aws

Amazon EMR Serverless User Guide

Amazon EMR

Amazon EMR: Amazon EMR Serverless User Guide

Table of Contents

What is Amazon EMR Serverless?

Amazon EMR Serverless is a deployment option for Amazon EMR that provides a serverless runtime environment. This simplifies the operation of analytics applications that use the latest open-source frameworks, such as Apache Spark and Apache Hive. With EMR Serverless, you don't have to configure, optimize, secure, or operate clusters to run applications with these frameworks.

EMR Serverless helps you avoid over- or under-provisioning resources for your data processing jobs. EMR Serverless automatically determines the resources that the application needs, gets these resources to process your jobs, and releases the resources when the jobs finish. For use cases where applications need a response within seconds, such as interactive data analysis, you can pre-initialize the resources that the application needs when you create the application.

With EMR Serverless, you'll continue to get the benefits of Amazon EMR, such as open source compatibility, concurrency, and optimized runtime performance for popular frameworks.

EMR Serverless is suitable for customers who want ease in operating applications using open source frameworks. It offers quick job startup, automatic capacity management, and straightforward cost controls.

Concepts

In this section, we cover EMR Serverless terms and concepts that appear throughout our EMR Serverless User Guide.

Release version

An Amazon EMR *release* is a set of open-source applications from the big data ecosystem. Each release includes different big data applications, components, and features that you select for EMR Serverless to deploy and configure so that they can run your applications. When you create an application, you must specify its release version. Choose the Amazon EMR release version and the open source framework version that you want to use in your application. To learn more about pre-release versions, see Amazon EMR Serverless release versions.

Application

With EMR Serverless, you can create one or more EMR Serverless applications that use open source analytics frameworks. To create an application, you must specify the following attributes:

- The Amazon EMR release version for the open source framework version that you want to use. To determine your release version, see Amazon EMR Serverless release versions.
- The specific runtime that you want your application to use, such as Apache Spark or Apache Hive.

After you create an application, you can submit data-processing jobs or interactive requests to your application.

Each EMR Serverless application runs on a secure Amazon Virtual Private Cloud (VPC) strictly apart from other applications. Additionally, you can use AWS Identity and Access Management (IAM) policies to define which users and roles can access the application. You can also specify limits to control and track usage costs incurred by the application.

Consider creating multiple applications when you need to do the following:

- Use different open source frameworks
- Use different versions of open source frameworks for different use cases
- Perform A/B testing when upgrading from one version to another
- Maintain separate logical environments for test and production scenarios
- Provide separate logical environments for different teams with independent cost controls and usage tracking
- Separate different line-of-business applications

EMR Serverless is a Regional service that simplifies how workloads run across multiple Availability Zones in a Region. To learn more about how to use applications with EMR Serverless, see Interact with and configure an EMR Serverless application.

Job run

A *job run* is a request submitted to an EMR Serverless application that the application asychronously executes and tracks through completion. Examples of jobs include a HiveQL query that you submit to an Apache Hive application, or a PySpark data processing script that you submit to an Apache Spark application. When you submit a job, you must specify a runtime role, authored in IAM, that the job uses to access AWS resources, such as Amazon S3 objects. You can submit multiple job run requests to an application, and each job run can use a different runtime role to access AWS resources. An EMR Serverless application starts executing jobs as soon as it receives

them and runs multiple job requests concurrently. To learn more about how EMR Serverless runs jobs, see Running jobs.

Workers

An EMR Serverless application internally uses *workers* to execute your workloads. The default sizes of these workers are based on your application type and Amazon EMR release version. When you schedule a job run, you can override these sizes.

When you submit a job, EMR Serverless computes the resources that the application needs for the job and schedules workers. EMR Serverless breaks down your workloads into tasks, downloads images, provisions and sets up workers, and decommissions them when the job finishes. EMR Serverless automatically scales workers up or down based on the workload and parallelism required at every stage of the job. This automatic scaling removes the need for you to estimate the number of workers that the application needs to run your workloads.

Pre-initialized capacity

EMR Serverless provides a *pre-initialized capacity* feature that keeps workers initialized and ready to respond in seconds. This capacity effectively creates a warm pool of workers for an application. To configure this feature for each application, set the `initial-capacity` parameter of an application. When you configure pre-initialized capacity, jobs can start immediately so that you can implement iterative applications and time-sensitive jobs. To learn more about pre-initialized workers, see Configuring an application when working with EMR Serverless.

EMR Studio

EMR Studio is the user console that you can use to manage your EMR Serverless applications. If an EMR Studio doesn't exist in your account when you create your first EMR Serverless application, we automatically create one for you. You can access EMR Studio either from the Amazon EMR console, or you can turn on federated access from your identity provider (IdP) through IAM or IAM Identity Center. When you do this, users can access Studio and manage EMR Serverless applications without direct access to the Amazon EMR console. To learn more about how EMR Serverless applications works with EMR Studio, see Creating an EMR Serverless application from the EMR Studio console and Running jobs from the EMR Studio console.

Prerequisites for getting started with EMR Serverless

This section describes the administrative prerequisites for running EMR Serverless. These include account configuration and permissions management.

Topics

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Grant permissions

In production environments, we recommend that you use finer-grained policies. For examples of such policies, see User access policy examples for EMR Serverless. To learn more about access management, see Access management for AWS resources in the IAM User Guide.

For users who need to get started with EMR Serverless in a sandbox environment, use a policy similar to the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRStudioCreate",
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:CreateStudioPresignedUrl",
                "elasticmapreduce:DescribeStudio",
                "elasticmapreduce:CreateStudio",
                "elasticmapreduce:ListStudios"
            ],
            "Resource": "*"
        },
        {
            "Sid": "EMRServerlessFullAccess",
            "Effect": "Allow",
            "Action": [
                "emr-serverless:*"
            ],
            "Resource": "*"
        },
```

```
        {
            "Sid": "AllowEC2ENICreationWithEMRTags",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:network-interface/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:CalledViaLast": "ops.emr-serverless.amazonaws.com"
                }
            }
        },
        {
            "Sid": "AllowEMRServerlessServiceLinkedRoleCreation",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/*"
        }
    ]
}
```

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*.

Grant programmatic access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM

Which user needs programmatic access?	To	By
		user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

Install and configure the AWS CLI

If you want to use EMR Serverless APIs, you must install the latest version of the AWS Command Line Interface (AWS CLI). You don't need the AWS CLI to use EMR Serverless from the EMR Studio console, and you can get started without the CLI by following the steps in Getting started with EMR Serverless from the console.

To set up the AWS CLI

1. To install the latest version of the AWS CLI for macOS, Linux, or Windows, see Installing or updating the latest version of the AWS CLI.
2. To configure the AWS CLI and secure setup of your access to AWS services, including EMR Serverless, see Quick configuration with `aws configure`.
3. To verify the setup, enter the following DataBrew command at the command prompt.

   ```
   aws emr-serverless help
   ```

   AWS CLI commands use the default AWS Region from your configuration, unless you set it with a parameter or a profile. To set your AWS Region with a parameter, you can add the `--region` parameter to each command.

To set your AWS Region with a profile, first add a named profile in the `~/.aws/config` file or the `%UserProfile%/.aws/config` file (for Microsoft Windows). Follow the steps in Named profiles for the AWS CLI. Next, set your AWS Region and other settings with a command similar to the one in the following example.

```
[profile emr-serverless]
aws_access_key_id = ACCESS-KEY-ID-OF-IAM-USER
aws_secret_access_key = SECRET-ACCESS-KEY-ID-OF-IAM-USER
region = us-east-1
output = text
```

Open the console

Most of the console-oriented topics in this section start from the Amazon EMR console. If you aren't already signed in to your AWS account, sign in, then open the Amazon EMR console and continue to the next section to continue getting started with Amazon EMR.

Getting started with Amazon EMR Serverless

This tutorial helps you get started with EMR Serverless when you deploy a sample Spark or Hive workload. You'll create, run, and debug your own application. We show default options in most parts of this tutorial.

Before you launch an EMR Serverless application, complete the following tasks.

Topics

Grant permissions to use EMR Serverless

To use EMR Serverless, you need a user or IAM role with an attached policy that grants permissions for EMR Serverless. To create a user and attach the appropriate policy to that user, follow the instructions in Grant permissions.

Prepare storage for EMR Serverless

In this tutorial, you'll use an S3 bucket to store output files and logs from the sample Spark or Hive workload that you'll run using an EMR Serverless application. To create a bucket, follow the instructions in Creating a bucket in the *Amazon Simple Storage Service Console User Guide*. Replace any further reference to `amzn-s3-demo-bucket` with the name of the newly created bucket.

Create an EMR Studio to run interactive workloads

If you want to use EMR Serverless to execute interactive queries through notebooks that are hosted in EMR Studio, you need to specify an S3 bucket and the minimum service role for EMR Serverless to create a Workspace. For steps to get set up, see Set up an EMR Studio in the *Amazon EMR*

Management Guide. For more information on interactive workloads, see Run interactive workloads with EMR Serverless through EMR Studio.

Create a job runtime role

Job runs in EMR Serverless use a runtime role that provides granular permissions to specific AWS services and resources at runtime. In this tutorial, a public S3 bucket hosts the data and scripts. The bucket *amzn-s3-demo-bucket* stores the output.

To set up a job runtime role, first create a runtime role with a trust policy so that EMR Serverless can use the new role. Next, attach the required S3 access policy to that role. The following steps guide you through the process.

Console

1. Navigate to the IAM console at https://console.aws.amazon.com/iam/.
2. In the left navigation pane, choose **Roles**.
3. Choose **Create role**.
4. For role type, choose **Custom trust policy** and paste the following trust policy. This allows jobs submitted to your Amazon EMR Serverless applications to access other AWS services on your behalf.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "emr-serverless.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

5. Choose **Next** to navigate to the **Add permissions** page, then choose **Create policy**.
6. The **Create policy** page opens on a new tab. Paste the policy JSON below.

Important

Replace *amzn-s3-demo-bucket* in the policy below with the actual bucket name created in Prepare storage for EMR Serverless. This is a basic policy for S3 access. For more job runtime role examples, see Job runtime roles for Amazon EMR Serverless.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadAccessForEMRSamples",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*.elasticmapreduce",
                "arn:aws:s3:::*.elasticmapreduce/*"
            ]
        },
        {
            "Sid": "FullAccessToOutputBucket",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket",
                "arn:aws:s3:::amzn-s3-demo-bucket/*"
            ]
        },
        {
            "Sid": "GlueCreateAndReadDataCatalog",
            "Effect": "Allow",
            "Action": [
```

```
                "glue:GetDatabase",
                "glue:CreateDatabase",
                "glue:GetDataBases",
                "glue:CreateTable",
                "glue:GetTable",
                "glue:UpdateTable",
                "glue:DeleteTable",
                "glue:GetTables",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:GetUserDefinedFunctions"
            ],
            "Resource": ["*"]
        }
    ]
}
```

7. On the **Review policy** page, enter a name for your policy, such as `EMRServerlessS3AndGlueAccessPolicy`.
8. Refresh the **Attach permissions policy** page, and choose `EMRServerlessS3AndGlueAccessPolicy`.
9. In the **Name, review, and create** page, for **Role name**, enter a name for your role, for example, `EMRServerlessS3RuntimeRole`. To create this IAM role, choose **Create role**.

CLI

1. Create a file named `emr-serverless-trust-policy.json` that contains the trust policy to use for the IAM role. The file should contain the following policy.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "EMRServerlessTrustPolicy",
        "Action": "sts:AssumeRole",
        "Effect": "Allow",
        "Principal": {
            "Service": "emr-serverless.amazonaws.com"
        }
    }]
```

```
}
```

2. Create an IAM role named `EMRServerlessS3RuntimeRole`. Use the trust policy that you created in the previous step.

```
aws iam create-role \
    --role-name EMRServerlessS3RuntimeRole \
    --assume-role-policy-document file://emr-serverless-trust-policy.json
```

Note the ARN in the output. You use the ARN of the new role during job submission, referred to after this as the *`job-role-arn`*.

3. Create a file named `emr-sample-access-policy.json` that defines the IAM policy for your workload. This provides read access to the script and data stored in public S3 buckets and read-write access to *`amzn-s3-demo-bucket`*.

> **Important**
>
> Replace *`amzn-s3-demo-bucket`* in the policy below with the actual bucket name created in Prepare storage for EMR Serverless.. This is a basic policy for AWS Glue and S3 access. For more job runtime role examples, see Job runtime roles for Amazon EMR Serverless.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadAccessForEMRSamples",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*.elasticmapreduce",
                "arn:aws:s3:::*.elasticmapreduce/*"
            ]
        },
        {
            "Sid": "FullAccessToOutputBucket",
```

```
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket",
                "arn:aws:s3:::amzn-s3-demo-bucket/*"
            ]
        },
        {
            "Sid": "GlueCreateAndReadDataCatalog",
            "Effect": "Allow",
            "Action": [
                "glue:GetDatabase",
                "glue:CreateDatabase",
                "glue:GetDataBases",
                "glue:CreateTable",
                "glue:GetTable",Understanding default application behavior,
 including auto-start and auto-stop, as well as maximum capacity and worker
 configurations for configuring an application with &EMRServerless;.
                "glue:UpdateTable",
                "glue:DeleteTable",
                "glue:GetTables",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:GetUserDefinedFunctions"
            ],
            "Resource": ["*"]
        }
    ]
}
```

4. Create an IAM policy named `EMRServerlessS3AndGlueAccessPolicy` with the policy file that you created in **Step 3**. Take note of the ARN in the output, as you will use the ARN of the new policy in the next step.

```
aws iam create-policy \
    --policy-name EMRServerlessS3AndGlueAccessPolicy \
```

```
    --policy-document file://emr-sample-access-policy.json
```

Note the new policy's ARN in the output. You'll substitute it for *policy-arn* in the next step.

5. Attach the IAM policy `EMRServerlessS3AndGlueAccessPolicy` to the job runtime role `EMRServerlessS3RuntimeRole`.

```
aws iam attach-role-policy \
    --role-name EMRServerlessS3RuntimeRole \
    --policy-arn policy-arn
```

Getting started with EMR Serverless from the console

This section describes working with EMR Serverless, including creating an EMR Studio. It also describes how to submit job runs and view logs.

Steps to complete

- Step 1: Create an EMR Serverless application
- Step 2: Submit a job run or interactive workload
- Step 3: View application UI and logs
- Step 4: Clean up

Step 1: Create an EMR Serverless application

Create a new application with EMR Serverless as follows.

1. Sign in to the AWS Management Console and open the Amazon EMR console at https://console.aws.amazon.com/emr.
2. In the left navigation pane, choose **EMR Serverless** to navigate to the EMR Serverless landing page.
3. To create or manage EMR Serverless applications, you need the EMR Studio UI.
   - If you already have an EMR Studio in the AWS Region where you want to create an application, then select **Manage applications** to navigate to your EMR Studio, or select the studio that you want to use.

- If you don't have an EMR Studio in the AWS Region where you want to create an application, choose **Get started** and then Choose **Create and launch Studio**. EMR Serverless creates a EMR Studio for you so that you can create and manage applications.

4. In the **Create studio** UI that opens in a new tab, enter the name, type, and release version for your application. If you only want to run batch jobs, select **Use default settings for batch jobs only**. For interactive workloads, select **Use default settings for interactive workloads**. You can also run batch jobs on interactive-enabled applications with this option. If you need to, you can change these settings later.

   For more information, see Create a studio.

5. Select **Create application** to create your first application.

Continue to the next section Step 2: Submit a job run or interactive workload to submit a job run or interactive workload.

Step 2: Submit a job run or interactive workload

Spark job run

In this tutorial, we use a PySpark script to compute the number of occurrences of unique words across multiple text files. A public, read-only S3 bucket stores both the script and the dataset.

To run a Spark job

1. Upload the sample script `wordcount.py` into your new bucket with the following command.

```
aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/
scripts/wordcount.py s3://amzn-s3-demo-bucket/scripts/
```

2. Completing Step 1: Create an EMR Serverless application takes you to the **Application details** page in EMR Studio. There, choose the **Submit job** option.
3. On the **Submit job** page, complete the following.
   - In the **Name** field, enter the name that you want to call your job run.
   - In the **Runtime role** field, enter the name of the role that you created in Create a job runtime role.

- In the **Script location** field, enter `s3://amzn-s3-demo-bucket/scripts/wordcount.py` as the S3 URI.
- In the **Script arguments** field, enter `["s3://amzn-s3-demo-bucket/emr-serverless-spark/output"]`.
- In the **Spark properties** section, choose **Edit as text** and enter the following configurations.

```
--conf spark.executor.cores=1 --conf spark.executor.memory=4g --
conf spark.driver.cores=1 --conf spark.driver.memory=4g --conf
 spark.executor.instances=1
```

4. To start the job run, choose **Submit job** .
5. In the **Job runs** tab, you should see your new job run with a **Running** status.

Hive job run

In this part of the tutorial, we create a table, insert a few records, and run a count aggregation query. To run the Hive job, first create a file that contains all Hive queries to run as part of single job, upload the file to S3, and specify this S3 path when starting the Hive job.

To run a Hive job

1. Create a file called `hive-query.ql` that contains all the queries that you want to run in your Hive job.

```
create database if not exists emrserverless;
use emrserverless;
create table if not exists test_table(id int);
drop table if exists Values__Tmp__Table__1;
insert into test_table values (1),(2),(2),(3),(3),(3);
select id, count(id) from test_table group by id order by id desc;
```

2. Upload `hive-query.ql` to your S3 bucket with the following command.

```
aws s3 cp hive-query.ql s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.ql
```

3. Completing Step 1: Create an EMR Serverless application takes you to the **Application details** page in EMR Studio. There, choose the **Submit job** option.

