigating to **AWS Glue** in the AWS Management Console.

Making a connection

Amazon Redshift cluster resides in a VPC, so you first need to create a connection using AWS Glue. Connections contain properties, including VPC networking information, needed to access your data stores. You eventually attach this connection to your Glue Python Shell Job so that it can reach your Amazon Redshift cluster.

Select **Connections** from the menu bar, and then select **Add connection**. Give your connection a name like *blog_rs_connection*, select **Amazon Redshift** as the **Connection type**, and then select **Next**, as shown in the following screenshot.



Set up your connection's properties.

For more information, see Working with Connections.

Connection name: blog_rs_conneciton

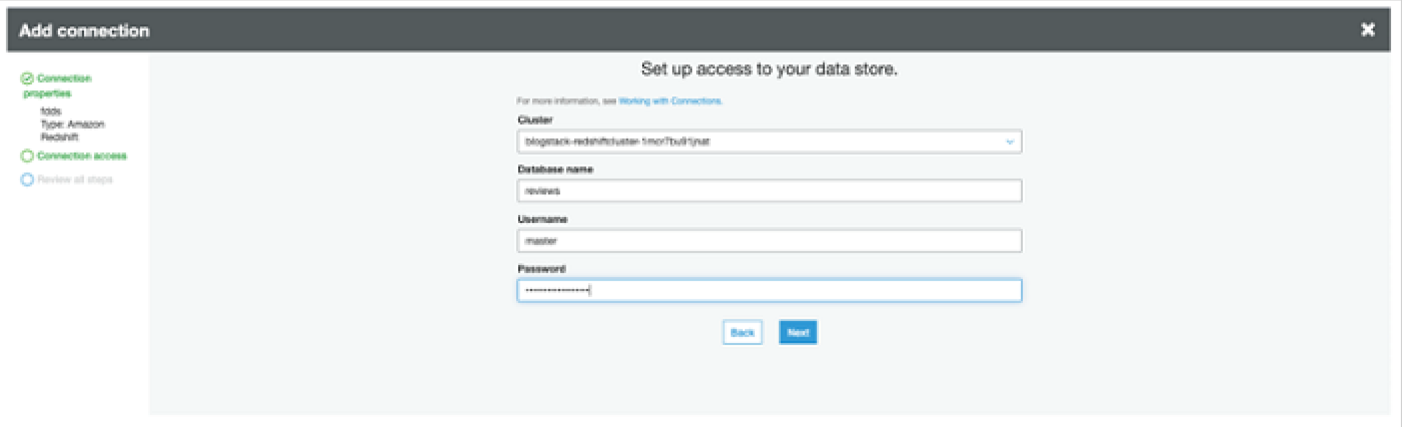
Connection type: Amazon Redshift

☐ Require SSL connection
Fail if unable to connect over SSL

Description (optional): Enter description...

Under **Cluster**, enter the name of the cluster that the AWS CloudFormation template launched, i.e *blogstack-redshiftcluster-####*. Because the Python code I provide for this blog already handles credential retrieval, the rest of the values around database information you enter here are largely placeholders. The key information you are associating with the connection is networking-related.

Please note that you are not able to test the connection without the correct cluster information. If you are interested in doing so, note that ***Database name*** *and* ***Username*** *are auto-populated after selecting the correct cluster, as shown in the following screenshot. Follow the instructions here to retrieve the password information from Secrets Manager to copy into the* ***Password*** *field.*



Add connection

Set up access to your data store.

For more information, see Working with Connections.

Cluster: blogstack-redshiftcluster-1mcr7bu91jnat

Database name: reviews

Username: master

Password: ••••••••••••••

ETL code review

Take a look at the two main Python scripts used in this example:

Pygresql_redshift_common.py is a set of functions that can retrieve cluster connection information and credentials from Secrets Manger, make a connection to the cluster, and submit queries respectively. By retrieving cluster information at runtime via a passed parameter, these functions allow the job to connect to any cluster to which it has access. You can package these functions into a library by following the instructions to create a python .egg file (already completed as a part of the AWS CloudFormation template launch). Note that AWS Glue Python Shell supports several python libraries natively.

```
import pg
import boto3
import base64
from botocore.exceptions import ClientError
import json

#uses session manager name to return connection and credential information
def connection_info(db):

    session = boto3.session.Session()
    client = session.client(
        service_name='secretsmanager'
    )

    get_secret_value_response = client.get_secret_value(SecretId=db)

    if 'SecretString' in get_secret_value_response:
        secret = json.loads(get_secret_value_response['SecretString'])
```

The AWS Glue Python Shell job runs *rs_query.py* when called. It starts by parsing job arguments that are passed at invocation. It uses some of those arguments to retrieve a .sql file from S3, then connects and submits the statements within the file to the cluster using the functions from *pygresql_redshift_common.py*. So, in addition to connecting to any cluster using the Python library you just packaged, it can also retrieve and run any SQL statement. This means you can manage a single AWS Glue Python Shell job for all of your Amazon Redshift-based ETL by simply passing in parameters on where it should connect and what it should submit to complete each task in your pipeline.

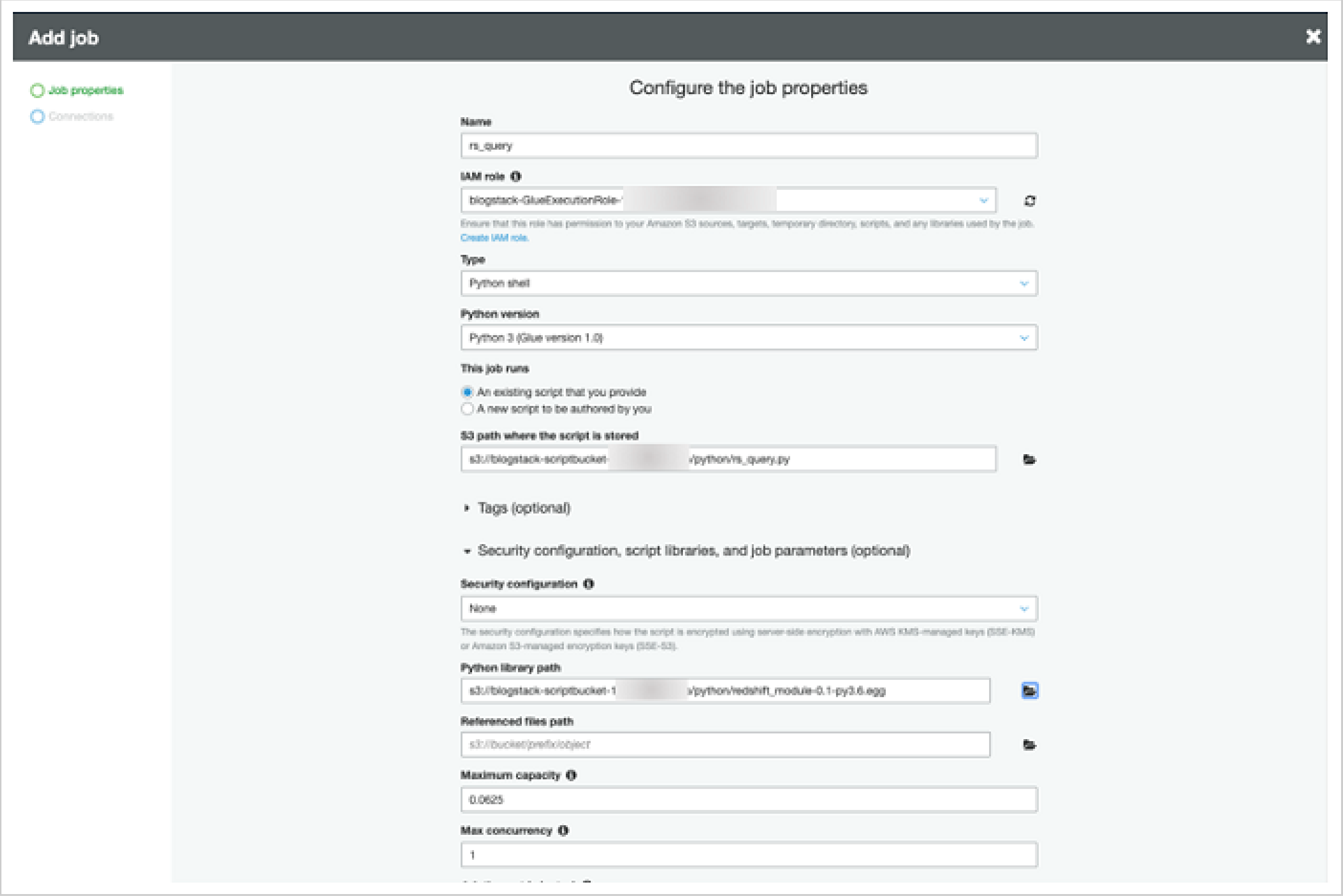
```
from redshift_module import pygresql_redshift_common as rs_common
import sys
from awsglue.utils import getResolvedOptions
import boto3

#get job args
args = getResolvedOptions(sys.argv,['db','db_creds','bucket','file'])
db = args['db']
db_creds = args['db_creds']
bucket = args['bucket']
file = args['file']

#get sql statements
s3 = boto3.client('s3')
sqls = s3.get_object(Bucket=bucket, Key=file)['Body'].read().decode('utf-8')
sqls = sqls.split(';')

#get database connection
print('connecting...')
```