4. On the **Submit job** page, complete the following.

   - In the **Name** field, enter the name that you want to call your job run.
   - In the **Runtime role** field, enter the name of the role that you created in Create a job runtime role.
   - In the **Script location** field, enter `s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-query.ql` as the S3 URI.
   - In the **Hive properties** section, choose **Edit as text**, and enter the following configurations.

     ```
     --hiveconf hive.log.explain.output=false
     ```

   - In the **Job configuration** section, choose **Edit as JSON**, and enter the following JSON.

     ```
     {
        "applicationConfiguration":
        [{
            "classification": "hive-site",
               "properties": {
                   "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-
     serverless-hive/hive/scratch",
                   "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
     serverless-hive/hive/warehouse",
                   "hive.driver.cores": "2",
                   "hive.driver.memory": "4g",
                   "hive.tez.container.size": "4096",
                   "hive.tez.cpu.vcores": "1"
               }
         }]
     }
     ```

5. To start the job run, choose **Submit job**.
6. In the **Job runs** tab, you should see your new job run with a **Running** status.

Interactive workload

With Amazon EMR 6.14.0 and higher, you can use notebooks that are hosted in EMR Studio to run interactive workloads for Spark in EMR Serverless. For more information including permissions and prerequisites, see Run interactive workloads with EMR Serverless through EMR Studio.

Once you've created your application and set up the required permissions, use the following steps to run an interactive notebook with EMR Studio:

1. Navigate to the **Workspaces** tab in EMR Studio. If you still need to configure an Amazon S3 storage location and EMR Studio service role, select the **Configure studio** button in the banner at the top of the screen.
2. To access a notebook, select a Workspace or create a new Workspace. Use **Quick launch** to open your Workspace in a new tab.
3. Go to the newly opened tab. Select the **Compute** icon from the left navigation. Select EMR Serverless as the **Compute type**.
4. Select the interactive-enabled application that you created in the previous section.
5. In the **Runtime role** field, enter the name of the IAM role that your EMR Serverless application can assume for the job run. To learn more about runtime roles, see Job runtime roles in the *Amazon EMR Serverless User Guide*.
6. Select **Attach**. This may take up to a minute. The page will refresh when attached.
7. Pick a kernel and start a notebook. You can also browse example notebooks on EMR Serverless and copy them to your Workspace. To access the example notebooks, navigate to the **`{...}`** menu in the left navigation and browse through notebooks that have `serverless` in the notebook file name.
8. In the notebook, you can access the driver log link and a link to the Apache Spark UI, a real-time interface that provides metrics to monitor your job. For more information, see Monitoring EMR Serverless applications and jobs in the *Amazon EMR Serverless User Guide*.

When you attach an application to an Studio workspace, the application start triggers automatically if it's not already running. You can also pre-start the application and keep it ready before you attach it to the workspace.

Step 3: View application UI and logs

To view the application UI, first identify the job run. An option for **Spark UI** or **Hive Tez UI** is available in the first row of options for that job run, based on the job type. Select the appropriate option.

If you chose the Spark UI, choose the **Executors** tab to view the driver and executors logs. If you chose the Hive Tez UI, choose the **All Tasks** tab to view the logs.

Once the job run status shows as **Success**, you can view the output of the job in your S3 bucket.

Step 4: Clean up

While the application you created should auto-stop after 15 minutes of inactivity, we still recommend that you release resources that you don't intend to use again.

To delete the application, navigate to the **List applications** page. Select the application that you created and choose **Actions** → **Stop** to stop the application. After the application is in the STOPPED state, select the same application and choose **Actions** → **Delete**.

For more examples of running Spark and Hive jobs, see Using Spark configurations when you run EMR Serverless jobs and Using Hive configurations when you run EMR Serverless jobs.

Getting started from the AWS CLI

Get started with EMR Serverless from the AWS CLI with commands to create an application, run jobs, check job run output, and delete your resources.

Step 1: Create an EMR Serverless application

Use the `emr-serverless create-application` command to create your first EMR Serverless application. You need to specify the application type and the the Amazon EMR release label associated with the application version you want to use. The name of the application is optional.

Spark

To create a Spark application, run the following command.

```
aws emr-serverless create-application \
    --release-label emr-6.6.0 \
    --type "SPARK" \
    --name my-application
```

Hive

To create a Hive application, run the following command.

```
aws emr-serverless create-application \
    --release-label emr-6.6.0 \
    --type "HIVE" \
```

```
    --name my-application
```

Note the application ID returned in the output. You'll use the ID to start the application and during job submission, referred to after this as the *application-id*.

Before you move on to Step 2: Submit a job run to your EMR Serverless application, make sure that your application has reached the CREATED state with the `get-application` API.

```
aws emr-serverless get-application \
    --application-id application-id
```

EMR Serverless creates workers to accommodate your requested jobs. By default, these are created on demand, but you can also specify a pre-initialized capacity by setting the `initialCapacity` parameter when you create the application. You can also limit the total maximum capacity that an application can use with the `maximumCapacity` parameter. To learn more about these options, see Configuring an application when working with EMR Serverless.

Step 2: Submit a job run to your EMR Serverless application

Now your EMR Serverless application is ready to run jobs.

Spark

In this step, we use a PySpark script to compute the number of occurrences of unique words across multiple text files. A public, read-only S3 bucket stores both the script and the dataset. The application sends the output file and the log data from the Spark runtime to `/output` and `/logs` directories in the S3 bucket that you created.

To run a Spark job

1. Use the following command to copy the sample script we will run into your new bucket.

   ```
   aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/
   scripts/wordcount.py s3://amzn-s3-demo-bucket/scripts/
   ```

2. In the following command, substitute *application-id* with your application ID. Substitute *job-role-arn* with the runtime role ARN you created in Create a job runtime role. Substitute *job-run-name* with the name you want to call your job run. Replace all *amzn-s3-demo-bucket* strings with the Amazon S3 bucket that you created, and add /

`output` to the path. This creates a new folder in your bucket where EMR Serverless can copy the output files of your application.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --name job-run-name \
    --job-driver '{
        "sparkSubmit": {
          "entryPoint": "s3://amzn-s3-demo-bucket/scripts/wordcount.py",
          "entryPointArguments": ["s3://amzn-s3-demo-bucket/emr-serverless-
spark/output"],
          "sparkSubmitParameters": "--conf spark.executor.cores=1
 --conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf
 spark.driver.memory=4g --conf spark.executor.instances=1"
        }
    }'
```

3. Note the job run ID returned in the output . Replace *job-run-id* with this ID in the following steps.

Hive

In this tutorial, we create a table, insert a few records, and run a count aggregation query. To run the Hive job, first create a file that contains all Hive queries to run as part of single job, upload the file to S3, and specify this S3 path when you start the Hive job.

To run a Hive job

1. Create a file called `hive-query.ql` that contains all the queries that you want to run in your Hive job.

```
create database if not exists emrserverless;
use emrserverless;
create table if not exists test_table(id int);
drop table if exists Values__Tmp__Table__1;
insert into test_table values (1),(2),(2),(3),(3),(3);
select id, count(id) from test_table group by id order by id desc;
```

2. Upload `hive-query.ql` to your S3 bucket with the following command.

```
aws s3 cp hive-query.ql s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.ql
```

3. In the following command, substitute *application-id* with your own application ID. Substitute *job-role-arn* with the runtime role ARN you created in Create a job runtime role. Replace all *amzn-s3-demo-bucket* strings with the Amazon S3 bucket that you created, and add `/output` and `/logs` to the path. This creates new folders in your bucket, where EMR Serverless can copy the output and log files of your application.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "hive": {
          "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.ql",
          "parameters": "--hiveconf hive.log.explain.output=false"
        }
    }' \
    --configuration-overrides '{
      "applicationConfiguration": [{
        "classification": "hive-site",
          "properties": {
            "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-serverless-
hive/hive/scratch",
            "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/warehouse",
            "hive.driver.cores": "2",
            "hive.driver.memory": "4g",
            "hive.tez.container.size": "4096",
            "hive.tez.cpu.vcores": "1"
            }
        }],
        "monitoringConfiguration": {
          "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/emr-serverless-hive/logs"
           }
        }
    }'
```

4. Note the job run ID returned in the output. Replace *job-run-id* with this ID in the following steps.

Step 3: Review your job run's output

The job run should typically take 3-5 minutes to complete.

Spark

You can check for the state of your Spark job with the following command.

```
aws emr-serverless get-job-run \
    --application-id application-id \
    --job-run-id job-run-id
```

With your log destination set to `s3://amzn-s3-demo-bucket/emr-serverless-spark/logs`, you can find the logs for this specific job run under `s3://amzn-s3-demo-bucket/emr-serverless-spark/logs/applications/application-id/jobs/job-run-id`.

For Spark applications, EMR Serverless pushes event logs every 30 seconds to the `sparklogs` folder in your S3 log destination. When your job completes, Spark runtime logs for the driver and executors upload to folders named appropriately by the worker type, such as `driver` or `executor`. The output of the PySpark job uploads to `s3://amzn-s3-demo-bucket/output/`.

Hive

You can check for the state of your Hive job with the following command.

```
aws emr-serverless get-job-run \
    --application-id application-id \
    --job-run-id job-run-id
```

With your log destination set to `s3://amzn-s3-demo-bucket/emr-serverless-hive/logs`, you can find the logs for this specific job run under `s3://amzn-s3-demo-bucket/emr-serverless-hive/logs/applications/application-id/jobs/job-run-id`.

For Hive applications, EMR Serverless continuously uploads the Hive driver to the `HIVE_DRIVER` folder, and Tez tasks logs to the `TEZ_TASK` folder, of your S3 log destination. After the job run reaches the `SUCCEEDED` state, the output of your Hive query becomes available in

the Amazon S3 location that you specified in the `monitoringConfiguration` field of `configurationOverrides`.

Step 4: Clean up

When you're done working with this tutorial, consider deleting the resources that you created. We recommend that you release resources that you don't intend to use again.

Delete your application

To delete an application, use the following command.

```
aws emr-serverless delete-application \
    --application-id application-id
```

Delete your S3 log bucket

To delete your S3 logging and output bucket, use the following command. Replace *amzn-s3-demo-bucket* with the actual name of the S3 bucket created in Prepare storage for EMR Serverless..

```
aws s3 rm s3://amzn-s3-demo-bucket --recursive
aws s3api delete-bucket --bucket amzn-s3-demo-bucket
```

Delete your job runtime role

To delete the runtime role, detach the policy from the role. You can then delete both the role and the policy.

```
aws iam detach-role-policy \
    --role-name EMRServerlessS3RuntimeRole \
    --policy-arn policy-arn
```

To delete the role, use the following command.

```
aws iam delete-role \
    --role-name EMRServerlessS3RuntimeRole
```

To delete the policy that was attached to the role, use the following command.

```
aws iam delete-policy \
    --policy-arn policy-arn
```

For more examples of running Spark and Hive jobs, see Using Spark configurations when you run EMR Serverless jobs and Using Hive configurations when you run EMR Serverless jobs.

Interact with and configure an EMR Serverless application

This section covers how you can interact with your Amazon EMR Serverless application with the AWS CLI. It also describes configuring an application, performing customizations, and defaults for Spark and Hive engines.

Topics

Application states

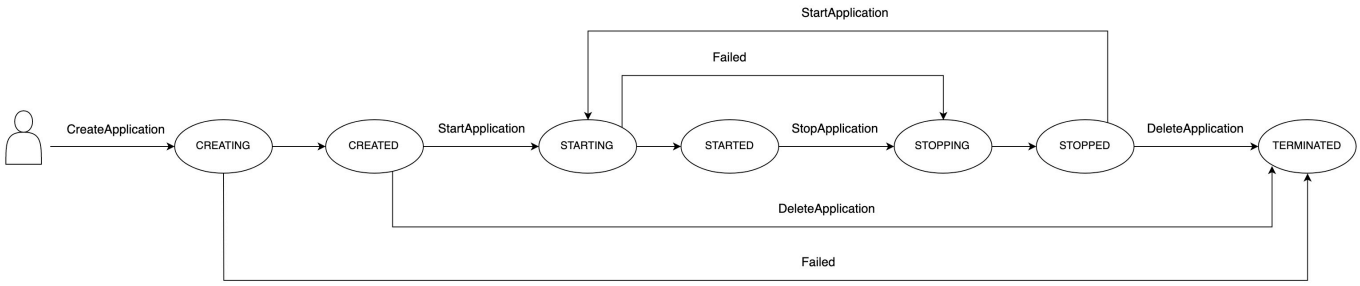
When you create an application with EMR Serverless, the application run enters the `CREATING` state. It then passes through the following states until it succeeds (exits with code `0`) or fails (exits with a non-zero code).

Applications can have the following states:

State	Description
Creating	The application is being prepared and isn't ready to use yet.
Created	The application has been created but hasn't provisioned capacity yet. You can modify the application to change its initial capacity configuration.

State	Description
Starting	The application is starting and is provisioning capacity.
Started	The application is ready to accept new jobs. The application only accepts jobs when it's in this state.
Stopping	All jobs have completed and the application is releasing its capacity.
Stopped	The application is stopped and no resources are running on the application. You can modify the application to change its initial capacity configuration.
Terminated	The application has been terminated and doesn't appear on your application list.

The following diagram shows the trajectory of EMR Serverless application states.



Creating an EMR Serverless application from the EMR Studio console

From the EMR Studio console, you can create, view, and manage EMR Serverless applications. To navigate to the EMR Studio console, follow the instructions in Getting started from the console.

Create an application

With the **Create application** page, you can create an EMR Serverless application by following these steps.

1. In the **Name** field, enter the name you want to call your application.
2. In the **Type** field, choose Spark or Hive as the type of the application.
3. In the **Release version** field, choose the EMR release number.
4. In the **Architecture** options, choose the instruction set architecture to use. For more information, see Amazon EMR Serverless architecture options.

   - **arm64** — 64-bit ARM architecture; to use Graviton processors
   - **x86_64** — 64-bit x86 architecture; to use x86-based processors
5. There are two application setup options for the remaining fields: default settings and custom settings. These fields are optional.

   Default settings — Default settings allow you to create an application quickly with pre-initialized capacity. This includes one driver and one executor for Spark, and one driver and one Tez Task for Hive. The default settings don't enable network connectivity to your VPCs. The application is configured to stop if idle for 15 minutes, and auto-starts on job submission.

   Custom settings — Custom settings allow you to modify the following properties.

   - **Pre-initialized capacity** — The number of drivers and executors or Hive Tez Task workers, and the size of each worker.
   - **Application limits** — The maximum capacity of an application.
   - **Application behavior** — The application's auto-start and auto-stop behavior.
   - **Network connections** — Network connectivity to VPC resources.
   - **Tags** — Custom tags that you can assign to the application.

   For more information about pre-initialized capacity, application limits, and application behavior, see Configuring an application when working with EMR Serverless. For more information about network connectivity, see Configuring VPC access for EMR Serverless applications to connect to data.
6. To create the application, choose **Create application** .

List applications from the EMR Studio console

You can view all existing EMR Serverless applications from the **List applications** page. You can choose an application's name to navigate to the **Details** page for that application.

Manage applications from the EMR Studio console

You can perform the following actions on an application from either the **List applications** page or from a specific application's **Details** page.

Start application

Choose this option to manually start an application.

Stop application

Choose this option to manually stop an application. An application should have no running jobs in order to be stopped. To learn more about application state transitions, see Application states.

Configure application

Edit the optional settings for an application from the **Configure application** page. You can change most application settings. For example, you can change the release label for an application to upgrade it to a different version of Amazon EMR, or you can switch the architecture from x86_64 to arm64. The other optional settings are the same as those that are in the **Custom settings** section on the **Create application** page. For more information about the application settings, see Create an application.

Delete application

Choose this option to manually delete an application. You must stop an application in order to delete it. To learn more about application state transitions, see Application states.

Interacting with your EMR Serverless application on the AWS CLI

From the AWS CLI, you can create, describe, and delete individual applications. You can also list all of your applications so that you can view them at a glance. This section describes how to perform these actions. For more application operations, such starting, stopping, and updating your application, see the EMR Serverless API Reference. For examples of how to use the EMR Serverless

API using the AWS SDK for Java, see Java examples in our GitHub repository. For examples of how to use the EMR Serverless API using the AWS SDK for Python (Boto), see Python examples in our GitHub repository.

To create an application, use `create-application`. You must specify `SPARK` or `HIVE` as the application `type`. This command returns the application's ARN, name, and ID.

```
aws emr-serverless create-application \
--name my-application-name \
--type 'application-type' \
--release-label release-version
```

To describe an application, use `get-application` and provide its `application-id`. This command returns the state and capacity-related configurations for your application.

```
aws emr-serverless get-application \
--application-id application-id
```

To list all of your applications, call `list-applications`. This command returns the same properties as `get-application` but includes all of your applications.

```
aws emr-serverless list-applications
```

To delete your application, call `delete-application` and supply your `application-id`.

```
aws emr-serverless delete-application \
--application-id application-id
```

Configuring an application when working with EMR Serverless

With EMR Serverless, you can configure the applications that you use. For example, you can set the maximum capacity that an application can scale up to, configure pre-initialized capacity to keep driver and workers ready to respond, and specify a common set of runtime and monitoring configurations at the application level. The following pages describe how to configure applications when you use EMR Serverless.

Topics

Understanding application behavior in EMR Serverless

This section describes job submission behavior, capacity configuration for scaling, and worker configuration settings for EMR Serverless.

Default application behavior

Auto-start — An application by default is configured to auto-start on job submission. You can turn this feature off.

Auto-stop — An application by default is configured to auto-stop when idle for 15 minutes. When an application changes to the STOPPED state, it releases any configured pre-initialized capacity. You can modify the amount of idle time before an application auto-stops, or you can turn this feature off.

Maximum capacity

You can configure the maximum capacity that an application can scale up to. You can specify your maximum capacity in terms of CPU, memory (GB), and disk (GB).

> **Note**
>
> We recommend configuring your maximum capacity to be proportional to your supported worker sizes by multiplying the number of workers by their sizes. For example, if you want to limit your application to 50 workers with 2 vCPUs, 16 GB for memory, and 20 GB for disk, set your maximum capacity to 100 vCPUs, 800 GB for memory, and 1000 GB for disk.

Supported worker configurations

The following table shows supported worker configurations and sizes that you can specify for EMR Serverless. You can configure different sizes for drivers and executors based on the need of your workload.

CPU	Memory	Default ephemeral storage
1 vCPU	Minimum 2 GB, maximum 8 GB, in 1 GB increments	20 GB - 200 GB
2 vCPU	Minimum 4 GB, maximum 16 GB, in 1 GB increments	20 GB - 200 GB
4 vCPU	Minimum 8 GB, maximum 30 GB, in 1 GB increments	20 GB - 200 GB
8 vCPU	Minimum 16 GB, maximum 60 GB, in 4 GB increments	20 GB - 200 GB
16 vCPU	Minimum 32 GB, maximum 120 GB, in 8 GB increments	20 GB - 200 GB

CPU — Each worker can have 1, 2, 4, 8, or 16 vCPUs.

Memory — Each worker has memory, specified in GB, within the limits listed in the earlier table. Spark jobs have a memory overhead, meaning that the memory they use is more than the specified container sizes. This overhead is specified with the properties `spark.driver.memory0verhead` and `spark.executor.memory0verhead`. The overhead has a default value of 10% of container memory, with a minimum of 384 MB. You should consider this overhead when you choose worker sizes.

For example, If you choose 4 vCPUs for your worker instance, and a pre-initialized storage capacity of 30 GB, then you should set a value of approximately 27 GB as executor memory for your Spark job. This maximizes the utilization of your pre-initialized capacity. Usable memory would be 27 GB, plus 10% of 27 GB (2.7 GB), for a total of 29.7 GB.

Disk — You can configure each worker with temporary storage disks with a minimum size of 20 GB and a maximum of 200 GB. You only pay for additional storage beyond 20 GB that you configure per worker.

Pre-initialized capacity for working with an application in EMR Serverless

EMR Serverless provides an optional feature that keeps driver and workers pre-initialized and ready to respond in seconds. This effectively creates a warm pool of workers for an application. This feature is called *pre-initialized capacity*. To configure this feature, you can set the `initialCapacity` parameter of an application to the number of workers you want to pre-initialize. With pre-initialized worker capacity, jobs start immediately. This is ideal when you want to implement iterative applications and time-sensitive jobs.

Pre-initialized capacity keeps a warm pool of workers ready for jobs and sessions to startup in seconds. You will be paying for provisioned pre-initialized workers even when the application is idle, hence we recommend enabling it for use cases that benefit from the fast start-up time and sizing it for optimal utilization of resources. EMR Serverless applications automatically shut down when idle. We recommend keeping this feature on when using pre-initialized workers to avoid unexpected charges.

When you submit a job, if workers from `initialCapacity` are available, the job uses those resources to start its run. If those workers are already in use by other jobs, or if the job needs more resources than available from `initialCapacity`, then the application requests and gets additional workers, up to the maximum limits on resources set for the application. When a job finishes its run, it releases the workers that it used, and the number of resources available for the application returns to `initialCapacity`. An application maintains the `initialCapacity` of resources even after jobs finish their runs. The application releases excess resources beyond `initialCapacity` when the jobs no longer need them to run.

Pre-initialized capacity is available and ready to use when the application has started. The pre-initialized capacity becomes inactive when the application is stopped. An application moves to the `STARTED` state only if the requested pre-initialized capacity has been created and is ready to use. The whole time that the application is in the `STARTED` state, EMR Serverless keeps the pre-initialized capacity available for use or in use by jobs or interactive workloads. The feature restores capacity for released or failed containers. This maintains the number of workers that the `InitialCapacity` parameter specifies. The state of an application with no pre-initialized capacity can immediately change from `CREATED` to `STARTED`.

You can configure the application to release pre-initialized capacity if it isn't used for a certain period of time, with a default of 15 minutes. A stopped application starts automatically when you

submit a new job. You can set these automatic start and stop configurations when you create the application, or you can change them when the application is in a CREATED or STOPPED state.

You can change the InitialCapacity counts, and specify compute configurations such as CPU, memory, and disk, for each worker. Because you can't make partial modifications, you should specify all compute configurations when you change values. You can only change configurations when the application is in the CREATED or STOPPED state.

> **Note**
>
> To optimize your application's use of resources, we recommend aligning your container sizes with your pre-initialized capacity worker sizes. For example, if you configure your Spark executor size to 2 CPUs and your memory to 8 GB, but your pre-initialized capacity worker size is 4 CPUs with 16 GB of memory, then the Spark executors only use half of the workers' resources when they are assigned to this job.

Customizing pre-initialized capacity for Spark and Hive

You can further customize pre-initialized capacity for workloads that run on specific big data frameworks. For example, when a workload runs on Apache Spark, you can specify how many workers start as drivers and how many start as executors. Similarly, when you use Apache Hive, you can specify how many workers start as Hive drivers, and how many should run Tez tasks.

Configuring an application running Apache Hive with pre-initialized capacity

The following API request creates an application running Apache Hive based on Amazon EMR release emr-6.6.0. The application starts with 5 pre-initialized Hive drivers, each with 2 vCPU and 4 GB of memory, and 50 pre-initialized Tez task workers, each with 4 vCPU and 8 GB of memory. When Hive queries run on this application, they first use the pre-initialized workers and start executing immediately. If all of the pre-initialized workers are busy and more Hive jobs are submitted, the application can scale to a total of 400 vCPU and 1024 GB of memory. You can optionally omit capacity for either the DRIVER or the TEZ_TASK worker.

```
aws emr-serverless create-application \
  --type "HIVE" \
  --name my-application-name \
  --release-label emr-6.6.0 \
  --initial-capacity '{
```

```
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "TEZ_TASK": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
  }'
```

Configuring an application running Apache Spark with pre-initialized capacity

The following API request creates an application that runs Apache Spark 3.2.0 based on Amazon EMR release 6.6.0. The application starts with 5 pre-initialized Spark drivers, each with 2 vCPU and 4 GB of memory, and 50 pre-initialized executors, each with 4 vCPU and 8 GB of memory. When Spark jobs run on this application, they first use the pre-initialized workers and start to execute immediately. If all of the pre-initialized workers are busy and more Spark jobs are submitted, the application can scale to a total of 400 vCPU and 1024 GB of memory. You can optionally omit capacity for either the `DRIVER` or the `EXECUTOR`.

> **Note**
>
> Spark adds a configurable memory overhead, with a 10% default value, to the memory requested for driver and executors. For jobs to use pre-initialized workers, the initial capacity memory configuration should be greater than the memory that the job and the overhead request.

```
aws emr-serverless create-application \
  --type "SPARK" \
  --name my-application-name \
```

```
  --release-label emr-6.6.0 \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "EXECUTOR": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
  }'
```

Default application configuration for EMR Serverless

You can specify a common set of runtime and monitoring configurations at the application level for all the jobs that you submit under the same application. This reduces the additional overhead that is associated with the need to submit the same configurations for each job.

You can modify the configurations at the following points in time:

- Declare application-level configurations at job submission.
- Override default configurations during job run.

The following sections provide more details and an example for further context.

Declaring configurations at the application level

You can specify application-level logging and runtime configuration properties for the jobs that you submit under the application.

`monitoringConfiguration`

To specify the log configurations for jobs that you submit with the application, use the `monitoringConfiguration` field. For more information on logging for EMR Serverless, see Storing logs.

`runtimeConfiguration`

To specify runtime configuration properties such as `spark-defaults`, provide a configuration object in the `runtimeConfiguration` field. This affects the default configurations for all the jobs that you submit with the application. For more information, see Hive configuration override parameter and Spark configuration override parameter.

Available configuration classifications vary by specific EMR Serverless release. For example, classifications for custom Log4j `spark-driver-log4j2` and `spark-executor-log4j2` are only available with releases 6.8.0 and higher. For a list of application-specific properties, see Spark job properties and Hive job properties.

You can also configure Apache Log4j2 properties, AWS Secrets Manager for data protection, and Java 17 runtime at the application level.

To pass Secrets Manager secrets at the application level, attach the following policy to users and roles that need to create or update EMR Serverless applications with secrets.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SecretsManagerPolicy",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:DescribeSecret",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:secretsmanager:your-secret-arn"
        }
    ]
}
```

For more information on creating custom policies for secrets, see Permissions policy examples for AWS Secrets Manager in the *AWS Secrets Manager User Guide*.

> **Note**
>
> The `runtimeConfiguration` that you specify at application level maps to `applicationConfiguration` in the `StartJobRun` API.

Example declaration

The following example shows how to declare default configurations with `create-application`.

```
aws emr-serverless create-application \
    --release-label release-version  \
    --type SPARK \
    --name my-application-name \
    --runtime-configuration '[
        {
            "classification": "spark-defaults",
            "properties": {
                "spark.driver.cores": "4",
                "spark.executor.cores": "2",
                "spark.driver.memory": "8G",
                "spark.executor.memory": "8G",
                "spark.executor.instances": "2",

 "spark.hadoop.javax.jdo.option.ConnectionDriverName":"org.mariadb.jdbc.Driver",
                "spark.hadoop.javax.jdo.option.ConnectionURL":"jdbc:mysql://db-host:db-
port/db-name",
                "spark.hadoop.javax.jdo.option.ConnectionUserName":"connection-user-
name",
                "spark.hadoop.javax.jdo.option.ConnectionPassword":
 "EMR.secret@SecretID"
            }
        },
        {
            "classification": "spark-driver-log4j2",
            "properties": {
                "rootLogger.level":"error",
                "logger.IdentifierForClass.name": "classpathForSettingLogger",
                "logger.IdentifierForClass.level": "info"
            }
        }
    ]' \
    --monitoring-configuration '{
```

```
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-logging-bucket/logs/app-level"
        },
        "managedPersistenceMonitoringConfiguration": {
            "enabled": false
        }
    }'
```

Overriding configurations during a job run

You can specify configuration overrides for the application configuration and monitoring configuration with the `StartJobRun` API. EMR Serverless then merges the configurations that you specify at the application level and the job level to determine the configurations for the job execution.

The granularity level when the merge occurs is as follows:

- **`ApplicationConfiguration`** - Classification type, for example `spark-defaults`.
- **`MonitoringConfiguration`** - Configuration type, for example `s3MonitoringConfiguration`.

> **Note**
>
> The priority of configurations that you provide at `StartJobRun` supersede the configurations that you provide at the application level.

For more information priority rankings, see Hive configuration override parameter and Spark configuration override parameter.

When you start a job, if you don't specify a particular configuration, it will be inherited from the application. If you declare the configurations at job level, you can perform the following operations:

- **Override an existing configuration** - Provide the same configuration parameter in the `StartJobRun` request with your override values.
- **Add an additional configuration** - Add the new configuration parameter in the `StartJobRun` request with the values that you want to specify.

- **Remove an existing configuration** - To remove an application *runtime configuration*, provide the key for the configuration that you want to remove, and pass an empty declaration `{}` for the configuration. We don't recommend removing any classifications that contain parameters that are required for a job run. For example, if you try to remove the required properties for a Hive job, the job will fail.

  To remove an application *monitoring configuration*, use the appropriate method for the relevant configuration type:

  - **`cloudWatchLoggingConfiguration`** - To remove `cloudWatchLogging`, pass the enabled flag as `false`.
  - **`managedPersistenceMonitoringConfiguration`** - To remove managed persistence settings and fall back to the default enabled state, pass an empty declaration `{}` for the configuration.
  - **`s3MonitoringConfiguration`** - To remove `s3MonitoringConfiguration`, pass an empty declaration `{}` for the configuration.

Example override

The following example shows different operations you can perform during job submission at `start-job-run`.

```
aws emr-serverless start-job-run \
    --application-id your-application-id \
    --execution-role-arn your-job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
            "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket1/
wordcount_output"]
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [
            {
                // Override existing configuration for spark-defaults in the
 application
                "classification": "spark-defaults",
                "properties": {
                    "spark.driver.cores": "2",
```

```
                    "spark.executor.cores": "1",
                    "spark.driver.memory": "4G",
                    "spark.executor.memory": "4G"
                }
            },
            {
                // Add configuration for spark-executor-log4j2
                "classification": "spark-executor-log4j2",
                "properties": {
                    "rootLogger.level": "error",
                    "logger.IdentifierForClass.name": "classpathForSettingLogger",
                    "logger.IdentifierForClass.level": "info"
                }
            },
            {
                // Remove existing configuration for spark-driver-log4j2 from the
application
                "classification": "spark-driver-log4j2",
                "properties": {}
            }
        ],
        "monitoringConfiguration": {
            "managedPersistenceMonitoringConfiguration": {
                // Override existing configuration for managed persistence
                "enabled": true
            },
            "s3MonitoringConfiguration": {
                // Remove configuration of S3 monitoring
            },
            "cloudWatchLoggingConfiguration": {
                // Add configuration for CloudWatch logging
                "enabled": true
            }
        }
    }'
```

At the time of job execution, the following classifications and configurations will apply based on the priority override ranking described in Hive configuration override parameter and Spark configuration override parameter.

- The classification `spark-defaults` will be updated with the properties specified at the job level. Only the properties included in `StartJobRun` would be considered for this classification.

- The classification `spark-executor-log4j2` will be added in the existing list of classifications.
- The classification `spark-driver-log4j2` will be removed.
- The configurations for `managedPersistenceMonitoringConfiguration` will be updated with configurations at job level.
- The configurations for `s3MonitoringConfiguration` will be removed.
- The configurations for `cloudWatchLoggingConfiguration` will be added to existing monitoring configurations.

Customizing an EMR Serverless image

Starting with Amazon EMR 6.9.0, you can use custom images to package application dependencies and runtime environments into a single container with Amazon EMR Serverless. This simplifies how you manage workload dependencies and makes your packages more portable. When you customize your EMR Serverless image, it provides the following benefits:

- Installs and configures packages that are optimized to your workloads. These packages might not be widely available in the public distribution of Amazon EMR runtime environments.
- Integrates EMR Serverless with current established build, test, and deployment processes within your organization, including local development and testing.
- Applies established security processes, such as image scanning, that meet compliance and governance requirements within your organization.
- Lets you use your own versions of JDK and Python for your applications.

EMR Serverless provides images that you can use as your base when you create your own images. The base image provides the essential jars, configuration, and libraries for the image to interact with EMR Serverless. You can find the base image in the Amazon ECR Public Gallery. Use the image that matches your application type (Spark or Hive) and release version. For example, if you create an application on Amazon EMR release 6.9.0, use the following images.

Type	Image
Spark	`public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest`

Type	Image
Hive	`public.ecr.aws/emr-serverless/hive/emr-6.9.0:latest`

Prerequisites

Before you create an EMR Serverless custom image, complete these prerequisites.

1. Create an Amazon ECR repository in the same AWS Region that you use to launch EMR Serverless applications. To create an Amazon ECR private repository, see Creating a private repository.
2. To grant users access to your Amazon ECR repository, add the following policies to users and roles that create or update EMR Serverless applications with images from this repository.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ECRRepositoryListGetPolicy",
            "Effect": "Allow",
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "ecr:DescribeImages"
            ],
            "Resource": "ecr-repository-arn"
        }
    ]
}
```

For more examples of Amazon ECR identity-based policies, see Amazon Elastic Container Registry identity-based policy examples.

Step 1: Create a custom image from EMR Serverless base images

First, create a Dockerfile that begins with a `FROM` instruction that uses your preferred base image. After the `FROM` instruction, you can include any modification that you want to make to the image.

The base image automatically sets the USER to hadoop. This setting might not have permissions for all the modifications you include. As a workaround, set the USER to `root`, modify your image, and then set the USER back to `hadoop:hadoop`. To see samples for common use cases, see Using custom images with EMR Serverless.

```
# Dockerfile
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root
# MODIFICATIONS GO HERE

# EMRS will run the image as hadoop
USER hadoop:hadoop
```

After you have the Dockerfile, build the image with the following command.

```
# build the docker image
docker build . -t aws-account-id.dkr.ecr.region.amazonaws.com/my-
repository[:tag]or[@digest]
```

Step 2: Validate image locally

EMR Serverless provides an offline tool that can statically check your custom image to validate basic files, environment variables, and correct image configurations. For information on how to install and run the tool, see the Amazon EMR Serverless Image CLI GitHub.