Creating the Glue Python Shell Job

Next, put that code into action:

1. Navigate to **Jobs** on the left menu of the AWS Glue console page and from there, select **Add job**.
2. Give the job a name like *blog_rs_query*.
3. For the **IAM role**, select the same **GlueExecutionRole** you previously noted from the **Resources** section of the AWS CloudFormation console.
4. For **Type**, select **Python shell**, leave **Python version** as the default of **Python 3**, and for **This job runs** select **An existing script that you provide**.
5. For **S3 path where the script is stored**, navigate to the script bucket created by the AWS CloudFormation template (look for **ScriptBucket** in the **Resources**), then select the *python/py* file.
6. Expand the **Security configuration, script libraries, and job parameters** section to add the Python .egg file with the Amazon Redshift connection library to the **Python library path**. It is also located in the script bucket under *python /redshift_module-0.1-py3.6.egg.*

When all is said and done everything should look as it does in the following screenshot:



Add job

○ Job properties
○ Connections

Configure the job properties

Name: rs_query

IAM role: blogstack-GlueExecutionRole-[redacted]

Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.
Create IAM role.

Type: Python shell

Python version: Python 3 (Glue version 1.0)

This job runs
● An existing script that you provide
○ A new script to be authored by you

S3 path where the script is stored: s3://blogstack-scriptbucket-[redacted]/python/rs_query.py

▸ Tags (optional)

▾ Security configuration, script libraries, and job parameters (optional)

Security configuration: None

The security configuration specifies how the script is encrypted using server-side encryption with AWS KMS-managed keys (SSE-KMS) or Amazon S3-managed encryption keys (SSE-S3).

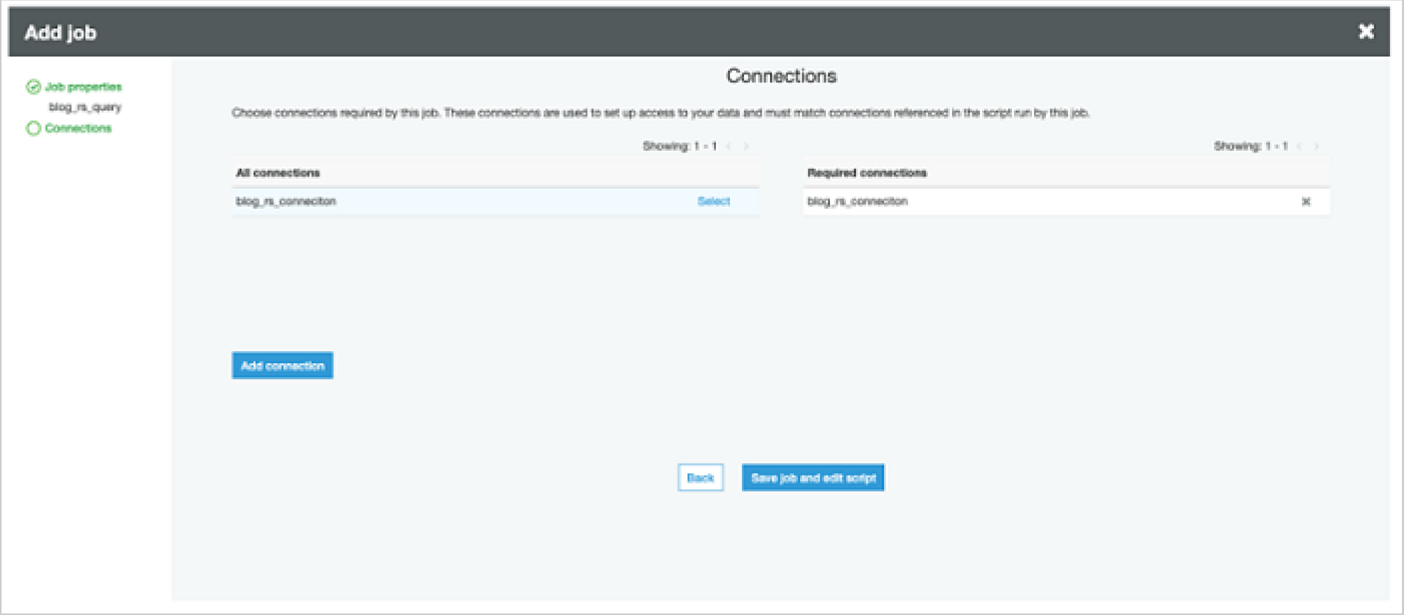
Python library path: s3://blogstack-scriptbucket-1[redacted]/python/redshift_module-0.1-py3.6.egg

Referenced files path: s3://bucket/prefix/object

Maximum capacity: 0.0625

Max concurrency: 1

Choose **Next**. Add the connection you created by choosing **Select** to move it under **Required connections**. (Recall from the *Making a connection* section that this gives the job the ability to interact with your VPC.) Choose **Save job and edit script** to finish, as shown in the following screenshot.



Add job

✓ Job properties
blog_rs_query
○ Connections

Connections

Choose connections required by this job. These connections are used to set up access to your data and must match connections referenced in the script run by this job.

Showing: 1 - 1

All connections	
blog_rs_connection	Select

Showing: 1 - 1

Required connections	
blog_rs_connecition	✕

Test driving the Python Shell job

After creating the job, you are taken to the AWS Glue Python Shell IDE. If everything went well, you should see the *rs_query.py* code. Right now, the Amazon Redshift cluster is sitting there empty, so use the Python code to run the following SQL statements and populate it with tables.