After you install the tool, run the following command to validate an image:

```
amazon-emr-serverless-image \
validate-image -r emr-6.9.0 -t spark \
-i aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/@digest
```

You should see an output similar to the following.

```
Amazon EMR Serverless - Image CLI
Version: 0.0.1
... Checking if docker cli is installed
... Checking Image Manifest
[INFO] Image ID: 9e2f4359cf5beb466a8a2ed047ab61c9d37786c555655fc122272758f761b41a
```

```
[INFO] Created On: 2022-12-02T07:46:42.586249984Z
[INFO] Default User Set to hadoop:hadoop : PASS
[INFO] Working Directory Set to  : PASS
[INFO] Entrypoint Set to /usr/bin/entrypoint.sh : PASS
[INFO] HADOOP_HOME is set with value: /usr/lib/hadoop : PASS
[INFO] HADOOP_LIBEXEC_DIR is set with value: /usr/lib/hadoop/libexec : PASS
[INFO] HADOOP_USER_HOME is set with value: /home/hadoop : PASS
[INFO] HADOOP_YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] HIVE_HOME is set with value: /usr/lib/hive : PASS
[INFO] JAVA_HOME is set with value: /etc/alternatives/jre : PASS
[INFO] TEZ_HOME is set with value: /usr/lib/tez : PASS
[INFO] YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] File Structure Test for hadoop-files in /usr/lib/hadoop: PASS
[INFO] File Structure Test for hadoop-jars in /usr/lib/hadoop/lib: PASS
[INFO] File Structure Test for hadoop-yarn-jars in /usr/lib/hadoop-yarn: PASS
[INFO] File Structure Test for hive-bin-files in /usr/bin: PASS
[INFO] File Structure Test for hive-jars in /usr/lib/hive/lib: PASS
[INFO] File Structure Test for java-bin in /etc/alternatives/jre/bin: PASS
[INFO] File Structure Test for tez-jars in /usr/lib/tez: PASS
----------------------------------------------------------------
Overall Custom Image Validation Succeeded.
----------------------------------------------------------------
```

Step 3: Upload the image to your Amazon ECR repository

Push your Amazon ECR image to your Amazon ECR repository with the following commands. Ensure you have the correct IAM permissions to push the image to your repository. For more information, see Pushing an image in the *Amazon ECR User Guide*.

```
# login to ECR repo
aws ecr get-login-password --region region | docker login --username AWS --password-
stdin aws-account-id.dkr.ecr.region.amazonaws.com

# push the docker image
docker push aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/@digest
```

Step 4: Create or update an application with custom images

Choose the AWS Management Console tab or AWS CLI tab according to how you want to launch your application, then complete the following steps.

Console

1. Sign in to the EMR Studio console at https://console.aws.amazon.com/emr. Navigate to your application, or create a new application with the instructions in Create an application.
2. To specify custom images when you create or update an EMR Serverless application, select **Custom settings** in the application setup options.
3. In the **Custom image settings** section, select the **Use the custom image with this application** check box.
4. Paste the Amazon ECR image URI into the **Image URI** field. EMR Serverless uses this image for all worker types for the application. Alternatively, you can choose **Different custom images** and paste different Amazon ECR image URIs for each worker type.

CLI

- Create an application with the `image-configuration` parameter. EMR Serverless applies this setting to all worker types.

```
aws emr-serverless create-application \
--release-label emr-6.9.0 \
--type SPARK \
--image-configuration '{
    "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/
@digest"
}'
```

To create an application with different image settings for each worker type, use the `worker-type-specifications` parameter.

```
aws emr-serverless create-application \
--release-label emr-6.9.0 \
--type SPARK \
--worker-type-specifications '{
    "Driver": {
        "imageConfiguration": {
            "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-
repository:tag/@digest"
        }
    },
    "Executor" : {
```

```
        "imageConfiguration": {
            "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-
repository:tag/@digest"
        }
    }
}'
```

To update an application, use the `image-configuration` parameter. EMR Serverless applies this setting to all worker types.

```
aws emr-serverless update-application \
--application-id application-id \
--image-configuration '{
    "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/
@digest"
}'
```

Step 5: Allow EMR Serverless to access the custom image repository

Add the following resource policy to the Amazon ECR repository to allow the EMR Serverless service principal to use the `get`, `describe`, and `download` requests from this repository.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Emr Serverless Custom Image Support",
      "Effect": "Allow",
      "Principal": {
        "Service": "emr-serverless.amazonaws.com"
      },
      "Action": [
        "ecr:BatchGetImage",
        "ecr:DescribeImages",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Condition":{
        "StringEquals":{
          "aws:SourceArn": "arn:aws:emr-serverless:region:aws-account-id:/
applications/application-id"
        }
```

```
      }
    }
  ]
}
```

As a security best practice, add an `aws:SourceArn` condition key to the repository policy. The IAM global condition key `aws:SourceArn` ensures that EMR Serverless uses the repository only for an application ARN. For more information on Amazon ECR repository policies, see Creating a private repository.

Considerations and limitations

When you work with custom images, consider the following:

- Use the correct base image that matches the type (Spark or Hive) and release label (for example, `emr-6.9.0`) for your application.
- EMR Serverless ignores `[CMD]` or `[ENTRYPOINT]` instructions in the Docker file. Use common instructions in the Docker file, such as `[COPY]`, `[RUN]`, and `[WORKDIR]`.
- You shouldn't modify environment variables `JAVA_HOME`, `SPARK_HOME`, `HIVE_HOME`, `TEZ_HOME` when you create a custom image.
- Custom images can't exceed 10 GB in size.
- If you modify binaries or jars in the Amazon EMR base images, it might cause application or job launch failures.
- The Amazon ECR repository should be in the same AWS Region that you use to launch EMR Serverless applications.

Configuring VPC access for EMR Serverless applications to connect to data

You can configure EMR Serverless applications to connect to your data stores within your VPC, such as Amazon Redshift clusters, Amazon RDS databases or Amazon S3 buckets with VPC endpoints. Your EMR Serverless application has outbound connectivity to the data stores within your VPC. By default, EMR Serverless blocks inbound access to your applications to improve security.

> **Note**
>
> You must configure VPC access if you want to use an external Hive metastore database for your application. For information about how to configure an external Hive metastore, see Metastore configuration.

Create application

On the **Create application** page, you can choose custom settings and specify the VPC, subnets and security groups that EMR Serverless applications can use.

VPCs

Choose the name of the virtual private cloud (VPC) that contains your data stores. The **Create application** page lists all VPCs for your chosen AWS Region.

Subnets

Choose the subnets within the VPC that contains your data store. The **Create application** page lists all subnets for the data stores in your VPC. Both public and private subnets are supported. You can pass either private or public subnets to your applications. The choice of whether to have a public or private subnet has a few associated considerations to be aware of.

For private subnets:

- The associated route tables must not have internet gateways.
- For outbound connectivity to the internet, if needed, configure outbound routes using a NAT Gateway. To configure a NAT Gateway, see NAT gateways.
- For Amazon S3 connectivity, configure either a NAT Gateway or a VPC endpoint. To configure an S3 VPC endpoint, see Create a gateway endpoint.
- For connectivity to other AWS services outside the VPC, such as to Amazon DynamoDB, configure either VPC endpoints or a NAT gateway. To configure VPC endpoints for AWS services, see Work with VPC endpoints.

Note

When you set up an Amazon EMR Serverless application in a private subnet, we recommend that you also set up VPC endpoints for Amazon S3. If your EMR Serverless application is in a private subnet without VPC endpoints for Amazon S3, you could incur additional NAT gateway charges that are associated with S3 traffic. This is because the traffic between your EMR application and Amazon S3 will not stay within your VPC when VPC endpoints aren't configured.

For public subnets:

- These have a route to an Internet Gateway.
- You must ensure proper security group configurations to control outbound traffic.

Workers can connect to the data stores within your VPC through outbound traffic. By default, EMR Serverless blocks inbound access to workers. This is to improve security.

When you use AWS Config, EMR Serverless creates an elastic network interface item record for every worker. To avoid costs related to this resource, consider turning off `AWS::EC2::NetworkInterface` in AWS Config.

Note

We recommend that you select multiple subnets across multiple Availability Zones. This is because the subnets that you choose determine the Availability Zones available for an EMR Serverless application to launch. Each worker consumes an IP address on the subnet where it is launched. Please ensure that the specified subnets have sufficient IP addresses for the number of workers you plan to launch. For more information on subnet planning, see the section called “Best practices for subnet planning”.

Considerations and limitations for subnets

- EMR Serverless with public subnets does not support AWS Lake Formation.
- Inbound traffic isn't supported for public subnets.

Security groups

Choose one or more security groups that can communicate with your data stores. The **Create application** page lists all security groups in your VPC. EMR Serverless associates these security groups with elastic network interfaces that are attached to your VPC subnets.

> **Note**
>
> We recommend that you create a separate security group for EMR Serverless applications. EMR Serverless will not allow you to **Create/Update/Start application** if security groups have ports open to the public internet on **0.0.0.0/0** or the **::/0** range. This provides enhanced security, isolation, and makes managing network rules more efficient. For example, this blocks unexpected traffic to workers with public IP addresses. To communicate with Amazon Redshift clusters, for instance, you can define the traffic rules between Redshift and EMR Serverless security groups, as demonstrated in the example below.

Example Example — Communication with Amazon Redshift clusters

1. Add a rule for inbound traffic to the Amazon Redshift security group from one of the EMR Serverless security groups.

Type	Protocol	Port range	Source
All TCP	TCP	5439	`emr-serverless-security-group`

2. Add a rule for outbound traffic from one of the EMR Serverless security groups. You can do this in one of two ways. First, you can open outbound traffic to all ports.

Type	Protocol	Port range	Destination
All traffic	TCP	ALL	0.0.0.0/0

Alternatively, you can restrict outbound traffic to Amazon Redshift clusters. This is useful only when the application must communicate with Amazon Redshift clusters and nothing else.

Type	Protocol	Port range	Source
All TCP	TCP	5439	`redshift-security-group`

Configure application

You can change the network configuration for an existing EMR Serverless application from the **Configure application** page.

View job run details

On the **Job run detail** page, you can view the subnet used by your job for a specific run. Note that a job runs only in one subnet selected from the specified subnets.

Best practices for subnet planning

AWS resources are created in a subnet which is a subset of available IP addresses in an Amazon VPC. For example, a VPC with a /16 netmask has up to 65,536 available IP addresses which can be broken into multiple smaller networks using subnet masks. As an example, you can split this range into two subnets with each using /17 mask and 32,768 available IP addresses. A subnet resides within an Availability Zone and cannot span across zones.

The subnets should be designed keeping in mind your EMR Serverless application scaling limits. For example, if you have an application requesting 4 vCpu workers and can scale up to 4,000 vCpu, then your application will require at most 1,000 workers for a total of 1,000 network interfaces. We recommend that you create subnets across multiple Availability Zones. This allows EMR Serverless to retry your job or provision pre-initialized capacity in a different Availability Zone in an unlikely event when an Availability Zone fails. Therefore, each subnet in at least two Availability Zones should have more than 1,000 available IP addresses.

You need subnets with mask size lower than or equal to 22 to provision 1,000 network interfaces. Any mask greater than 22 will not meet the requirement. For example, a subnet mask of /23

provides 512 IP addresses, while a mask of /22 provides 1024 and a mask of /21 provides 2048 IP addresses. Below is an example of 4 subnets with /22 mask in a VPC of /16 netmask that can be allocated to different Availability Zones. There is a difference of five between available and usable IP addresses because first four IP addresses and last IP address in each subnet is reserved by AWS.

Subnet ID	Subnet Address	Subnet Mask	IP Address Range	Available IP Addresses	Usable IP Addresses
1	10.0.0.0	255.255.252.0/22	10.0.0.0 - 10.0.3.255	1,024	1,019
2	10.0.4.0	255.255.252.0/22	10.0.4.0 - 10.0.7.255	1,024	1,019
3	10.0.8.0	255.255.252.0/22	10.0.4.0 - 10.0.7.255	1,024	1,019
4	10.0.12.0	255.255.252.0/22	10.0.12.0 - 10.0.15.255	1,024	1,019

You should evaluate if your workload is best suited for larger worker sizes. Using larger worker sizes requires fewer network interfaces. For example, using 16vCpu workers with an application scaling limit of 4,000 vCpu will require at most 250 workers for a total of 250 available IP addresses to provision network interfaces. You need subnets in multiple Availability Zones with mask size lower than or equal to 24 to provision 250 network interfaces. Any mask size greater than 24 offers less than 250 IP addresses.

If you share subnets across multiple applications, each subnet should be designed keeping in mind collective scaling limits of all your applications. For example, if you have 3 applications requesting 4 vCpu workers and each can scale up to 4000 vCpu with 12,000 vCpu account-level service based quota, each subnet will require 3000 available IP addresses. If the VPC that you want to use doesn't have a sufficient number of IP addresses, try to increase the number of available IP addresses. You can do this by associating additional Classless Inter-Domain Routing (CIDR) blocks with your VPC. For more information, see Associate additional IPv4 CIDR blocks with your VPC in the *Amazon VPC User Guide*.

You can use one of the many tools available online to quickly generate subnet definitions and review their available range of IP addresses.

Amazon EMR Serverless architecture options

The instruction set architecture of your Amazon EMR Serverless application determines the type of processors that the application uses to run the job. Amazon EMR provides two architecture options for your application: **x86_64** and **arm64**. EMR Serverless automatically updates to the latest generation of instances as they become available, so your applications can use the newer instances without requiring additional effort from you.

Topics

Using x86_64 architecture

The **x86_64** architecture is also known as x86 64-bit or x64. **x86_64** is the default option for EMR Serverless applications. This architecture uses x86-based processors and is compatible with most third-party tools and libraries.

Most applications are compatible with the x86 hardware platform and can run successfully on the default **x86_64** architecture. However, if your application is compatible with 64-bit ARM, then you can switch to **arm64** to use Graviton processors for improved performance, compute power, and memory. It costs less to run instances on arm64 architecture than when you run instances of equal size on x86 architecture.

Using arm64 architecture (Graviton)

AWS Graviton processors are custom designed by AWS with 64-bit ARM Neoverse cores and leverage the arm64 architecture (also known as Arch64 or 64-bit ARM). The AWS Graviton line of processors available on EMR Serverless include Graviton3 and Graviton2 processors. These processors deliver superior price-performance for Spark and Hive workloads compared to equivalent workloads that run on the x86_64 architecture. EMR Serverless automatically uses the latest generation of processors when available without any effort from your side to upgrade to the latest generation of processors.

Launching new applications with Graviton support

Use one of the following methods to launch an application that uses the **arm64** architecture.

AWS CLI

To launch an application using Graviton processors from AWS CLI, specify ARM64 as the `architecture` parameter in the `create-application` API. Provide the appropriate values for your application in the other parameters.

```
aws emr-serverless create-application \
 --name my-graviton-app \
 --release-label emr-6.8.0 \
 --type "SPARK" \
 --architecture "ARM64" \
 --region us-west-2
```

EMR Studio

To launch an application using Graviton processors from EMR Studio, choose **arm64** as the **Architecture** option when you create or update an application.

Configuring existing applications to use Graviton

You can configure your existing Amazon EMR Serverless applications to use the Graviton (arm64) architecture with the SDK, AWS CLI, or EMR Studio.

To convert an existing application from x86 to arm64

1. Confirm that you are using the latest major version of the AWS CLI/SDK that supports the `architecture` parameter.
2. Confirm that there are no jobs running and then stop the application.

```
aws emr-serverless stop-application \
 --application-id application-id \
 --region us-west-2
```

3. To update the application to use Graviton, specify ARM64 for the `architecture` parameter in the `update-application` API.

```
aws emr-serverless update-application \
 --application-id application-id \
 --architecture 'ARM64' \
 --region us-west-2
```

4. To verify that the CPU architecture of the application is now ARM64, use the `get-application` API.

```
aws emr-serverless get-application \
 --application-id application-id \
 --region us-west-2
```

5. When you're ready, restart the application.