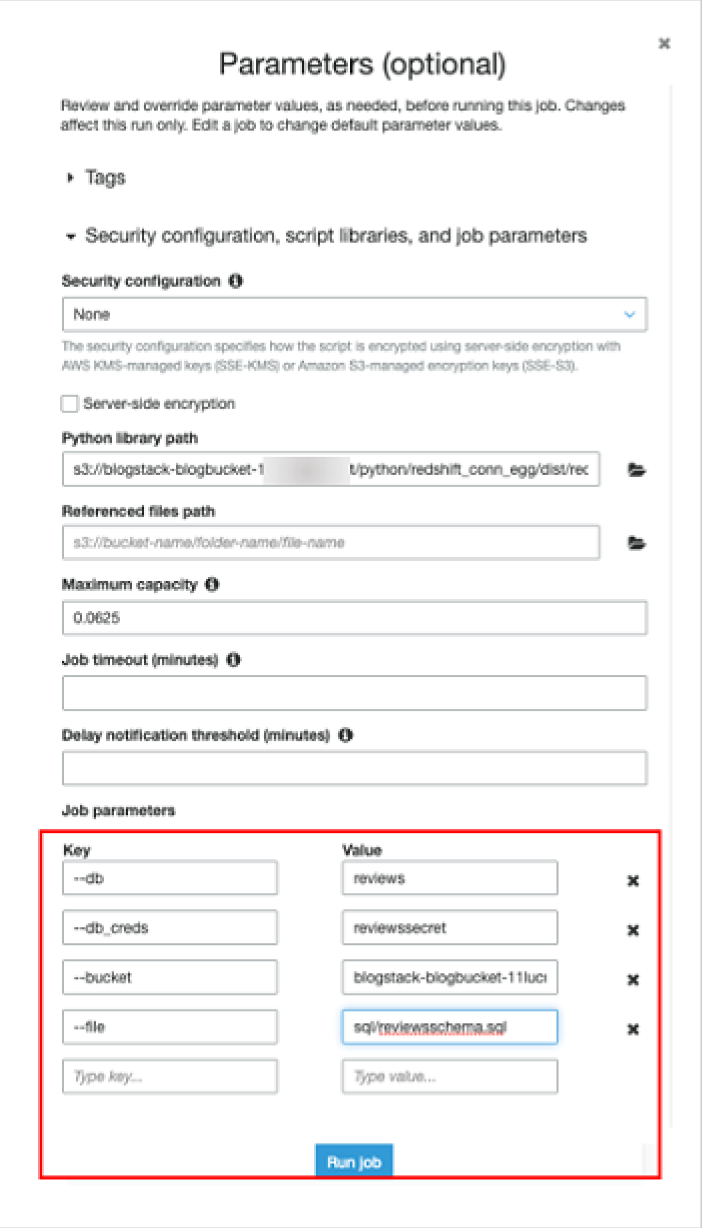
1. Create an external database (*amzreviews).*
2. Create an external table (*reviews)* from which Amazon Redshift Spectrum can read from the source data in S3 (the public reviews dataset). The table is partitioned by *product_category* because the source files are organized by category, but in general you should partition on frequently filtered columns (see #4).
3. Add partitions to the external table.
4. Create an internal table (*reviews)* local to the Amazon Redshift cluster. *product_id* works well as a DISTKEY because it has high cardinality, even distribution, and most likely (although not explicitly part of this blog's scenario) a column that will be used to join with other tables. I choose *review_date*

as a SORTKEY to efficiently filter out review data that is not part of my target query (after 2015). Learn more about how to best choose DISTKEY/SORTKEY as well as additional table design parameters for optimizing performance by reading the *Designing Tables documentation*.

```
CREATE EXTERNAL SCHEMA amzreviews
from data catalog
database 'amzreviews'
iam_role 'rolearn'
CREATE EXTERNAL database IF NOT EXISTS;



CREATE EXTERNAL TABLE amzreviews.reviews(
  marketplace varchar(10),
  customer_id varchar(15),
  review_id varchar(15),
  product_id varchar(25),
  product_parent varchar(15),
  product_title varchar(50),
  star_rating int,
  helpful_votes int,
  total_votes int,
  vine varchar(5),
```

Do this first job run manually so you can see where all of the elements I've discussed come into play. Select **Run Job** at the top of the IDE screen. Expand the **Security configuration, script libraries, and job parameters** section. This is where you add in the parameters as key-value pairs, as shown in the following screenshot.

Key	**Value**
–db	reviews
–db_creds	reviewssecret
–bucket	*<name of s3 script bucket>*
–file	sql/reviewsschema.sql



Parameters (optional)

Review and override parameter values, as needed, before running this job. Changes affect this run only. Edit a job to change default parameter values.

▸ Tags

▾ Security configuration, script libraries, and job parameters

Security configuration: None

The security configuration specifies how the script is encrypted using server-side encryption with AWS KMS-managed keys (SSE-KMS) or Amazon S3-managed encryption keys (SSE-S3).

□ Server-side encryption

Python library path: s3://blogstack-blogbucket-1[redacted]t/python/redshift_conn_egg/dist/rec

Referenced files path: s3://bucket-name/folder-name/file-name

Maximum capacity: 0.0625

Job timeout (minutes):

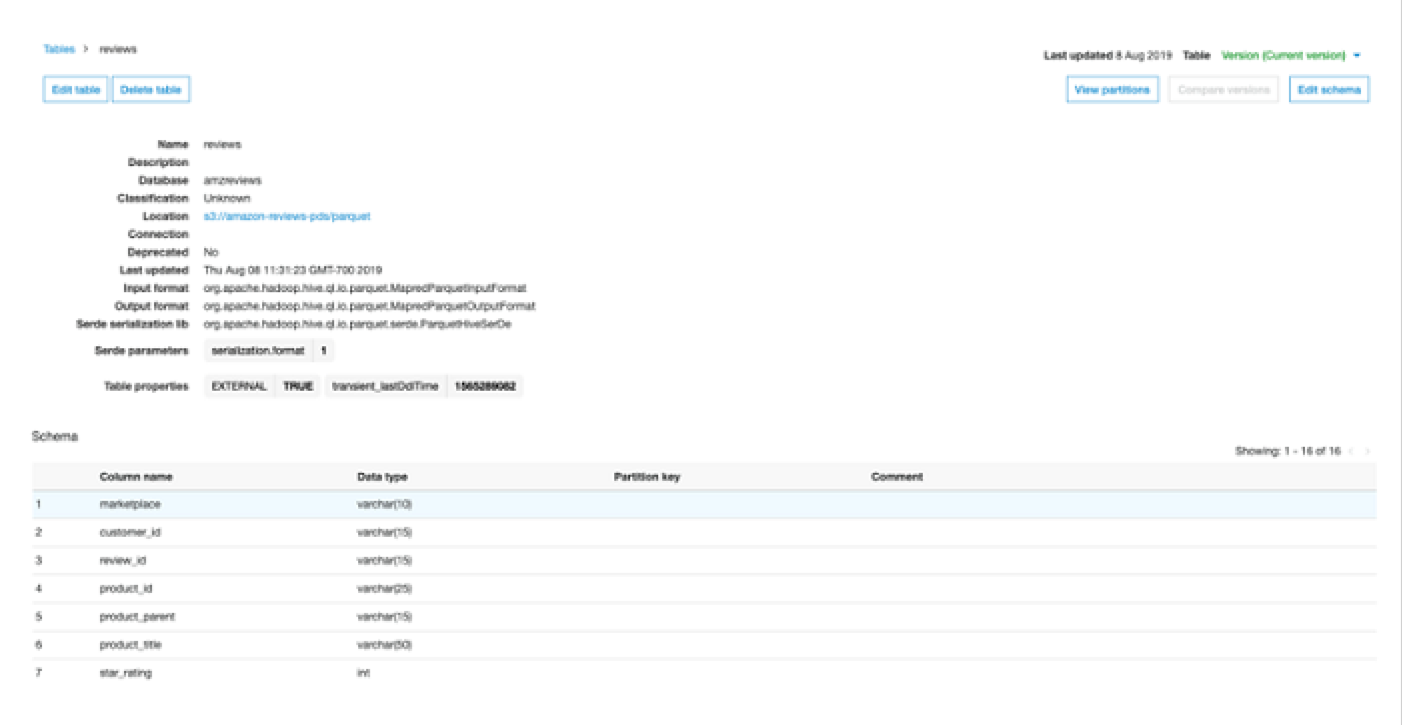
Delay notification threshold (minutes):

Job parameters

Key	Value
--db	reviews
--db_creds	reviewssecret
--bucket	blogstack-blogbucket-11luc
--file	sql/reviewsschema.sql
Type key...	Type value...

Select **Run job** to start it. The job should take a few seconds to complete. You can look for log outputs below the code in the IDE to watch job progress.