```
aws emr-serverless start-application \
 --application-id application-id \
 --region us-west-2
```

Considerations when using Graviton

Before you launch an EMR Serverless application using arm64 for Graviton support, confirm the following.

Library compatibility

When you select Graviton (arm64) as an architecture option, ensure that third-party packages and libraries are compatible with the 64-bit ARM architecture. For information on how to package Python libraries into a Python virtual environment that is compatible with your selected architecture, see Using Python libraries with EMR Serverless.

To learn more about how to configure a Spark or Hive workload to use 64-bit ARM, see the AWS Graviton Getting Started repository on GitHub. This repository contains essential resources that can help you get started with the ARM-based Graviton.

Job concurrency and queuing for an EMR Serverless application

Starting with Amazon EMR version 7.0.0 and later, you can specify job run queue timeout and concurrency configuration for your application. When you specify this configuration, Amazon EMR

Serverless starts by queuing your job and begins execution based on concurrency utilization on your application. For example, if your job run concurrency is 10, only ten jobs are run at a time on your application. Remaining jobs are queued until one of the running jobs terminates. If queue timeout is reached earlier, your job times out. For more information, see Job run states.

Key benefits of concurrency and queuing

Job concurrency and queuing provides the following benefits when many job submissions are required:

- It helps control concurrent executing jobs to efficiently use your application level capacity limits.
- The queue can contain a sudden burst of job submissions, with a configurable timeout setting.

Getting started with concurrency and queuing

The following procedures show a couple different ways to implement concurrency and queuing.

Using the AWS CLI

1. Create an Amazon EMR Serverless application with queue timeout and concurrent job runs:

```
aws emr-serverless create-application \
--release-label emr-7.0.0 \
--type SPARK \
--scheduler-configuration '{"maxConcurrentRuns": 1, "queueTimeoutMinutes": 30}'
```

2. Update an application to change the job queue timeout and concurrency:

```
aws emr-serverless update-application \
--application-id application-id \
--scheduler-configuration '{"maxConcurrentRuns": 5, "queueTimeoutMinutes": 30}'
```

> **Note**
>
> You can update your existing application to enable job concurrency and queuing. To do this, the application must have a release label *emr-7.0.0* or later.

Using the AWS Management Console

The following steps show you how to get started with job concurrency and queuing, using the AWS Management Console:

1. Go to EMR Studio and choose to create an application with release label EMR-7.0.0 or higher.
2. Under **Application setup options**, select the option **Use custom settings**.
3. Under **Additional configurations** there is a section for **Job Run Settings**. Select the option **Enable job concurrency** to enable the feature.
4. Once selected, you can select both **Concurrent job runs** and **Queue timeout** to configure the number of concurrent job runs and queue timeout, respectively. If you do not enter values for these settings, the default values are used.
5. Choose **Create Application** and the application will be created with this feature enabled. To verify, go to the dashboard, select your application and check under properties tab to determine if the feature is enabled.

Following configuration, you can submit jobs with this feature enabled.

Considerations for concurrency and queuing

Take the following into consideration when you implement concurrency and queuing:

- Job concurrency and queuing is supported on Amazon EMR release 7.0.0 and higher.
- Job concurrency and queuing is enabled by default on Amazon EMR release 7.3.0 and higher.
- You can update concurrency for an application in the **STARTED** state.
- The valid range for `maxConcurrentRuns` is 1 to 1000, and for `queueTimeoutMinutes` it is 15 to 720.
- A maximum of 2000 jobs can be in the **QUEUED** state for an account.
- Concurrency and queuing applies to batch and streaming jobs. It cannot be used for interactive jobs. For more information, see Run interactive workloads with EMR Serverless through EMR Studio.

Get data into S3 Express One Zone with EMR Serverless

With Amazon EMR releases 7.2.0 and higher, you can use EMR Serverless with the Amazon S3 Express One Zone storage class for improved performance when you run jobs and workloads. S3 Express One Zone is a a high-performance, single-zone Amazon S3 storage class that delivers consistent, single-digit millisecond data access for most latency-sensitive applications. At the time of its release, S3 Express One Zone delivers the lowest latency and highest performance cloud object storage in Amazon S3.

Prerequisites

- S3 Express One Zone permissions – When S3 Express One Zone initially performs an action like `GET`, `LIST`, or `PUT` on an S3 object, the storage class calls `CreateSession` on your behalf. Your IAM policy must allow the `s3express:CreateSession` permission so that the S3A connector can invoke the `CreateSession` API. For an example policy with this permission, see Getting started with S3 Express One Zone.
- S3A connector – To configure Spark to access data from an Amazon S3 bucket that uses the S3 Express One Zone storage class, you must use the Apache Hadoop connector S3A. To use the connector, ensure all S3 URIs use the `s3a` scheme. If they don't, you can change the filesystem implementation that you use for `s3` and `s3n` schemes.

To change the `s3` scheme, specify the following cluster configurations:

```
[
  {
    "Classification": "core-site",
    "Properties": {
      "fs.s3.impl": "org.apache.hadoop.fs.s3a.S3AFileSystem",
      "fs.AbstractFileSystem.s3.impl": "org.apache.hadoop.fs.s3a.S3A"
    }
  }
]
```

To change the `s3n` scheme, specify the following cluster configurations:

```
[
  {
```

```
    "Classification": "core-site",
    "Properties": {
      "fs.s3n.impl": "org.apache.hadoop.fs.s3a.S3AFileSystem",
      "fs.AbstractFileSystem.s3n.impl": "org.apache.hadoop.fs.s3a.S3A"
    }
  }
]
```

Getting started with S3 Express One Zone

Follow these steps to get started with S3 Express One Zone.

1. Create a VPC endpoint. Add the endpoint `com.amazonaws.us-west-2.s3express` to the VPC endpoint.
2. Follow Getting started with Amazon EMR Serverless to create an application with Amazon EMR release label 7.2.0 or higher.
3. Configure your application to use the newly created VPC endpoint, a private subnet group, and a security group.
4. Add the `CreateSession` permission to your job execution role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Resource": "*",
            "Action": [
                "s3express:CreateSession"
            ]
        }
    ]
}
```

5. Run your job. Note that you must use the S3A scheme to access S3 Express One Zone buckets.

```
aws emr-serverless start-job-run \
--application-id <application-id> \
--execution-role-arn <job-role-arn> \
--name <job-run-name> \
--job-driver '{
```

```
"sparkSubmit": {


"entryPoint": "s3a://<DOC-EXAMPLE-BUCKET>/scripts/wordcount.py",
"entryPointArguments":["s3a://<DOC-EXAMPLE-BUCKET>/emr-serverless-spark/output"],
"sparkSubmitParameters": "--conf spark.executor.cores=4
--conf spark.executor.memory=8g --conf spark.driver.cores=4
--conf spark.driver.memory=8g --conf spark.executor.instances=2
--conf spark.hadoop.fs.s3a.change.detection.mode=none
--conf spark.hadoop.fs.s3a.endpoint.region={<AWS_REGION>}
--conf spark.hadoop.fs.s3a.select.enabled=false
--conf spark.sql.sources.fastS3PartitionDiscovery.enabled=false
}'
```

Running jobs

After you provision your application, you can submit jobs to the application. This section covers how to use the AWS CLI to run these jobs. This section also identifies the default values for each type of application that is available on EMR Serverless.

Topics

Job run states

When you submit a job run to an Amazon EMR Serverless job queue, the job run enters the `SUBMITTED` state. A job state's passes from `SUBMITTED` through `RUNNING` until it reaches `FAILED`, `SUCCESS`, or `CANCELLING`.

Job runs can have the following states:

State	Description
Submitted	The initial job state when you submit a job run to EMR Serverless. The job waits to be scheduled for the application. EMR Serverless begins to prioritize and schedule the job run.

State	Description
Queued	The job run waits in this state when applicati on level job run concurrency is fully occupied. For more information about queuing and concurrency, see Job concurrency and queuing for an EMR Serverless application.
Pending	The scheduler is evaluating the job run to prioritize and schedule the run for the application.
Scheduled	EMR Serverless has scheduled the job run for the application, and is allocating resources to execute the job.
Running	EMR Serverless has allocated the resources that the job initially needs, and the job is running in the application. In Spark applicati ons, this means that the Spark driver process is in the `running` state.
Failed	EMR Serverless failed to submit the job run to the application, or it completed unsuccessfully. See `StateDetails` for additional informati on about this job failure.
Success	The job run has completed successfully.
Cancelling	The `CancelJobRun` API has requested job run cancellation, or the job run has timed out. EMR Serverless is trying to cancel the job in the application and release the resources.
Cancelled	The job run was cancelled successfully, and the resources that it used have been released.

Running jobs from the EMR Studio console

You can submit job runs to EMR Serverless applications and view the jobs from the EMR Studio console. To create or navigate to your EMR Serverless application on the EMR Studio console, follow the instructions in Getting started from the console.

Submit a job

On the **Submit job** page, you can submit a job to an EMR Serverless application as follows.

Spark

1. In the **Name** field, enter a name for your job run.
2. In the **Runtime role** field, enter the name of the IAM role that your EMR Serverless application can assume for the job run. To learn more about runtime roles, see Job runtime roles for Amazon EMR Serverless.
3. In the **Script location** field, enter the Amazon S3 location for the script or JAR that you want to run. For Spark jobs, the script can be a Python (`.py`) file or a JAR (`.jar`) file.
4. If your script location is a JAR file, enter the class name that is the entry point for the job in the **Main class** field.
5. (Optional) Enter values for the remaining fields.

   - **Script arguments** — Enter any arguments that you want to pass to your main JAR or Python script. Your code reads these parameters. Separate each argument in the array by a comma.
   - **Spark properties** — Expand the Spark properties section and enter any Spark configuration parameters in this field.

     > **Note**
     >
     > If you specify Spark driver and executor sizes, you must take memory overhead into account. Specify memory overhead values in the properties `spark.driver.memoryOverhead` and `spark.executor.memoryOverhead`. Memory overhead has a default value of 10% of container memory, with a minimum of 384 MB. The executor memory and the memory overhead together can't exceed the worker memory. For example, the maximum `spark.executor.memory` on a 30 GB worker must be 27 GB.

- **Job configuration** — Specify any job configuration in this field. You can use these job configurations to override the default configurations for applications.
- **Additional settings** — Active or deactivate the AWS Glue Data Catalog as a metastore and modify application log settings. To learn more about metastore configurations, see Metastore configuration for EMR Serverless. To learn more about application logging options, see Storing logs.
- **Tags** — Assign custom tags to the application.

6. Choose **Submit job**.

Hive

1. In the **Name** field, enter a name for your job run.
2. In the **Runtime role** field, enter the name of the IAM role that your EMR Serverless application can assume for the job run.
3. In the **Script location** field, enter the Amazon S3 location for the script or JAR that you want to run. For Hive jobs, the script must be a Hive (`.sql`) file.
4. (Optional) Enter values for the remaining fields.

   - **Initialization script location** – Enter the location of the script that initializes tables before the Hive script runs.
   - **Hive properties** – Expand the Hive properties section and enter any Hive configuration parameters in this field.
   - **Job configuration** – Specify any job configuration. You can use these job configurations to override the default configurations for applications. For Hive jobs, `hive.exec.scratchdir` and `hive.metastore.warehouse.dir` are required properties in the `hive-site` configuration.

```
{
    "applicationConfiguration": [
        {
            "classification": "hive-site",
            "configurations": [],
            "properties": {
                "hive.exec.scratchdir": "s3://DOC-EXAMPLE_BUCKET/hive/
scratch",
                "hive.metastore.warehouse.dir": "s3://DOC-EXAMPLE_BUCKET/hive/
warehouse"
```

```
            }
        }
    ],
    "monitoringConfiguration": {}
}
```

- **Additional settings** — Activate or deactivate the AWS Glue Data Catalog as a metastore and modify application log settings. To learn more about metastore configurations, see Metastore configuration for EMR Serverless. To learn more about application logging options, see Storing logs.
- **Tags** — Assign any custom tags to the application.

5. Choose **Submit job**.

View job runs

From the **Job runs** tab on an application's **Details** page, you can view job runs and perform the following actions for job runs.

Cancel job — To cancel a job run that is in the `RUNNING` state, choose this option. To learn more about job run transitions, see Job run states.

Clone job — To clone a previous job run and resubmit it, choose this option.

Running jobs from the AWS CLI

You can create, describe, and delete individual jobs on the AWS CLI. You can also list all of your jobs to view them at a glance.

To submit a new job, use `start-job-run`. Provide the ID of the application that you want to run, along with job-specific properties. For Spark examples, see Using Spark configurations when you run EMR Serverless jobs. For Hive examples, see Using Hive configurations when you run EMR Serverless jobs. This command returns your `application-id`, ARN, and new `job-id`.

Each job run has a set timeout duration. If the job run exceeds this duration, EMR Serverless will automatically cancel it. The default timeout is 12 hours. When you start your job run, you can configure this timeout setting to a value that meets your job requirements. Configure the value with the `executionTimeoutMinutes` property.

```
aws emr-serverless start-job-run \
```

```
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --execution-timeout-minutes 15 \
  --job-driver '{
    "hive": {
        "query": "s3://amzn-s3-demo-bucket/scripts/create_table.sql",
        "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/
hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/hive/
warehouse"
    }
   }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "hive-site",
        "properties": {
            "hive.client.cores": "2",
            "hive.client.memory": "4GIB"
        }
    }]
}'
```

To describe a job, use `get-job-run`. This command returns job-specific configurations and the set capacity for your new job.

```
aws emr-serverless get-job-run \
--job-run-id job-id \
--application-id application-id
```

To list your jobs, use `list-job-runs`. This command returns an abbreviated set of properties that includes job type, state, and other high-level attributes. If you don't want to see all of your jobs, you can specify the maximum number of jobs you want to see, up to 50. The following example specifies that you want to see your two last job runs.

```
aws emr-serverless list-job-runs \
--max-results 2 \
--application-id application-id
```

To cancel a job, use `cancel-job-run`. Provide the `application-id` and the `job-id` of the job that you want to cancel.

```
aws emr-serverless cancel-job-run \
```

```
--job-run-id job-id \
--application-id application-id
```

For more information on how to run jobs from the AWS CLI, see the EMR Serverless API Reference.

Using shuffle-optimized disks

With Amazon EMR releases 7.1.0 and higher, you can use shuffle-optimized disks when you run Apache Spark or Hive jobs to improve performance for I/O-intensive workloads. Compared to standard disks, shuffle-optimized disks provide higher IOPS (I/O operations per second) for faster data movement and reduced latency during shuffle operations. Shuffle-optimized disks let you attach disk sizes of up to 2 TB per worker, so you can configure the appropriate capacity for your workload requirements.

Key benefits

Shuffle-optimized disks provide the following benefits.

- **High IOPS performance** – shuffle-optimized disks provide higher IOPS than standard disks, leading to more efficient and rapid data shuffling during Spark and Hive jobs and other shuffle-intensive workloads.
- **Larger disk size** – Shuffle-optimized disks support disk sizes from 20GB to 2TB per worker, so you can choose the appropriate capacity based on your workloads.

Getting started

See the following steps to use shuffle-optimized disks in your workflows.

Spark

1. Create an EMR Serverless release 7.1.0 application with the following command.

```
aws emr-serverless create-application \
  --type "SPARK" \
  --name my-application-name \
  --release-label emr-7.1.0 \
  --region <AWS_REGION>
```

2. Configure your Spark job to include the parameters `spark.emr-serverless.driver.disk.type` and/or `spark.emr-serverless.executor.disk.type` to run with shuffle-optimized disks. You can use either one or both parameters, depending on your use case.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
            "entryPointArguments": ["1"],
            "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi
            --conf spark.executor.cores=4
            --conf spark.executor.memory=20g
            --conf spark.driver.cores=4
            --conf spark.driver.memory=8g
            --conf spark.executor.instances=1
            --conf spark.emr-serverless.executor.disk.type=shuffle_optimized"
        }
    }'
```

For more information, see Spark job properties.

Hive

1. Create an EMR Serverless release 7.1.0 application with the following command.

```
aws emr-serverless create-application \
  --type "HIVE" \
  --name my-application-name \
  --release-label emr-7.1.0 \
  --region <AWS_REGION>
```

2. Configure your Hive job to include the parameters `hive.driver.disk.type` and/or `hive.tez.disk.type` to run with shuffle-optimized disks. You can use either one or both parameters, depending on your use case.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
```

```
    --job-driver '{
        "hive": {
            "query": "s3://<DOC-EXAMPLE-BUCKET>/emr-serverless-hive/query/hive-
query.ql",
            "parameters": "--hiveconf hive.log.explain.output=false"
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "hive-site",
            "properties": {
                "hive.exec.scratchdir": "s3://<DOC-EXAMPLE-BUCKET>/emr-
serverless-hive/hive/scratch",
                "hive.metastore.warehouse.dir": "s3://<DOC-EXAMPLE-BUCKET>/emr-
serverless-hive/hive/warehouse",
                "hive.driver.cores": "2",
                "hive.driver.memory": "4g",
                "hive.tez.container.size": "4096",
                "hive.tez.cpu.vcores": "1",
                "hive.driver.disk.type": "shuffle_optimized",
                "hive.tez.disk.type": "shuffle_optimized"
            }
        }]
    }'
```

For more information, Hive job properties.

Configuring an application with pre-initialized capacity

See the following examples to create applications based on Amazon EMR release 7.1.0. These applications have the following properties:

- 5 pre-initialized Spark drivers, each with 2 vCPU, 4 GB of memory, and 50 GB of shuffle-optimized disk.
- 50 pre-initialized executors, each with 4 vCPU, 8 GB of memory, and 500 GB of shuffle-optimized disk.

When this application runs Spark jobs, it first consumes the pre-initialized workers and then scales the on-demand workers up to the maximum capacity of 400 vCPU and 1024 GB of memory. Optionally, you can omit capacity for either `DRIVER` or `EXECUTOR`.

Spark

```
aws emr-serverless create-application \
  --type "SPARK" \
  --name <my-application-name> \
  --release-label emr-7.1.0 \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB",
            "disk": "50GB",
            "diskType": "SHUFFLE_OPTIMIZED"
        }
    },
    "EXECUTOR": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB",
            "disk": "500GB",
            "diskType": "SHUFFLE_OPTIMIZED"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
  }'
```

Hive

```
aws emr-serverless create-application \
  --type "HIVE" \
  --name <my-application-name> \
  --release-label emr-7.1.0 \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB",
```

```
            "disk": "50GB",
            "diskType": "SHUFFLE_OPTIMIZED"
        }
    },
    "EXECUTOR": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB",
            "disk": "500GB",
            "diskType": "SHUFFLE_OPTIMIZED"
        }
    }
}' \
--maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
}'
```

Streaming jobs for processing continuously streamed data

A streaming job in EMR Serverless is a job mode that lets you analyze and process streaming data in near real-time. These long-running jobs poll streaming data and continuously process results as data arrives. Streaming jobs are best suited for tasks that require real-time data processing, such as near real-time analytics, fraud detection, and recommendations engines. EMR Serverless streaming jobs provide optimizations, such as built-in job resiliency, real-time monitoring, enhanced log management, and integration with streaming connectors.

The following are some use cases with streaming jobs:

- **Near real-time analytics** – streaming jobs in Amazon EMR Serverless let you process streaming data in near real-time, so you can perform real-time analytics on continuous data streams, such as log data, sensor data, or clickstream data to derive insights and make timely decisions based on the latest information.
- **Fraud detection** – you can use streaming jobs to run near real-time fraud detection in financial transactions, credit card operations, or online activities when you analyze data streams and identify suspicious patterns or anomalies as they occur.

- **Recommendation engines** – streaming jobs can process user-activity data and update recommendations models. Doing so opens up possibilities for personalized and real-time recommendations based on behaviors and preferences.
- **Social media analytics** – streaming jobs can process social media data, such as tweets, comments, and posts, so organizations can monitor trends, sentiment analysis, and manage brand reputation in near real-time.
- **Internet of Things (IoT) analytics** – streaming jobs can handle and analyze high-velocity streams of data from IoT devices, sensors, and connected machinery, so you can run anomaly detection, predictive maintenance, and other IoT analytics use cases.
- **Clickstream analysis** – streaming jobs can process and analyze clickstream data from websites or mobile applications. Businesses that use such data can run analytics to learn more about user behavior, personalize user experiences, and optimize marketing campaigns.
- **Log monitoring and analysis** – streaming jobs can also process log data from servers, applications, and network devices. This provides you with anomaly detection, troubleshooting, and system health and performance.

Key benefits

Streaming jobs in EMR Serverless automatically provide *job-resiliency*, which is a combination of the following factors:

- **Auto-retry** – EMR Serverless automatically retries any jobs that failed without any manual input from you.
- **Availability Zone (AZ) resiliency** – EMR Serverless automatically switches streaming jobs to a healthy AZ if the original AZ experiences issues.
- **Log management:**
  - **Log rotation** – for more efficient disk storage management, EMR Serverless periodically rotates logs for long streaming jobs. Doing so prevents log accumulation that might consume all of the disk space.
  - **Log compaction** – helps you efficiently manage and optimize log files in managed-persistence. Compaction also improves the debug experience when you use the managed spark history server.

Supported data sources and data sinks

EMR Serverless works with a number of input data sources and output data sinks:

- Supported input data sources – Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka, and self-managed Apache Kafka clusters. By default, Amazon EMR releases 7.1.0 and higher include the Amazon Kinesis Data Streams connector, so you don't need to build or download any additional packages.
- Supported output data sinks – AWS Glue Data Catalog tables, Amazon S3, Amazon Redshift, MySQL, PostgreSQL Oracle, Oracle, Microsoft SQL, Apache Iceberg, Delta Lake, and Apache Hudi.

Considerations and limitations

When you use streaming jobs, keep in mind the following considerations and limitations.

- Streaming jobs are supported with Amazon EMR releases 7.1.0 and higher.
- EMR Serverless expects streaming jobs to run for a long time, so you can't set execution timeout to limit the runtime of the job.
- Streaming jobs are only compatible with the Spark engine, which is built on-top of the structured streaming framework.
- EMR Serverless indefinitely retries streaming jobs, and you can't customize the number of maximum attempts. Thrash prevention is automatically included to stop the job retry if the amount of failed attempts has surpassed a threshold set over an hourly window. The default threshold is five failed attempts over one hour. You can configure this threshold to be between 1 and 10 attempts. For more information, see Job resiliency.
- Streaming jobs have checkpoints to save runtime state and progress, so EMR Serverless can resume the streaming job from the latest checkpoint. For more information, see Recovering from failures with Checkpointing in the Apache Spark documentation.

Getting started streaming jobs

See the following instructions to learn how to get started with streaming jobs.

1. Follow Getting started with Amazon EMR Serverless to create an application. Note that your application must run Amazon EMR release 7.1.0 or higher.
2. Once your application is ready, set the `mode` parameter to `STREAMING` to submit a streaming job, similar to the following AWS CLI example.

```
aws emr-serverless start-job-run \
--application-id <APPPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'STREAMING' \
--job-driver '{
    "sparkSubmit": {
        "entryPoint": "s3://<streaming script>",
        "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],
        "sparkSubmitParameters": "--conf spark.executor.cores=4
            --conf spark.executor.memory=16g
            --conf spark.driver.cores=4
            --conf spark.driver.memory=16g
            --conf spark.executor.instances=3"
    }
}'
```

Supported streaming connectors

Streaming connectors facilitate reading data from a streaming source and can also write data to a streaming sink.

The following are the supported streaming connectors:

Amazon Kinesis Data Streams connector

The Amazon Kinesis Data Streams connector for Apache Spark enables building streaming applications and pipelines that consume data from and write data to Amazon Kinesis Data Streams. The connector supports enhanced fan-out consumption with a dedicated read throughput rate of up to 2MB/second per shard. By default, Amazon EMR Serverless 7.1.0 and higher includes the connector, so you don't need to build or download any additional packages. For more information about the connector, see the spark-sql-kinesis-connector page on GitHub.

The following is an example of how to start a job run with the Kinesis Data Streams connector dependency.

```
aws emr-serverless start-job-run \
--application-id <APPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'STREAMING' \
--job-driver '{
```

```
    "sparkSubmit": {
        "entryPoint": "s3://<Kinesis-streaming-script>",
        "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],
        "sparkSubmitParameters": "--conf spark.executor.cores=4
                --conf spark.executor.memory=16g
                --conf spark.driver.cores=4
                --conf spark.driver.memory=16g
                --conf spark.executor.instances=3
                --jars /usr/share/aws/kinesis/spark-sql-kinesis/lib/spark-streaming-
sql-kinesis-connector.jar"
    }
}'
```

To connect to Kinesis Data Streams, you must configure the EMR Serverless application with VPC access and use a VPC endpoint to allow private access. or use a NAT Gateway to get public access. For more information, see Configuring VPC access. You must also make sure that your job runtime role has the necessary read and write permissions to access the required data streams. To learn more about how to configure a job runtime role, see Job runtime roles for Amazon EMR Serverless. For a full list of all of the required permissions, see the spark-sql-kinesis-connector page on GitHub.

Apache Kafka connector

The Apache Kafka connector for Spark structured streaming is an open-source connector from the Spark community and is available in a Maven repository. This connector facilitates Spark structured streaming applications to read data from and write data to self-managed Apache Kafka and Amazon Managed Streaming for Apache Kafka. For more information about the connector, see the Structured Streaming + Kafka Integration Guide in the Apache Spark documentation.

The following example demonstrates how to include the Kafka connector in your job run request.

```
aws emr-serverless start-job-run \
--application-id <APPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'STREAMING' \
--job-driver '{
    "sparkSubmit": {
        "entryPoint": "s3://<Kafka-streaming-script>",
        "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],
        "sparkSubmitParameters": "--conf spark.executor.cores=4
                --conf spark.executor.memory=16g
```

```
                --conf spark.driver.cores=4
                --conf spark.driver.memory=16g
                --conf spark.executor.instances=3
                --packages org.apache.spark:spark-sql-
kafka-0-10_2.12:<KAFKA_CONNECTOR_VERSION>"
    }
}'
```

The Apache Kafka connector version depends on your EMR Serverless release version and corresponding Spark version. To find the correct Kafka version, see the see the Structured Streaming + Kafka Integration Guide.

To use Amazon Managed Streaming for Apache Kafka with IAM authentication, you must include another dependency to enable the Kafka connector to connect to Amazon MSK with IAM. For more information, see the aws-msk-iam-auth repository on GitHub. You must also make sure that the job runtime role has the necessary IAM permissions. The following example demonstrates how to use the connector with IAM authentication.

```
aws emr-serverless start-job-run \
--application-id <APPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'STREAMING' \
--job-driver '{
    "sparkSubmit": {
        "entryPoint": "s3://<Kafka-streaming-script>",
        "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],
        "sparkSubmitParameters": "--conf spark.executor.cores=4
                --conf spark.executor.memory=16g
                --conf spark.driver.cores=4
                --conf spark.driver.memory=16g
                --conf spark.executor.instances=3
                --packages org.apache.spark:spark-sql-
kafka-0-10_2.12:<KAFKA_CONNECTOR_VERSION>,software.amazon.msk:aws-msk-iam-
auth:<MSK_IAM_LIB_VERSION>"
    }
}'
```

To use the Kafka connector and the IAM authentication library from Amazon MSK you must configure the EMR Serverless application with VPC access. Your subnets must have Internet access and use a NAT Gateway to access the the Maven dependencies. For more information, see Configuring VPC access. The subnets must have network connectivity to access the Kafka

cluster. This is true regardless of whether your Kafka cluster is self-managed or if you use Amazon Managed Streaming for Apache Kafka.

Streaming job log management

Streaming jobs support log rotation for Spark application logs and event logs, and log compaction for Spark event logs. This helps you manage your resources effectively.

Log rotation

Streaming jobs support log rotation for Spark application logs and event logs. Log rotation prevents long streaming jobs from generating large log files that might take up all of your available disk space. Log rotation helps you save disk storage and prevents job failures because of low disk space. For more information, see Rotating logs.

Log compaction

Streaming jobs also support log compaction for Spark event logs whenever managed logging is available. For more details about managed logging, see Logging with managed storage. Streaming jobs can run for a long time, and the amount of event data can build up over time and significantly increase log file sizes. The Spark History Server reads and loads these events into memory for the Spark application UI. This process can cause high latencies and costs, especially if event logs stored in Amazon S3 are very large.

Log compaction reduces the event log size, so the Spark History Server doesn't have to load more than 1 GB of event logs at any time. For more information, see Monitoring and Instrumentation in the Apache Spark documentation.

Using Spark configurations when you run EMR Serverless jobs

You can run Spark jobs on an application with the `type` parameter set to SPARK. Jobs must be compatible with the Spark version compatible with the Amazon EMR release version. For example, when you run jobs with Amazon EMR release 6.6.0, your job must be compatible with Apache Spark 3.2.0. For information on the application versions for each release, see Amazon EMR Serverless release versions.

Spark job parameters

When you use the `StartJobRun` API to run a Spark job, you can specify the following parameters.

Required parameters

- Spark job runtime role
- Spark job driver parameter
- Spark configuration override parameter
- Spark dynamic resource allocation optimization

Spark job runtime role

Use **`executionRoleArn`** to specify the ARN for the IAM role that your application uses to execute Spark jobs. This role must contain the following permissions:

- Read from S3 buckets or other data sources where your data resides
- Read from S3 buckets or prefixes where your PySpark script or JAR file resides
- Write to S3 buckets where you intend to write your final output
- Write logs to a S3 bucket or prefix that `S3MonitoringConfiguration`specifies
- Access to KMS keys if you use KMS keys to encrypt data in your S3 bucket
- Access to the AWS Glue Data Catalog if you use SparkSQL

If your Spark job reads or writes data to or from other data sources, specify the appropriate permissions in this IAM role. If you don't provide these permissions to the IAM role, the job might fail. For more information, see Job runtime roles for Amazon EMR Serverless and Storing logs.

Spark job driver parameter

Use **`jobDriver`** to provide input to the job. The job driver parameter accepts only one value for the job type that you want to run. For a Spark job, the parameter value is `sparkSubmit`. You can use this job type to run Scala, Java, PySpark, SparkR, and any other supported jobs through Spark submit. Spark jobs have the following parameters:

- **`sparkSubmitParameters`** – These are the additional Spark parameters that you want to send to the job. Use this parameter to override default Spark properties such as driver memory or number of executors, like those defined in the `--conf` or `--class` arguments.
- **`entryPointArguments`** – This is an array of arguments that you want to pass to your main JAR or Python file. You should handle reading these parameters using your entrypoint code. Separate each argument in the array by a comma.

- **`entryPoint`** – This is the reference in Amazon S3 to the main JAR or Python file that you want to run. If you are running a Scala or Java JAR, specify the main entry class in the `SparkSubmitParameters` using the `--class` argument.

For additional information, see Launching Applications with spark-submit.

Spark configuration override parameter

Use **`configurationOverrides`** to override monitoring-level and application-level configuration properties. This parameter accepts a JSON object with the following two fields:

- **`monitoringConfiguration`** - Use this field to specify the Amazon S3 URL (`s3MonitoringConfiguration`) where you want the EMR Serverless job to store logs of your Spark job. Make sure you've created this bucket with the same AWS account that hosts your application, and in the same AWS Region where your job is running.
- **`applicationConfiguration`** – To override the default configurations for applications, you can provide a configuration object in this field. You can use a shorthand syntax to provide the configuration, or you can reference the configuration object in a JSON file. Configuration objects consist of a classification, properties, and optional nested configurations. Properties consist of the settings that you want to override in that file. You can specify multiple classifications for multiple applications in a single JSON object.

> **Note**
>
> Available configuration classifications vary by specific EMR Serverless release. For example, classifications for custom Log4j `spark-driver-log4j2` and `spark-executor-log4j2` are only available with releases 6.8.0 and higher.

If you use the same configuration in an application override and in Spark submit parameters, the Spark submit parameters take priority. Configurations rank in priority as follows, from highest to lowest:

- Configuration that EMR Serverless provides when it creates `SparkSession`.
- Configuration that you provide as part of `sparkSubmitParameters` with the `--conf` argument.
- Configuration that you provide as part of your application overrides when you start a job.

- Configuration that you provide as part of your `runtimeConfiguration` when you create an application.
- Optimized configurations that Amazon EMR uses for the release.
- Default open source configurations for the application.

For more information on declaring configurations at the application level, and overriding configurations during job run, see Default application configuration for EMR Serverless.

Spark dynamic resource allocation optimization

Use `dynamicAllocationOptimization` to optimize resource usage in EMR Serverless. Setting this property to `true` in your Spark configuration classification indicates to EMR Serverless to optimize executor resource allocation to better align the rate at which Spark requests and cancels executors with the rate at which EMR Serverless creates and releases workers. By doing so, EMR Serverless more optimally reuses workers across stages, resulting in lower cost when running jobs with multiple stages while maintaining the same performance.

This property is available in all Amazon EMR release versions.

The following is a sample configuration classification with `dynamicAllocationOptimization`.

```
[
  {
    "Classification": "spark",
    "Properties": {
      "dynamicAllocationOptimization": "true"
    }
  }
]
```

Consider the following if you're using dynamic allocation optimization:

- This optimization is available for the Spark jobs for which you enabled dynamic resource allocation.
- To achieve the best cost efficiency, we recommend configuring an upper scaling bound on workers using either the job-level setting `spark.dynamicAllocation.maxExecutors` or the application-level maxium capacity setting based on your workload.

- You might not see cost improvement in simpler jobs. For example, if your job runs on a small dataset or finishes running in one stage, Spark might not need a larger number of executors or multiple scaling events.
- Jobs with a sequence of a large stage, smaller stages, and then a large stage again might experience regression in job runtime. As EMR Serverless uses resources more efficiently, it might lead to fewer available workers for larger stages, leading to longer runtime.

Spark job properties

The following table lists optional Spark properties and their default values that you can override when you submit a Spark job.

Key	Description	Default value
`spark.archives`	A comma-separated list of archives that Spark extracts into each executor's working directory. Supported file types include `.jar`, `.tar.gz`, `.tgz` and `.zip`. To specify the directory name to extract, add # after the file name that you want to extract. For example, `file.zip#directory` .	`NULL`
`spark.authenticate`	Option that turns on authentication of Spark's internal connections.	`TRUE`
`spark.driver.cores`	The number of cores that the driver uses.	4
`spark.driver.extra JavaOptions`	Extra Java options for the Spark driver.	`NULL`

Key	Description	Default value
spark.driver.memory	The amount of memory that the driver uses.	14G
spark.dynamicAllocation.enabled	Option that turns on dynamic resource allocation. This option scales up or down the number of executors registered with the application, based on the workload.	TRUE
spark.dynamicAllocation.executorIdleTimeout	The length of time that an executor can remain idle before Spark removes it. This only applies if you turn on dynamic allocation.	60s
spark.dynamicAllocation.initialExecutors	The initial number of executors to run if you turn on dynamic allocation.	3
spark.dynamicAllocation.maxExecutors	The upper bound for the number of executors if you turn on dynamic allocation.	For 6.10.0 and higher, infinity For 6.9.0 and lower, 100
spark.dynamicAllocation.minExecutors	The lower bound for the number of executors if you turn on dynamic allocation.	0
spark.emr-serverless.allocation.batch.size	The number of containers to request in each cycle of executor allocation. There is a one-second gap between each allocation cycle.	20

Key	Description	Default value
spark.emr-serverless.driver.disk	The Spark driver disk.	20G
spark.emr-serverless.driverEnv. *[KEY]*	Option that adds environment variables to the Spark driver.	NULL
spark.emr-serverless.executor.disk	The Spark executor disk.	20G
spark.emr-serverless.memoryOverheadFactor	Sets the memory overhead to add to the driver and executor container memory.	0.1
spark.emr-serverless.driver.disk.type	The disk type attached to Spark driver.	Standard
spark.emr-serverless.executor.disk.type	The disk type attached to Spark executors.	Standard
spark.executor.cores	The number of cores that each executor uses.	4
spark.executor.extraJavaOptions	Extra Java options for the Spark executor.	NULL
spark.executor.instances	The number of Spark executor containers to allocate.	3
spark.executor.memory	The amount of memory that each executor uses.	14G
spark.executorEnv. *[KEY]*	Option that adds environment variables to the Spark executors.	NULL

Key	Description	Default value
`spark.files`	A comma-separated list of files to go in the working directory of each executor. You can access the file paths of these files in the executor with `SparkFiles.get( fileName)`.	`NULL`
`spark.hadoop.hive.metastore.client.factory.class`	The Hive metastore implementation class.	`NULL`
`spark.jars`	Additional jars to add to the runtime classpath of the driver and executors.	`NULL`
`spark.network.crypto.enabled`	Option that turns on AES-based RPC encryption. This includes the authentication protocol added in Spark 2.2.0.	`FALSE`
`spark.sql.warehouse.dir`	The default location for managed databases and tables.	The value of `$PWD/spark-warehouse`
`spark.submit.pyFiles`	A comma-separated list of `.zip`, `.egg`, or `.py` files to place in the `PYTHONPATH` for Python apps.	`NULL`

The following table lists the default Spark submit parameters.