Once the job completes, navigate to **Databases** in the AWS Glue console and look for the *amzreviews* database and *reviews* table, as shown in the following screenshot. If they are there, then everything worked as planned! You can also connect to your Amazon Redshift cluster using the Redshift Query Editor or with your own SQL client tool and look for the local *reviews* table.



Last updated 8 Aug 2019 Table Version (Current version)

Name	reviews
Description	
Database	amzreviews
Classification	Unknown
Location	s3://amazon-reviews-pds/parquet
Connection	
Deprecated	No
Last updated	Thu Aug 08 11:31:23 GMT-700 2019
Input format	org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat
Output format	org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
Serde serialization lib	org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
Serde parameters	serialization.format 1
Table properties	EXTERNAL TRUE transient_lastDdlTime 1565289082

Schema

Showing: 1 - 16 of 16

	Column name	Data type	Partition key	Comment
1	marketplace	varchar(10)		
2	customer_id	varchar(15)		
3	review_id	varchar(15)		
4	product_id	varchar(25)		
5	product_parent	varchar(15)		
6	product_title	varchar(50)		
7	star_rating	int		

Step Functions Orchestration

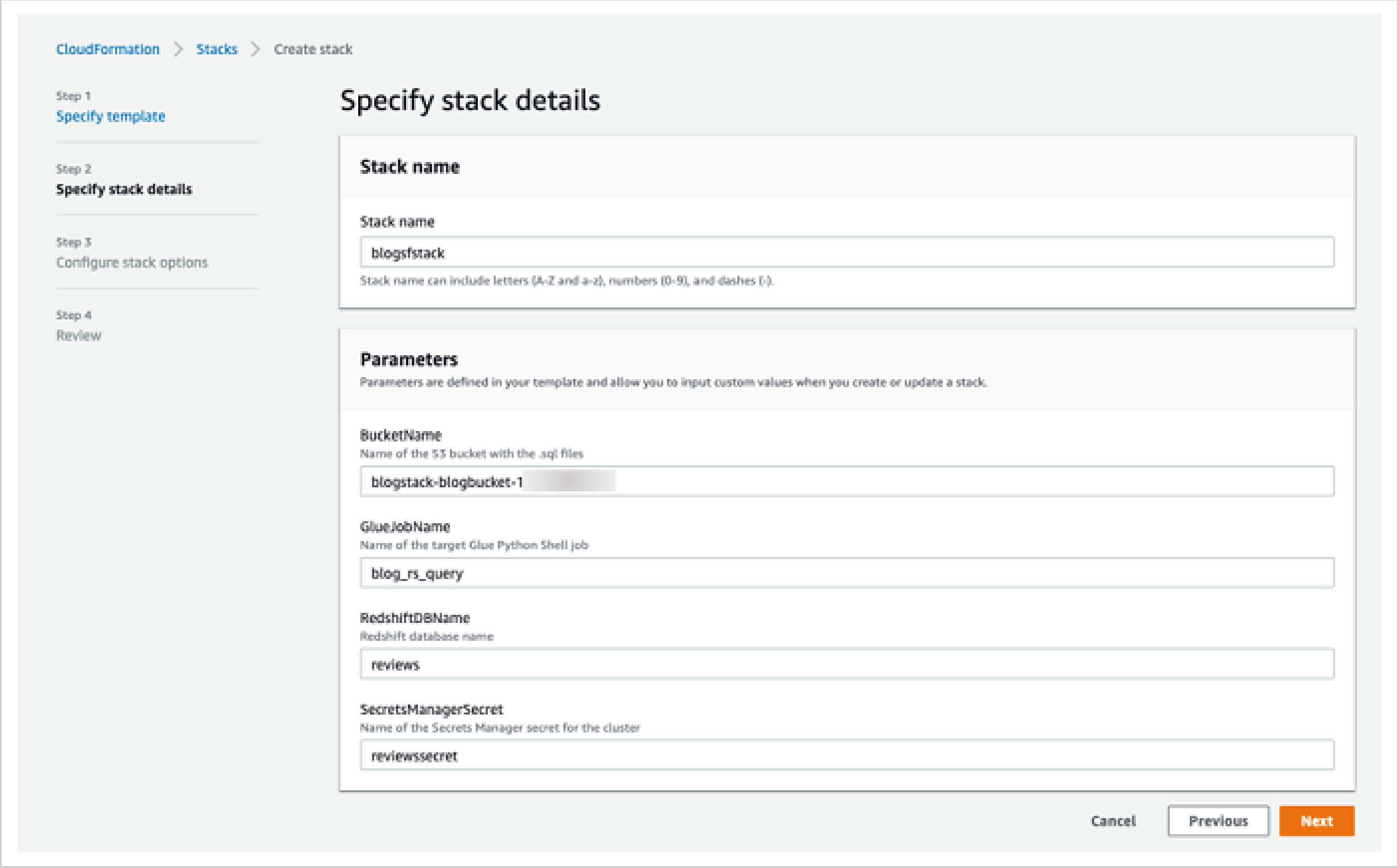
Now that you've had a chance to run a job manually, it's time to move onto something more programmatic that is orchestrated by Step Functions.

Launch Template

I provide a third AWS CloudFormation template for kickstarting this process as well. It creates a Step Functions state machine that calls two instances of the AWS Glue Python Shell job you just created to complete the two tasks I outlined at the beginning of this post.

Launch Stack

For **BucketName**, paste the name of the script bucket created in the second AWS CloudFormation stack. For **GlueJobName**, type in the name of the job you just created. Leave the other information as default, as shown in the following screenshot. Launch the stack and wait for it to display **Create Complete**—this should take only a couple of minutes—before moving on to the next section.


CloudFormation > Stacks > Create stack

Step 1
Specify template

Step 2
Specify stack details

Step 3
Configure stack options

Step 4
Review

Specify stack details

Stack name

Stack name: blogsfstack

Stack name can include letters (A-Z and a-z), numbers (0-9), and dashes (-).

Parameters

Parameters are defined in your template and allow you to input custom values when you create or update a stack.

BucketName
Name of the S3 bucket with the .sql files
blogstack-blogbucket-1[redacted]

GlueJobName
Name of the target Glue Python Shell job
blog_rs_query

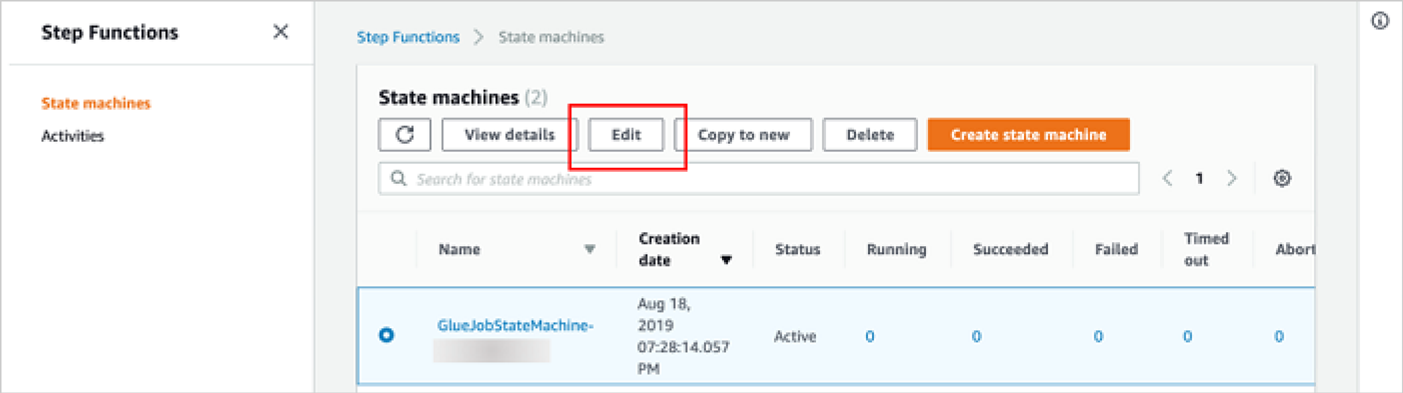
RedshiftDBName
Redshift database name
reviews