Key	Description	Default value
archives	A comma-separated list of archives that Spark extracts into each executor's working directory.	NULL
class	The application's main class (for Java and Scala apps).	NULL
conf	An arbitrary Spark configuration property.	NULL
driver-cores	The number of cores that the driver uses.	4
driver-memory	The amount of memory that the driver uses.	14G
executor-cores	The number of cores that each executor uses.	4
executor-memory	The amount of memory that the executor uses.	14G
files	A comma-separated list of files to place in the working directory of each executor. You can access the file paths of these files in the executor with SparkFiles.get(*fileName*).	NULL
jars	A comma-separated list of jars to include on the driver and executor classpaths.	NULL
num-executors	The number of executors to launch.	3

Key	Description	Default value
py-files	A comma-separated list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps.	NULL
verbose	Option that turns on additional debug output.	NULL

Spark examples

The following example shows how to use the StartJobRun API to run a Python script. For an end-to-end tutorial that uses this example, see Getting started with Amazon EMR Serverless. You can find additional examples of how to run PySpark jobs and add Python dependencies in the EMR Serverless Samples GitHub repository.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
            "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket/
wordcount_output"],
            "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf
 spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g --
conf spark.executor.instances=1"
        }
    }'
```

The following example shows how to use the StartJobRun API to run a Spark JAR.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
```

```
            "entryPointArguments": ["1"],
            "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi --conf
 spark.executor.cores=4 --conf spark.executor.memory=20g --conf spark.driver.cores=4 --
conf spark.driver.memory=8g --conf spark.executor.instances=1"
        }
    }'
```

Using Hive configurations when you run EMR Serverless jobs

You can run Hive jobs on an application with the `type` parameter set to `HIVE`. Jobs must be compatible with the Hive version compatible with the Amazon EMR release version. For example, when you run jobs on an application with Amazon EMR release 6.6.0, your job must be compatible with Apache Hive 3.1.2. For information on the application versions for each release, see Amazon EMR Serverless release versions.

Hive job parameters

When you use the `StartJobRun` API to run a Hive job, you must specify the following parameters.

Required parameters

- Hive job runtime role
- Hive job driver parameter
- Hive configuration override parameter

Hive job runtime role

Use **`executionRoleArn`** to specify the ARN for the IAM role that your application uses to execute Hive jobs. This role must contain the following permissions:

- Read from S3 buckets or other data sources where your data resides
- Read from S3 buckets or prefixes where your Hive query file and init query file reside
- Read and write to S3 buckets where your Hive Scratch directory and Hive Metastore warehouse directory reside
- Write to S3 buckets where you intend to write your final output
- Write logs to an S3 bucket or prefix that `S3MonitoringConfiguration` specifies
- Access to KMS keys if you use KMS keys to encrypt data in your S3 bucket

- Access to the AWS Glue Data Catalog

If your Hive job reads or writes data to or from other data sources, specify the appropriate permissions in this IAM role. If you don't provide these permissions to the IAM role, your job might fail. For more information, see Job runtime roles for Amazon EMR Serverless.

Hive job driver parameter

Use **`jobDriver`** to provide input to the job. The job driver parameter accepts only one value for the job type that you want to run. When you specify `hive` as the job type, EMR Serverless passes a Hive query to the `jobDriver` parameter. Hive jobs have the following parameters:

- **`query`** – This is the reference in Amazon S3 to the Hive query file that you want to run.
- **`parameters`** – These are the additional Hive configuration properties that you want to override. To override properties, pass them to this parameter as `--hiveconf` *`property=value`*. To override variables, pass them to this parameter as `--hivevar` *`key=value`*.
- **`initQueryFile`** – This is the init Hive query file. Hive runs this file prior to your query and can use it to initialize tables.

Hive configuration override parameter

Use **`configurationOverrides`** to override monitoring-level and application-level configuration properties. This parameters accepts a JSON object with the following two fields:

- **`monitoringConfiguration`** – Use this field to specify the Amazon S3 URL (`s3MonitoringConfiguration`) where you want the EMR Serverless job to store logs of your Hive job. Make sure that you create this bucket with the same AWS account that hosts your application, and in the same AWS Region where your job is running.
- **`applicationConfiguration`** – You can provide a configuration object in this field to override the default configurations for applications. You can use a shorthand syntax to provide the configuration, or you can reference the configuration object in a JSON file. Configuration objects consist of a classification, properties, and optional nested configurations. Properties consist of the settings that you want to override in that file. You can specify multiple classifications for multiple applications in a single JSON object.

> **Note**
>
> Available configuration classifications vary by specific EMR Serverless release. For example, classifications for custom Log4j `spark-driver-log4j2` and `spark-executor-log4j2` are only available with releases 6.8.0 and higher.

If you pass the same configuration in an application override and in Hive parameters, the Hive parameters take priority. The following list ranks configurations from highest priority to lowest priority.

- Configuration that you provide as part of Hive parameters with `--hiveconf` *`property=value`*.
- Configuration that you provide as part of your application overrides when you start a job.
- Configuration that you provide as part of your `runtimeConfiguration` when you create an application.
- Optimized configurations that Amazon EMR assigns for the release.
- Default open-source configurations for the application.

For more information on declaring configurations at the application level, and overriding configurations during job run, see Default application configuration for EMR Serverless.

Hive job properties

The following table lists the mandatory properties that you must configure when you submit a Hive job.

Setting	Description
`hive.exec.scratchdir`	The Amazon S3 location where EMR Serverless creates temporary files during the Hive job execution.
`hive.metastore.warehouse.dir`	The Amazon S3 location of databases for managed tables in Hive.

The following table lists the optional Hive properties and their default values that you can override when you submit a Hive job.

Setting	Description	Default value
`fs.s3.customAWSCredentialsProvider`	The AWS Credentials provider you want to use.	com.amazonaws.auth.DefaultAWSCredentialsProviderChain
`fs.s3a.aws.credentials.provider`	The AWS Credentials provider you want to use with a S3A file system.	com.amazonaws.auth.DefaultAWSCredentialsProviderChain
`hive.auto.convert.join`	Option that turns on auto-conversion of common joins into mapjoins, based on the input file size.	TRUE
`hive.auto.convert.join.noconditionaltask`	Option that turns on optimization when Hive converts a common join into a mapjoin based on the input file size.	TRUE
`hive.auto.convert.join.noconditionaltask.size`	A join converts directly to a mapjoin below this size.	Optimal value is calculated based on Tez task memory
`hive.cbo.enable`	Option that turns on cost-based optimizations with the Calcite framework.	TRUE
`hive.cli.tez.session.async`	Option to start a background Tez session while your Hive query compiles. When set to `false`, Tez AM launches after your Hive query compiles.	TRUE

Setting	Description	Default value
`hive.compute.query.using.stats`	Option that activates Hive to answer certain queries with statistics stored in the metastore. For basic statistics, set `hive.stats.autogather` to TRUE. For a more advanced collection of queries, run `analyze table queries`.	TRUE
`hive.default.fileformat`	The default file format for `CREATE TABLE` statements. You can explicitly override this if you specify `STORED AS [FORMAT]` in your `CREATE TABLE` command.	`TEXTFILE`
`hive.driver.cores`	The number of cores to use for the Hive driver process.	2
`hive.driver.disk`	The disk size for the Hive driver.	20G
`hive.driver.disk.type`	The disk type for the Hive driver.	Standard
`hive.tez.disk.type`	The disk size for the tez workers.	Standard
`hive.driver.memory`	The amount of memory to use per Hive driver process. The Hive CLI and Tez Application Master share this memory equally with 20% of headroom.	6G

Setting	Description	Default value
`hive.emr-serverless.launch.env.[KEY]`	Option to set the *KEY* environment variable in all Hive-specific processes, such as your Hive driver, Tez AM, and Tez task.	
`hive.exec.dynamic.partition`	Options that turns on dynamic partitions in DML/DDL.	TRUE
`hive.exec.dynamic.partition.mode`	Option that specifies whether you want to use strict mode or non-strict mode. In strict mode, you must specify at least one static partition in case you accidentally overwrite all partitions. In non-strict mode, all partitions are allowed to be dynamic.	`strict`
`hive.exec.max.dynamic.partitions`	The maximum number of dynamic partitions that Hive creates in total.	1000
`hive.exec.max.dynamic.partitions.pernode`	Maximum number of dynamic partitions that Hive creates in each mapper and reducer node.	100

Setting	Description	Default value
`hive.exec.orc.split.strategy`	Expects one of the following values: `BI`, `ETL`, or `HYBRID`. This isn't a user-level configuration. `BI` specifies that you want to spend less time in split generation as opposed to query execution. `ETL` specifies that you want to spend more time in split generation. `HYBRID` specifies a choice of the above strategies based on heuristics.	`HYBRID`
`hive.exec.reducers.bytes.per.reducer`	The size per reducer. The default is 256 MB. If the input size is 1G, the job uses 4 reducers.	256000000
`hive.exec.reducers.max`	The maximum number of reducers.	256
`hive.exec.stagingdir`	The name of the directory that stores temporary files that Hive creates inside table locations and in the scratch directory location specified in the `hive.exec.scratchdir` property.	`.hive-staging`
`hive.fetch.task.conversion`	Expects one of the following values: `NONE`, `MINIMAL`, or `MORE`. Hive can convert select queries to a single `FETCH` task. This minimizes latency.	`MORE`

Setting	Description	Default value
hive.groupby.position.alias	Option that causes Hive to use a column position alias in GROUP BY statements.	FALSE
hive.input.format	The default input format. Set to HiveInputFormat if you encounter problems with CombineHiveInputFormat .	org.apache.hadoop.hive.ql.io.CombineHiveInputFormat
hive.log.explain.output	Option that turns on explanations of extended output for any query in your Hive log.	FALSE
hive.log.level	The Hive logging level.	INFO
hive.mapred.reduce.tasks.speculative.execution	Option that turns on speculative launch for reducers. Only supported with Amazon EMR 6.10.x and lower.	TRUE
hive.max-task-containers	The maximum number of concurrent containers. The configured mapper memory is multiplied by this value to determine available memory that split computation and task preemption use.	1000
hive.merge.mapfiles	Option that causes small files to merge at the end of a map-only job.	TRUE
hive.merge.size.per.task	The size of merged files at the end of the job.	256000000

Setting	Description	Default value
hive.merge.tezfiles	Option that turns on a merge of small files at the end of a Tez DAG.	FALSE
hive.metastore.client.factory.class	The name of the factory class that produces objects that implement the IMetaStoreClient interface.	com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory
hive.metastore.glue.catalogid	If the AWS Glue Data Catalog acts as a metastore but runs in a different AWS account than the jobs, the ID of the AWS account where the jobs are running.	NULL
hive.metastore.uris	The thrift URI that the metastore client uses to connect to remote metastore.	NULL
hive.optimize.ppd	Option that turns on predicate pushdown.	TRUE
hive.optimize.ppd.storage	Option that turns on predicate pushdown to storage handlers.	TRUE
hive.orderby.position.alias	Option that causes Hive to use a column position alias in ORDER BY statements.	TRUE
hive.prewarm.enabled	Option that turns on container prewarm for Tez.	FALSE
hive.prewarm.numcontainers	The number of containers to pre-warm for Tez.	10

Setting	Description	Default value
hive.stats.autogather	Option that causes Hive to gather basic statistics automatically during the INSERT OVERWRITE command.	TRUE
hive.stats.fetch.column.stats	Option that turns off the fetch of column statistics from the metastore. A fetch of column statistics can be expensive when the number of columns is high.	FALSE
hive.stats.gather.num.threads	The number of threads that the partialscan and noscan analyze commands use for partitioned tables. This only applies to file formats that implement StatsProvidingRecordReader (like ORC).	10
hive.strict.checks.cartesian.product	Options that turns on strict Cartesian join checks. These checks disallow a Cartesian product (a cross join).	FALSE
hive.strict.checks.type.safety	Option that turns on strict type safety checks and turns off comparison of bigint with both string and double.	TRUE

Setting	Description	Default value
hive.support.quoted.identifiers	Expects value of NONE or COLUMN. NONE implies only alphanumeric and underscore characters are valid in identifiers. COLUMN implies column names can contain any character.	COLUMN
hive.tez.auto.reducer.parallelism	Option that turns on the Tez auto-reducer parallelism feature. Hive still estimates data sizes and sets parallelism estimates. Tez samples the output sizes of source vertices and adjusts the estimates at runtime as necessary.	TRUE
hive.tez.container.size	The amount of memory to use per Tez task process.	6144
hive.tez.cpu.vcores	The number of cores to use for each Tez task.	2
hive.tez.disk.size	The disk size for each task container.	20G
hive.tez.input.format	The input format for splits generation in the Tez AM.	org.apache.hadoop.hive.ql.io.HiveInputFormat
hive.tez.min.partition.factor	Lower limit of reducers that Tez specifies when you turn on auto-reducer parallelism.	0.25

Setting	Description	Default value
hive.vectorized.execution.enabled	Option that turns on vectorized mode of query execution.	TRUE
hive.vectorized.execution.reduce.enabled	Option that turns on vectorized mode of a query execution's reduce-side.	TRUE
javax.jdo.option.ConnectionDriverName	The driver class name for a JDBC metastore.	org.apache.derby.jdbc.EmbeddedDriver
javax.jdo.option.ConnectionPassword	The password associated with a metastore database.	NULL
javax.jdo.option.ConnectionURL	The JDBC connect string for a JDBC metastore.	jdbc:derby:;databaseName=metastore_db;create=true
javax.jdo.option.ConnectionUserName	The user name associated with a metastore database.	NULL
mapreduce.input.fileinputformat.split.maxsize	The maximum size of a split during split computation when your input format is org.apache.hadoop.hive.ql.io.CombineHiveInputFormat . A value of 0 indicates no limit.	0
tez.am.dag.cleanup.on.completion	Option that turns on cleanup of shuffle data when DAG completes.	TRUE

Setting	Description	Default value
`tez.am.emr-serverless.launch.env.[KEY]`	Option to set the *KEY* environment variable in the Tez AM process. For Tez AM, this value overrides the `hive.emr-serverless.launch.env.[KEY]` value.	
`tez.am.log.level`	The root logging level that EMR Serverless passes to the Tez app master.	`INFO`
`tez.am.sleep.time.before.exit.millis`	EMR Serverless should push ATS events after this period of time following AM shutdown request.	0
`tez.am.speculation.enabled`	Option that causes speculative launch of slower tasks. This can help reduce job latency when some tasks are running slower due bad or slow machines. Only supported with Amazon EMR 6.10.x and lower.	`FALSE`
`tez.am.task.max.failed.attempts`	The maximum number of attempts that can fail for a particular task before the task fails. This number doesn't count manually terminated attempts.	3

Setting	Description	Default value
tez.am.vertex.cleanup.height	A distance at which, if all dependent vertices are complete, Tez AM will delete vertex shuffle data. This feature is turned off when the value is 0. Amazon EMR versions 6.8.0 and later support this feature.	0
tez.client.asynchronous-stop	Option that causes EMR Serverless to push ATS events before it ends the Hive driver.	FALSE
tez.grouping.max-size	The upper size limit (in bytes) of a grouped split. This limit prevents excessively large splits.	1073741824
tez.grouping.min-size	The lower size limit (in bytes) of a grouped split. This limit prevents too many small splits.	16777216
tez.runtime.io.sort.mb	The size of the soft buffer when Tez sorts the output is sorted.	Optimal value is calculated based on Tez task memory
tez.runtime.unordered.output.buffer.size-mb	The size of the buffer to use if Tez doesn't write directly to disk.	Optimal value is calculated based on Tez task memory

Setting	Description	Default value
`tez.shuffle-vertex-manager.max-src-fraction`	The fraction of source tasks that must complete before EMR Serverless schedules all tasks for the current vertex (in case of a `ScatterGather` connection). The number of tasks ready for scheduling on the current vertex scales linearly between `min-fraction` and `max-fraction`. This defaults the default value or `tez.shuffle-vertex-manager.min-src-fraction`, whichever is greater.	0.75
`tez.shuffle-vertex-manager.min-src-fraction`	The fraction of source tasks that must complete before EMR Serverless schedules tasks for the current vertex (in case of a `ScatterGather` connection).	0.25
`tez.task.emr-serverless.launch.env.[`*KEY*`]`	Option to set the *KEY* environment variable in the Tez task process. For Tez tasks, this value overrides the `hive.emr-serverless.launch.env.[`*KEY*`]` value.	
`tez.task.log.level`	The root logging level that EMR Serverless passes to the Tez tasks.	`INFO`