SecretsManagerSecret
Name of the Secrets Manager secret for the cluster
reviewssecret

Working with the Step Functions State Machine

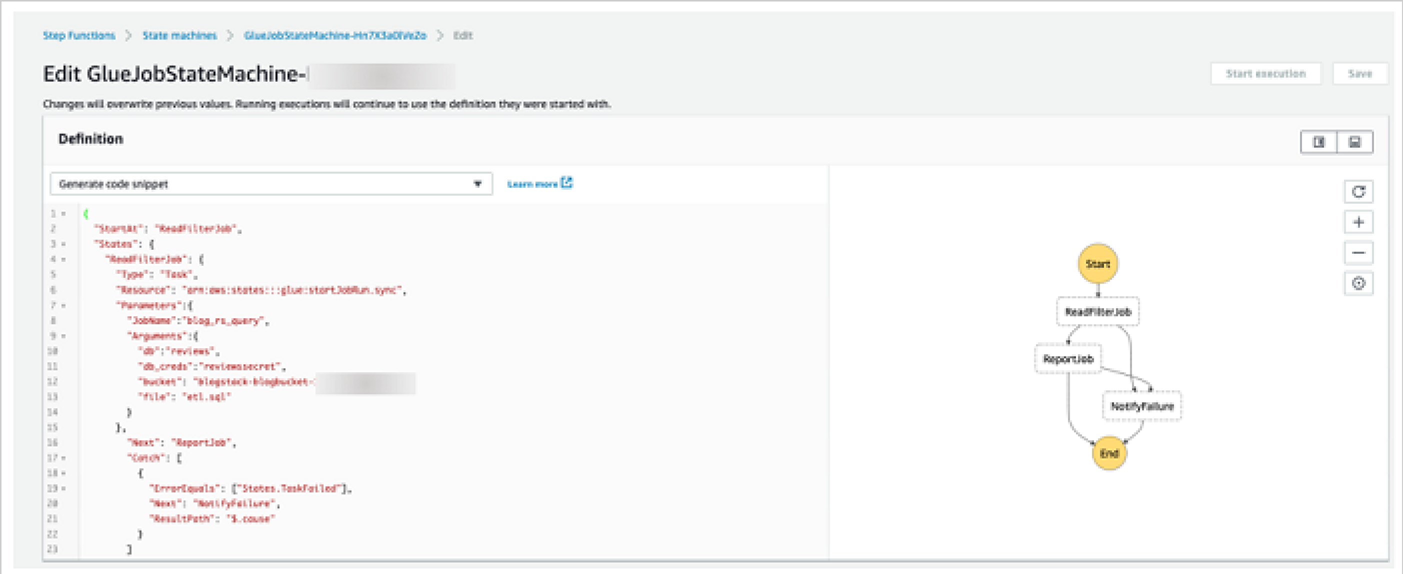
State Machines are made up of a series of steps, allowing you to stitch together services into robust ETL workflows. You can monitor each step of execution as it happens, which means you can identify and fix problems in your ETL workflow quickly, and even automatically.

Take a look at the state machine you just launched to get a better idea. Navigate to **Step Functions** in the AWS Console and look for a state machine with a name like *GlueJobStateMachine-######.* Choose **Edit** to view the state machine configuration, as shown in the following screenshot.



	Name	Creation date	Status	Running	Succeeded	Failed	Timed out	Abort
●	GlueJobStateMachine-[redacted]	Aug 18, 2019 07:28:14.057 PM	Active	0	0	0	0	0

It should look as it does in the following screenshot:


Step Functions > State machines > GlueJobStateMachine-Hn7X3a0lVeZo > Edit

Edit GlueJobStateMachine-[redacted]

Changes will overwrite previous values. Running executions will continue to use the definition they were started with.

```
{
  "StartAt": "ReadFilterJob",
  "States": {
    "ReadFilterJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::glue:startJobRun.sync",
      "Parameters":{
        "JobName":"blog_rs_query",
        "Arguments":{
          "db":"reviews",
          "db_creds":"reviewssecret",
          "bucket": "blogstack-blogbucket-[redacted]
          "file": "etl.sql"
        }
      },
        "Next": "ReportJob",
        "Catch": [
          {
            "ErrorEquals": ["States.TaskFailed"],
            "Next": "NotifyFailure",
            "ResultPath": "$.cause"
          }
        ]
```

As you can see, state machines are created using JSON templates made up of task definitions and workflow logic. You can run parallel tasks, catch errors, and even pause workflows and wait for manual callback to continue. The example I provide contains two tasks for running the SQL statements that complete the goals I outlined at the beginning of the post:

1. Load data from S3 using Redshift Spectrum
2. Transform and writing data back to S3

Each task contains basic error handling which, if caught, routes the workflow to an error notification task. This example is a simple one to show you how to build a basic workflow, but you can refer to the Step Functions documentation for an example of more complex workflows to help build a robust ETL pipeline. Step Functions also supports reusing modular components with Nested Workflows.

SQL Review

The state machine will retrieve and run the following SQL statements:

```
INSERT INTO reviews
SELECT marketplace, customer_id, review_id, product_id, product_parent, product_title, star_rating, helpful_votes, tota
FROM amzreviews.reviews
WHERE year > 2015;
```

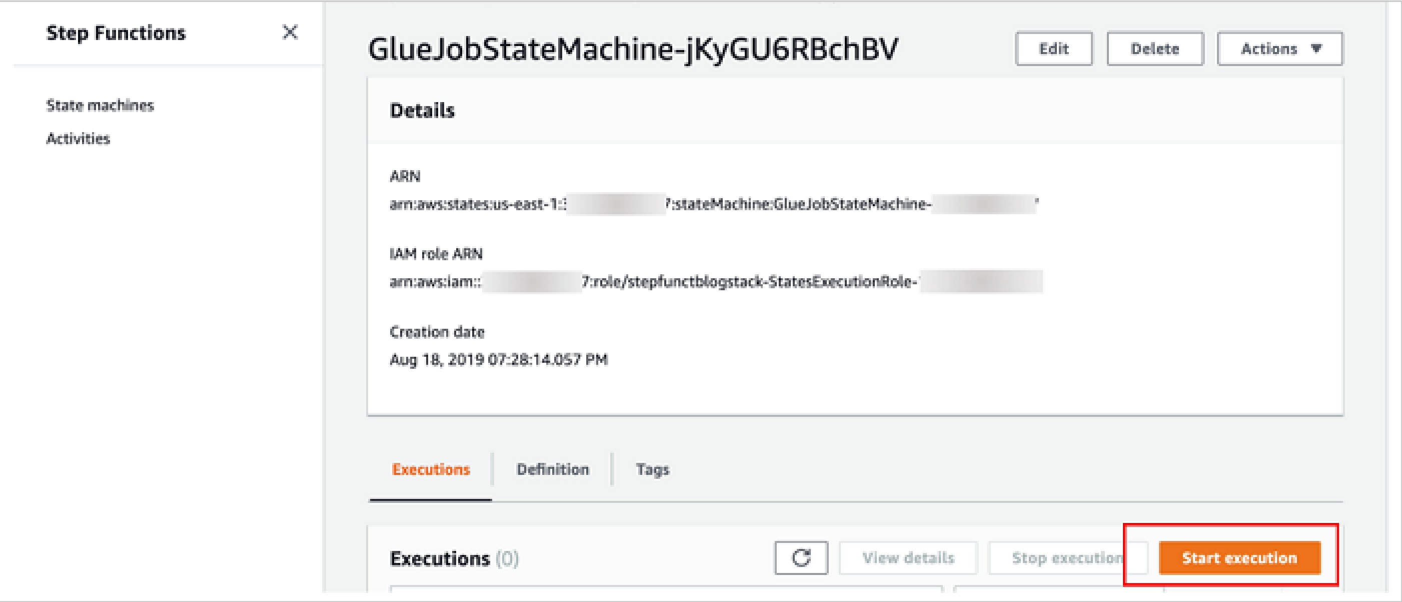
As I mentioned previously, Amazon Redshift Spectrum is a great way to run ETL using an INSERT INTO statement. This example is a simple load of the data as it is in S3, but keep in mind you can add more complex SQL statements to transform your data prior to loading.

```
UNLOAD ('SELECT marketplace, product_category, product_title, review_id, helpful_votes, AVG(star_rating) as average_sta
TO 's3://bucket/testunload/'
iam_role 'rolearn';
```

This statement groups reviews by product, ordered by number of helpful votes, and writes to Amazon S3 using UNLOAD.

State Machine execution

Now that everything is in order, start an execution. From the state machine main page select **Start an Execution**.



GlueJobStateMachine-jKyGU6RBchBV

Details

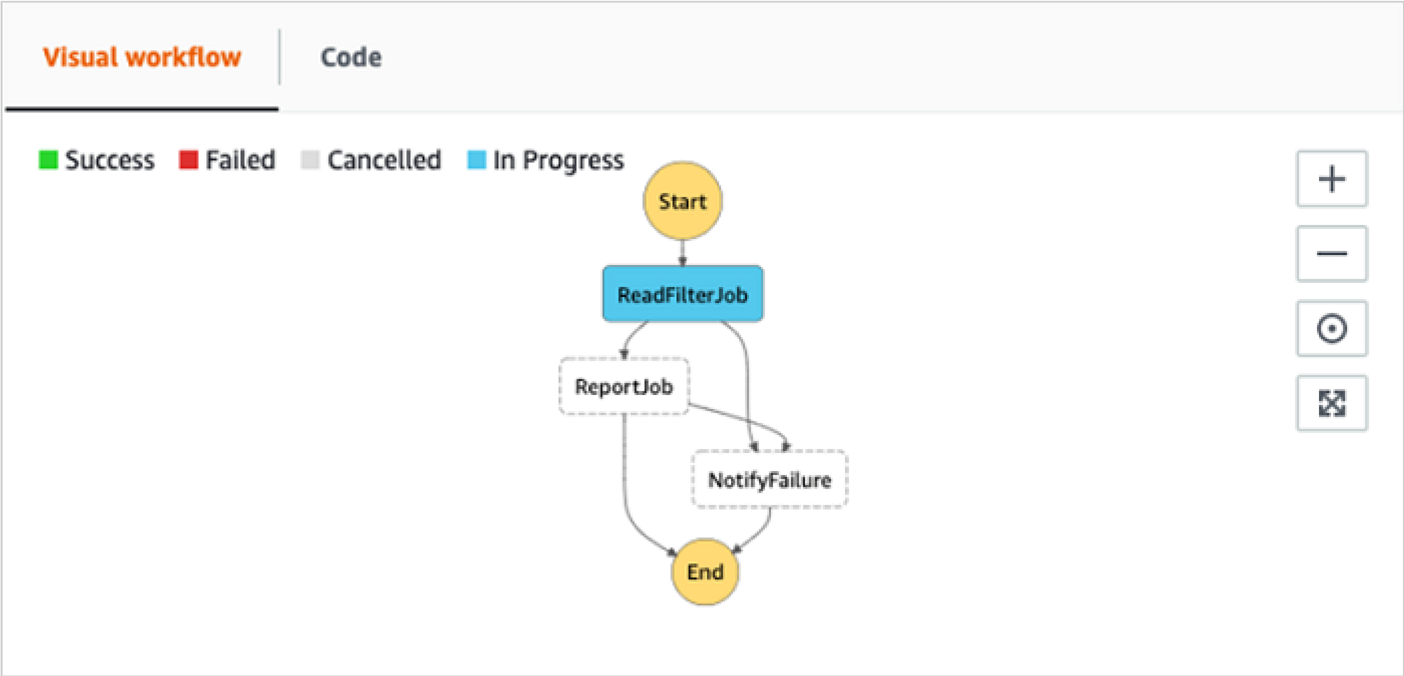
ARN
arn:aws:states:us-east-1:[redacted]:stateMachine:GlueJobStateMachine-[redacted]

IAM role ARN
arn:aws:iam::[redacted]:role/stepfunctblogstack-StatesExecutionRole-[redacted]

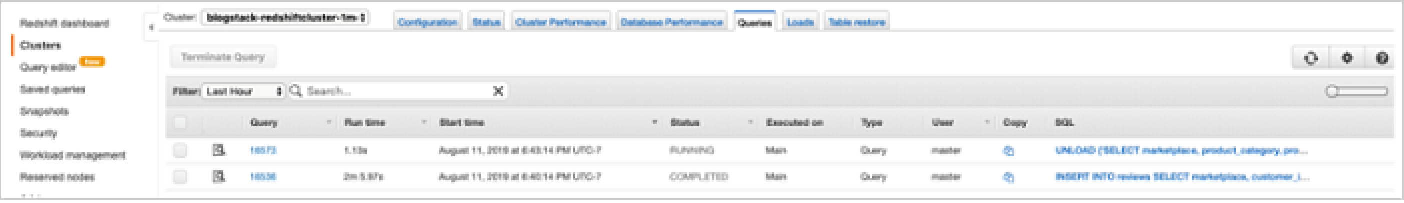
Creation date
Aug 18, 2019 07:28:14.057 PM

Executions (0)

Leave the defaults as they are and select **Start** to begin execution. Once execution begins you are taken to a visual workflow interface where you can follow the execution progress, as shown in the following screenshot.



Each of the queries takes a few minutes to run. In the meantime, you can watch the Amazon Redshift query logs to track the query progress in real time. These can be found by navigating to **Amazon Redshift** in the AWS Console, selecting your Amazon Redshift cluster, and then selecting the **Queries** tab, as shown in the following screenshot.

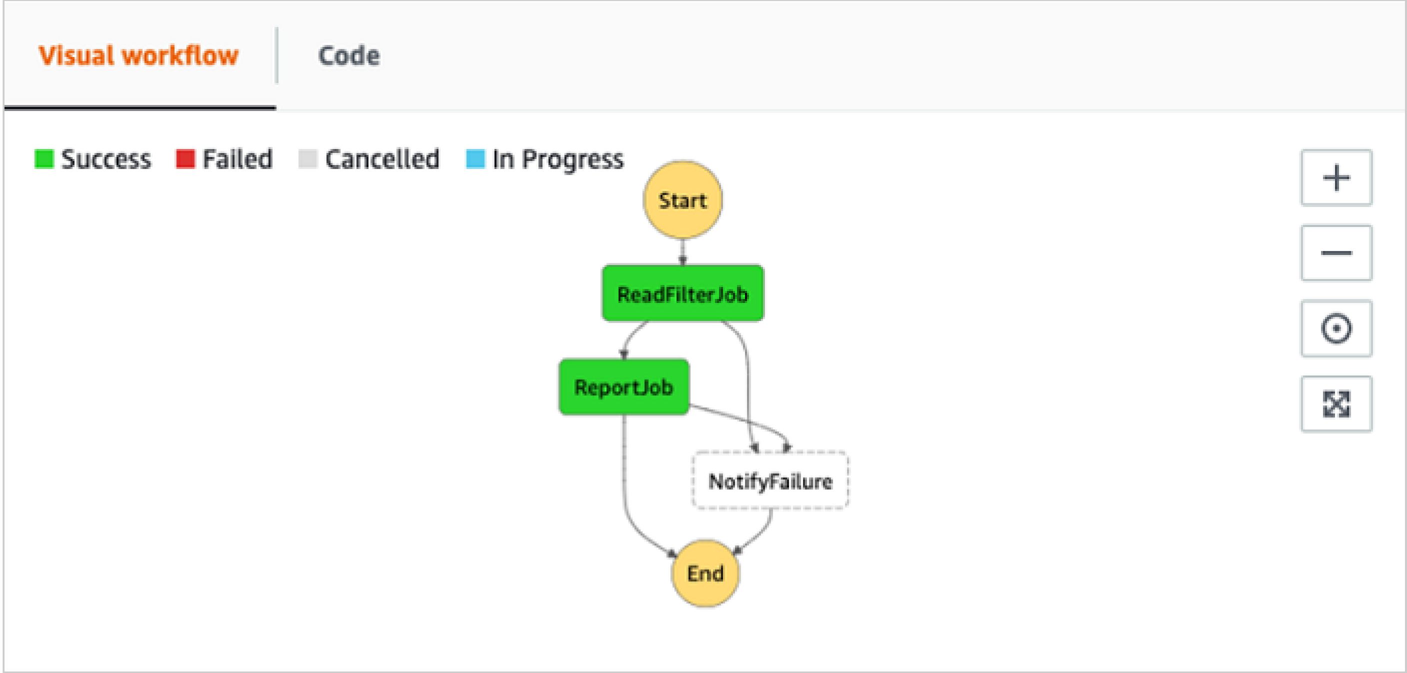


Cluster: blogstack-redshiftcluster-1m-

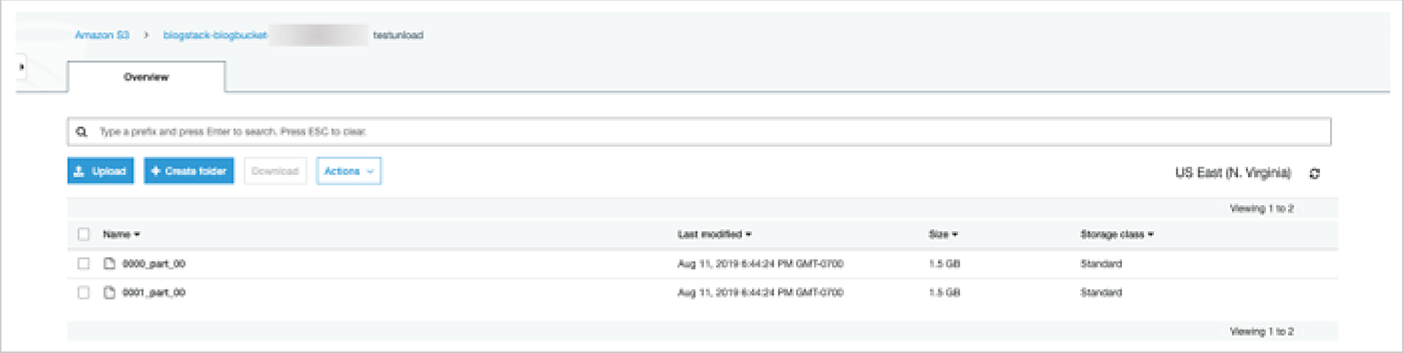
Filter: Last Hour

	Query	Run time	Start time	Status	Executed on	Type	User	Copy	SQL
□	16573	1.13s	August 11, 2019 at 6:43:14 PM UTC-7	RUNNING	Main	Query	master		UNLOAD ('SELECT marketplace, product_category, pro...
□	16536	2m 5.97s	August 11, 2019 at 6:40:14 PM UTC-7	COMPLETED	Main	Query	master		INSERT INTO reviews SELECT marketplace, customer_i...

Once you see **COMPLETED** for both queries, navigate back to the state machine execution. You should see success for each of the states, as shown in the following screenshot.



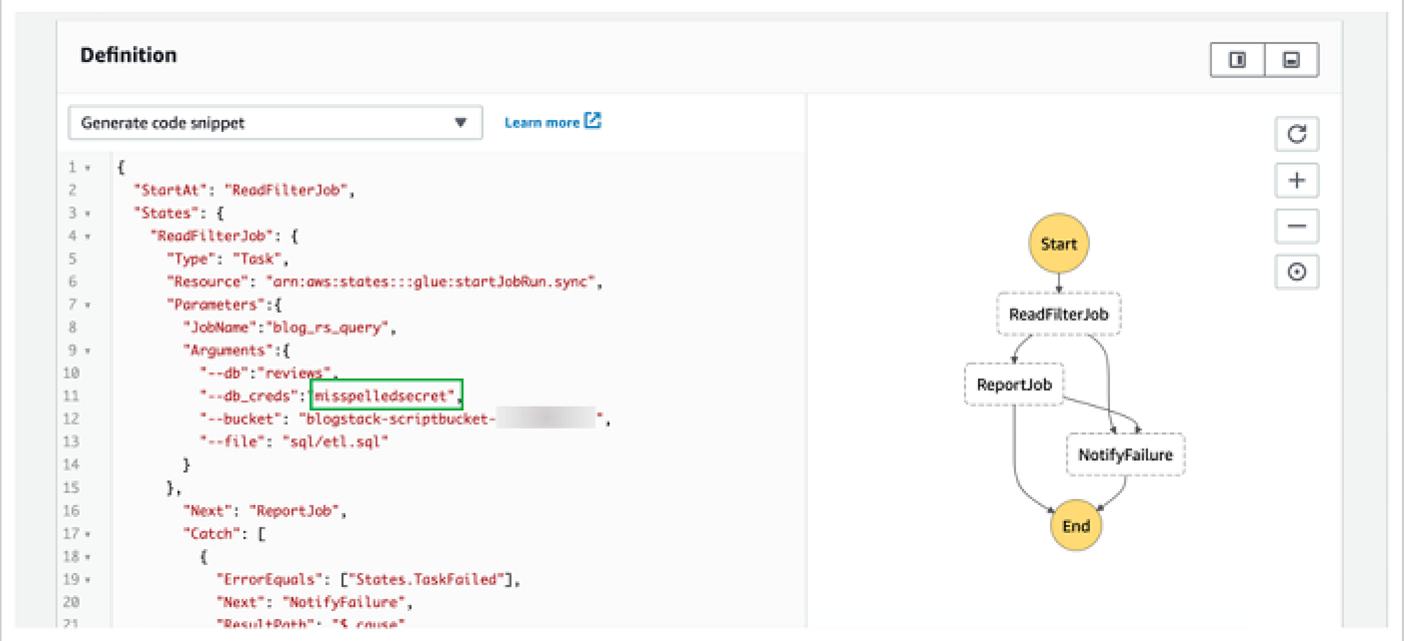
Next, navigate to the data bucket in the S3 AWS Console page (refer to the **DataBucket** in the **CloudFormation Resources** tab). If all went as planned, you should see a folder named *testunload* in the bucket with the unloaded data, as shown in the following screenshot.


Amazon S3 > blogstack-blogbucket-[redacted] testunload
Overview
Type a prefix and press Enter to search. Press ESC to clear.
Upload
Create folder
Download
Actions
US East (N. Virginia)
Viewing 1 to 2

	Name	Last modified	Size	Storage class
□	0000_part_00	Aug 11, 2019 6:44:24 PM GMT-0700	1.5 GB	Standard
□	0001_part_00	Aug 11, 2019 6:44:24 PM GMT-0700	1.5 GB	Standard

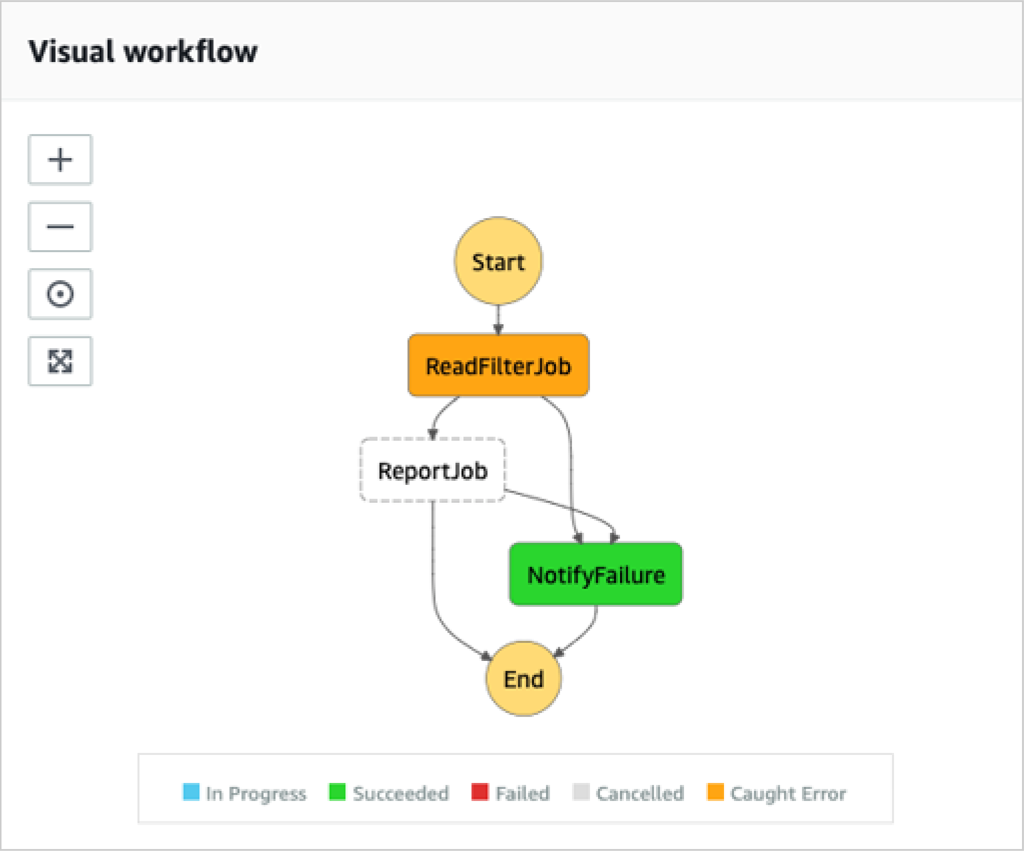
Inject Failure into Step Functions State Machine

Next, test the error handling component of the state machine by intentionally causing an error. An easy way to do this is to edit the state machine and misspell the name of the Secrets Manager secret in the **ReadFilterJob** task, as shown in the following screenshot.

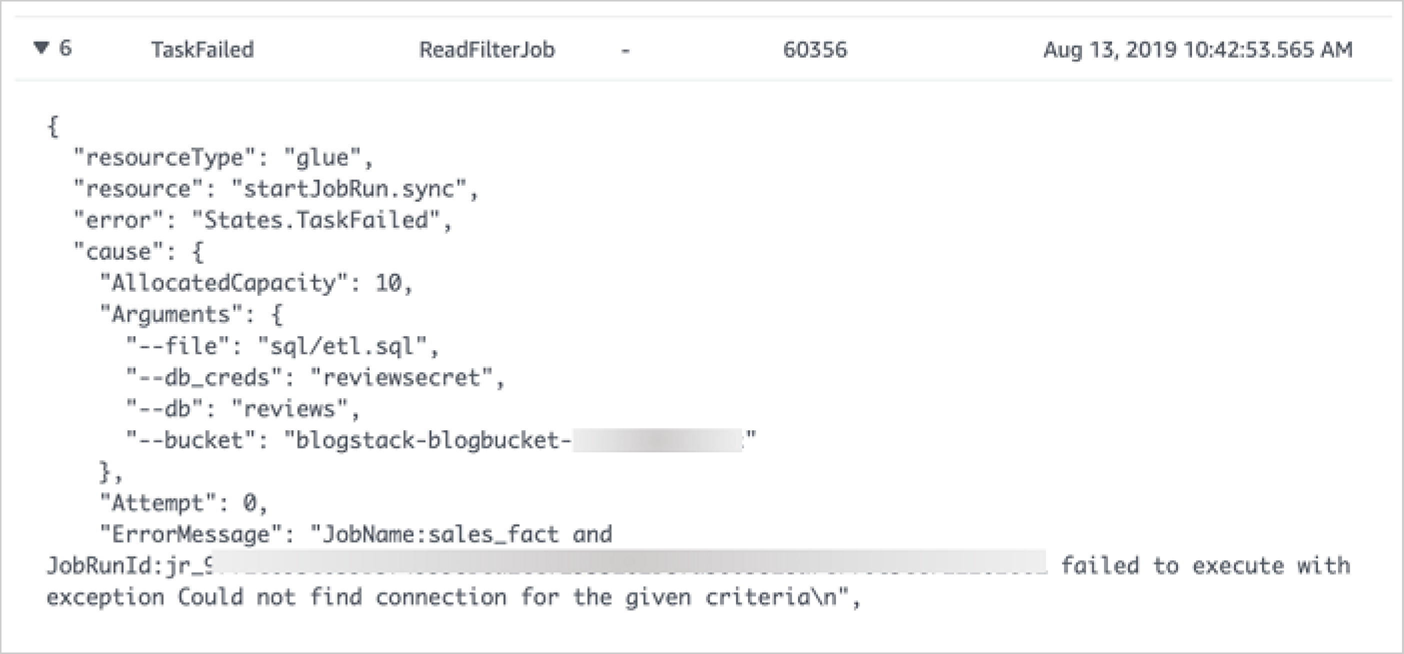


```
{
  "StartAt": "ReadFilterJob",
  "States": {
    "ReadFilterJob": {
      "Type": "Task",
      "Resource": "arn:aws:states:::glue:startJobRun.sync",
      "Parameters":{
        "JobName":"blog_rs_query",
        "Arguments":{
          "--db":"reviews",
          "--db_creds":"misspelledsecret",
          "--bucket": "blogstack-scriptbucket-[redacted]",
          "--file": "sql/etl.sql"
        }
      },
        "Next": "ReportJob",
        "Catch": [
          {
            "ErrorEquals": ["States.TaskFailed"],
            "Next": "NotifyFailure",
            "ResultPath": "$.cause"
```

If you want the error output sent to you, optionally subscribe to the error notification SNS Topic. Start another state machine execution as you did previously. This time the workflow should take the path toward the *NotifyFailure* task, as shown in the following screenshot. If you subscribed to the SNS Topic associated with it, you should receive a message shortly thereafter.


Visual workflow
+
−
Start
ReadFilterJob
ReportJob
NotifyFailure
End
🟦 In Progress 🟩 Succeeded 🟥 Failed ⬜ Cancelled 🟧 Caught Error

The state machine logs will show the error in more detail, as shown in the following screenshot.



```
▼ 6    TaskFailed    ReadFilterJob    -    60356    Aug 13, 2019 10:42:53.565 AM

{
  "resourceType": "glue",
  "resource": "startJobRun.sync",
  "error": "States.TaskFailed",
  "cause": {
    "AllocatedCapacity": 10,
    "Arguments": {
      "--file": "sql/etl.sql",
      "--db_creds": "reviewsecret",
      "--db": "reviews",
      "--bucket": "blogstack-blogbucket-[redacted]"
    },
    "Attempt": 0,
    "ErrorMessage": "JobName:sales_fact and
JobRunId:jr_[redacted] failed to execute with
exception Could not find connection for the given criteria\n",
```

Conclusion

In this post I demonstrated how you can orchestrate Amazon Redshift-based ETL using serverless AWS Step Functions and AWS Glue Python Shells jobs. As I mentioned in the introduction, the concepts can also be more generally applied to other SQL-based ETL, so use them to start building your own SQL-based ETL pipelines today!

About the Author

Ben Romano is a Data Lab solution architect at AWS. Ben helps our customers architect and build data and analytics prototypes in just four days in the AWS Data Lab.

TAGS: Amazon Redshift, AWS Glue, AWS Step Functions