Setting	Description	Default value
`tez.yarn.ats.event.flush.timeout.millis`	The maximum amount of time that AM should wait for events to be flushed before shutting down.	300000

Hive job examples

The following code example shows how to run a Hive query with the `StartJobRun` API.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "hive": {
            "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.ql",
            "parameters": "--hiveconf hive.log.explain.output=false"
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "hive-site",
            "properties": {
                "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-serverless-hive/
hive/scratch",
                "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/warehouse",
                "hive.driver.cores": "2",
                "hive.driver.memory": "4g",
                "hive.tez.container.size": "4096",
                "hive.tez.cpu.vcores": "1"
            }
        }]
    }'
```

You can find additional examples of how to run Hive jobs in the EMR Serverless Samples GitHub repository.

EMR Serverless Job resiliency

EMR Serverless releases 7.1.0 and higher include support for job resiliency, so it automatically retries any failed jobs without any manual input from you. Another benefit of job resiliency is that EMR Serverless moves job runs to different Availability Zone (AZ) should an AZ experience any issues.

To enable job resiliency for a job, set the retry policy for your job. A retry policy makes sure that EMR Serverless automatically restarts a job if it fails at any point. Retry policies are supported for both batch and streaming jobs, so you can customize job resiliency according to your use case. The following table compares the behaviors and differences of job resiliency across batch and streaming jobs.

	Batch jobs	Streaming jobs
Default behavior	Doesn't rerun the job.	Always retries running the job as the application creates checkpoints while running the job.
Retry point	Batch jobs don't have checkpoints, so EMR Serverless always re-runs the job from the beginning.	Streaming jobs support checkpoints, so you can configure the streaming query to save runtime state and progress to a checkpoint location in Amazon S3. EMR Serverless resumes the job run from the checkpoint. For more information, see Recovering from failures with Checkpointing in the Apache Spark documentation.
Maximum of retry attempts	Allows for a maximum of 10 retries.	Streaming jobs have built-in thrash prevention control, so the application stops retrying jobs if they continue failing after one hour. The default

	Batch jobs	Streaming jobs
		number of retries within one hour is five attempts. You can configure this number of retries to be between 1 or 10. You can't customize the number of maximum attempts. A value of 1 indicates no retries.

When EMR Serverless attempts to rerun a job, it also indexes the job with an attempt number, so you can track the lifecycle of a job across its attempts.

You can use the EMR Serverless API operations or the AWS CLI to change job resiliency or see information related to job resiliency. For more information, see the EMR Serverless API guide.

By default, EMR Serverless doesn't rerun batch jobs. To enable retries for batch jobs, configure the `maxAttempts` parameter when starting a batch job run. The `maxAttempts` parameter is applicable only to batch jobs. The default is 1, which means to not rerun the job. Accepted values are 1 to 10, inclusive.

The following example demonstrates how to specify a max number of 10 attempts when starting a job run.

```
aws emr-serverless start-job-run
 --application-id <APPLICATION_ID> \
 --execution-role-arn <JOB_EXECUTION_ROLE> \
 --mode 'BATCH' \
 --retry-policy '{
    "maxAttempts": 10
 }' \
 --job-driver '{
    "sparkSubmit": {
         "entryPoint": "/usr/lib/spark/examples/jars/spark-examples-does-not-
exist.jar",
         "entryPointArguments": ["1"],
         "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi"
     }
}'
```

EMR Serverless indefinitely retries streaming jobs if they fail. To prevent thrashing because of repeated unrecoverable failures, use the `maxFailedAttemptsPerHour` to configure thrash prevention control for streaming job retries. This parameter lets you specify the maximum number of failed attempts allowed with an hour before EMR Serverless stops retrying. The default is five. Accepted values are 1 to 10, inclusive.

```
aws emr-serverless start-job-run
 --application-id <APPPLICATION_ID> \
 --execution-role-arn <JOB_EXECUTION_ROLE> \
 --mode 'STREAMING' \
 --retry-policy '{
    "maxFailedAttemptsPerHour": 7
 }' \
 --job-driver '{
    "sparkSubmit": {
         "entryPoint": "/usr/lib/spark/examples/jars/spark-examples-does-not-
exist.jar",
         "entryPointArguments": ["1"],
         "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi"
     }
}'
```

You can also use the other job run API operations get information about jobs. For example, you can use the `attempt` parameter with the `GetJobRun` operation to get details about a specific job attempt. If you don't include the `attempt` parameter, the operation returns information about the latest attempt.

```
aws emr-serverless get-job-run \
    --job-run-id job-run-id \
    --application-id application-id \
    --attempt 1
```

The `ListJobRunAttempts` operation returns information about all attempts related to a job run.

```
aws emr-serverless list-job-run-attempts \
  --application-id application-id \
  --job-run-id job-run-id
```

The `GetDashboardForJobRun` operation creates and returns a URL that you can use to access the application UIs for a job run. The `attempt` parameter lets you get a URL for a specific attempt.

If you don't include the `attempt` parameter, the operation returns information about the latest attempt.

```
aws emr-serverless get-dashboard-for-job-run \
    --application-id application-id \
    --job-run-id job-run-id \
    --attempt 1
```

Monitoring a job with a retry policy

Job resiliency support also adds the new event **EMR Serverless job run retry**. EMR Serverless publishes this event on every retry of the job. You can use this notification to track retries of the job. For more information about events, see Amazon EventBridge events.

Logging with retry policy

Every time EMR Serverless retries a job, the attempt generates its own set of logs. To ensure that EMR Serverless can successfully deliver these logs to Amazon S3 and Amazon CloudWatch without overwriting any, EMR Serverless adds a prefix to the format of the S3 log path and CloudWatch log stream name to include the attempt number of the job.

The following is an example of what the format looks like.

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'.
```

This format ensures EMR Serverless publishes all of the logs for each attempt of the job to its own designated location in Amazon S3 and CloudWatch. For more details, see Storing logs.

> **Note**
>
> EMR Serverless only uses this prefix format with all streaming jobs and any batch jobs that have retry enabled.

Metastore configuration for EMR Serverless

A *Hive metastore* is a centralized location that stores structural information about your tables, including schemas, partition names, and data types. With EMR Serverless, you can persist this table metadata in a metastore that has access to your jobs.

You have two options for a Hive metastore:

- The AWS Glue Data Catalog
- An external Apache Hive metastore

Using the AWS Glue Data Catalog as a metastore

You can configure your Spark and Hive jobs to use the AWS Glue Data Catalog as its metastore. We recommend this configuration when you require a persistent metastore or a metastore shared by different applications, services, or AWS accounts. For more information about the Data Catalog, see Populating the AWS Glue Data Catalog. For information about AWS Glue pricing, see AWS Glue pricing.

You can configure your EMR Serverless job to use the AWS Glue Data Catalog either in the same AWS account as your application, or in a different AWS account.

Configure the AWS Glue Data Catalog

To configure the Data Catalog, choose which type of EMR Serverless application that you want to use.

Spark

When you use EMR Studio to run your jobs with EMR Serverless Spark applications, the AWS Glue Data Catalog is the default metastore.

When you use SDKs or AWS CLI, you can set the `spark.hadoop.hive.metastore.client.factory.class` configuration to `com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory` in the `sparkSubmit` parameters of your job run. The following example shows how to configure the Data Catalog with the AWS CLI.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://amzn-s3-demo-bucket/code/pyspark/
extreme_weather.py",
```

```
            "sparkSubmitParameters": "--conf
 spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGl
 --conf spark.driver.cores=1 --conf spark.driver.memory=3g --conf
 spark.executor.cores=4 --conf spark.executor.memory=3g"
        }
    }'
```

Alternatively, you can set this configuration when you create a new `SparkSession` in your Spark code.

```
from pyspark.sql import SparkSession

spark = (
    SparkSession.builder.appName("SparkSQL")
    .config(
        "spark.hadoop.hive.metastore.client.factory.class",
        "com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory",
    )
    .enableHiveSupport()
    .getOrCreate()
)

# we can query tables with SparkSQL
spark.sql("SHOW TABLES").show()

# we can also them with native Spark
print(spark.catalog.listTables())
```

Hive

For EMR Serverless Hive applications, the Data Catalog is the default metastore. That is, when you run jobs on a EMR Serverless Hive application, Hive records metastore information in the Data Catalog in the same AWS account as your application. You don't need a virtual private cloud (VPC) to use the Data Catalog as your metastore.

To access the Hive metastore tables, add the required AWS Glue policies outlined in Setting up IAM Permissions for AWS Glue.

Configure cross-account access for EMR Serverless and AWS Glue Data Catalog

To set up cross-account access for EMR Serverless, you must first sign in to the following AWS accounts:

- `AccountA` – An AWS account where you have created an EMR Serverless application.
- `AccountB` – An AWS account that contains a AWS Glue Data Catalog that you want your EMR Serverless job runs to access.

1. Make sure an administrator or other authorized identity in `AccountB` attaches a resource policy to the Data Catalog in `AccountB`. This policy grants `AccountA` specific cross-account permissions to perform operations on resources in the `AccountB` catalog.

```
{
  "Version" : "2012-10-17",
  "Statement" : [ {
    "Effect" : "Allow",
    "Principal": {
        "AWS": [
            "arn:aws:iam::accountA:role/job-runtime-role-A"
        ]},
    "Action" : [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue:DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
    ],
    "Resource": ["arn:aws:glue:region:AccountB:catalog"]
  } ]
}
```

2. Add an IAM policy to the EMR Serverless job runtime role in `AccountA` so that role can access Data Catalog resources in `AccountB`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue:DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
      ],
      "Resource": ["arn:aws:glue:region:AccountB:catalog"]
    }
  ]
}
```

3. Start your job run. This step is slightly different depending on `AccountA`'s EMR Serverless application type.

   Spark

   Set the `spark.hadoop.hive.metastore.glue.catalogid` property in the `hive-site` classification as shown in the following example. Replace *`AccountB-catalog-id`* with the ID of the Data Catalog in `AccountB`.

```
aws emr-serverless start-job-run \
--application-id "application-id" \
--execution-role-arn "job-role-arn" \
--job-driver '{
    "sparkSubmit": {
```

```
        "query": "s3://amzn-s3-demo-bucket/hive/scripts/create_table.sql",
        "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/
hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/
hive/warehouse"
    }
  }' \
--configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "hive-site",
        "properties": {
            "spark.hadoop.hive.metastore.glue.catalogid": "AccountB-catalog-id"
        }
    }]
}'
```

Hive

Set the `hive.metastore.glue.catalogid` property in the `hive-site` classification as shown in the following example. Replace *AccountB-catalog-id* with the ID of the Data Catalog in AccountB.

```
aws emr-serverless start-job-run \
--application-id "application-id" \
--execution-role-arn "job-role-arn" \
--job-driver '{
    "hive": {
    "query": "s3://amzn-s3-demo-bucket/hive/scripts/create_table.sql",
    "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/
hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/
hive/warehouse"
    }
}' \
--configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "hive-site",
        "properties": {
            "hive.metastore.glue.catalogid": "AccountB-catalog-id"
        }
    }]
}'
```

Considerations when using the AWS Glue Data Catalog

You can add auxiliary JARs with `ADD JAR` in your Hive scripts. For additional considerations, see Considerations when using AWS Glue Data Catalog.

Using an external Hive metastore

You can configure your EMR Serverless Spark and Hive jobs to connect to an external Hive metastore, such as Amazon Aurora or Amazon RDS for MySQL. This section describes how to set up an Amazon RDS Hive metastore, configure your VPC, and configure your EMR Serverless jobs to use an external metastore.

Create an external Hive metastore

1. Create an Amazon Virtual Private Cloud (Amazon VPC) with private subnets by following the instructions in Create a VPC.
2. Create your EMR Serverless application with your new Amazon VPC and private subnets. When you configure your EMR Serverless application with a VPC, it first provisions an elastic network interface for each subnet that you specify. It then attaches your specified security group to that network interface. This gives your application access control. For more details about how to set up your VPC, see Configuring VPC access for EMR Serverless applications to connect to data.
3. Create a MySQL or Aurora PostgreSQL database in a private subnet in your Amazon VPC. For information about how to create an Amazon RDS database, see Creating an Amazon RDS DB instance.
4. Modify the security group of your MySQL or Aurora database to allow JDBC connections from your EMR Serverless security group by following the steps in Modifying an Amazon RDS DB instance. Add a rule for inbound traffic to the RDS security group from one of your EMR Serverless security groups.

Type	Protocol	Port range	Source
All TCP	TCP	3306	`emr-serverless-security-group`

Configure Spark options

Using JDBC

To configure your EMR Serverless Spark application to connect to a Hive metastore based on an Amazon RDS for MySQL or Amazon Aurora MySQL instance, use a JDBC connection. Pass the `mariadb-connector-java.jar` with `--jars` in the `spark-submit` parameters of your job run.

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://amzn-s3-demo-bucket/scripts/spark-jdbc.py",
            "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
            --conf
 spark.hadoop.javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
            --conf spark.hadoop.javax.jdo.option.ConnectionUserName=<connection-user-
name>
            --conf spark.hadoop.javax.jdo.option.ConnectionPassword=<connection-
password>
            --conf spark.hadoop.javax.jdo.option.ConnectionURL=<JDBC-Connection-
string>
            --conf spark.driver.cores=2
            --conf spark.executor.memory=10G
            --conf spark.driver.memory=6G
            --conf spark.executor.cores=4"
        }
    }' \
    --configuration-overrides '{
        "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
        }
    }
}'
```

The following code example is a Spark entrypoint script that interacts with a Hive metastore on Amazon RDS.

```
from os.path import expanduser, join, abspath
```

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')
spark = SparkSession \
    .builder \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("SHOW DATABASES").show()
spark.sql("CREATE EXTERNAL TABLE `sampledb`.`sparknyctaxi`(`dispatching_base_num`
 string, `pickup_datetime` string, `dropoff_datetime` string, `pulocationid` bigint,
 `dolocationid` bigint, `sr_flag` bigint) STORED AS PARQUET LOCATION 's3://<s3 prefix>/
nyctaxi_parquet/'")
spark.sql("SELECT count(*) FROM sampledb.sparknyctaxi").show()
spark.stop()
```

Using the thrift service

You can configure your EMR Serverless Hive application to connect to a Hive metastore based on an Amazon RDS for MySQL or Amazon Aurora MySQL instance. To do this, run a thrift server on the master node of an existing Amazon EMR cluster. This option is ideal if you already have an Amazon EMR cluster with a thrift server that you want to use to simplify your EMR Serverless job configurations.

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://amzn-s3-demo-bucket/thriftscript.py",
            "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
            --conf spark.driver.cores=2
            --conf spark.executor.memory=10G
            --conf spark.driver.memory=6G
            --conf spark.executor.cores=4"
        }
    }' \
    --configuration-overrides '{
        "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
```

```
        }
    }
}'
```

The following code example is an entrypoint script (`thriftscript.py`) that uses thrift protocol to connect to a Hive metastore. Note that the `hive.metastore.uris` property needs to be set to read from an external Hive metastore.

```
from os.path import expanduser, join, abspath
from pyspark.sql import SparkSession
from pyspark.sql import Row
# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')
spark = SparkSession \
    .builder \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .config("hive.metastore.uris","thrift://thrift-server-host:thift-server-port") \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("SHOW DATABASES").show()
spark.sql("CREATE EXTERNAL TABLE sampledb.`sparknyctaxi`( `dispatching_base_num`
 string, `pickup_datetime` string, `dropoff_datetime` string, `pulocationid` bigint,
 `dolocationid` bigint, `sr_flag` bigint) STORED AS PARQUET LOCATION 's3://<s3 prefix>/
nyctaxi_parquet/'")
spark.sql("SELECT * FROM sampledb.sparknyctaxi").show()
spark.stop()
```

Configure Hive options

Using JDBC

If you want to specify an external Hive database location on either an Amazon RDS MySQL or Amazon Aurora instance, you can override the default metastore configuration.

> **Note**
>
> In Hive, you can perform multiple writes to metastore tables at the same time. If you share metastore information between two jobs, make sure that you don't write to the same metastore table simultaneously unless you write to different partitions of the same metastore table.

Set the following configurations in the `hive-site` classification to activate the external Hive metastore.

```
{
    "classification": "hive-site",
    "properties": {
        "hive.metastore.client.factory.class":
 "org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClientFactory",
        "javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver",
        "javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-port/db-name",
        "javax.jdo.option.ConnectionUserName": "username",
        "javax.jdo.option.ConnectionPassword": "password"
    }
}
```

Using a thrift server

You can configure your EMR Serverless Hive application to connect to a Hive metastore based on an Amazon RDS for MySQL or Amazon Aurora MySQLinstance. To do this, run a thrift server on the main node of an existing Amazon EMR cluster. This option is ideal if you already have an Amazon EMR cluster that runs a thrift server and you want to use your EMR Serverless job configurations.

Set the following configurations in the `hive-site` classification so that EMR Serverless can access the remote thrift metastore. Note that you must set the `hive.metastore.uris` property to read from an external Hive metastore.

```
{
    "classification": "hive-site",
    "properties": {
        "hive.metastore.client.factory.class":
 "org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClientFactory",
        "hive.metastore.uris": "thrift://thrift-server-host:thirft-server-port"
    }
}
```

Working with AWS Glue multi-catalog hierarchy on EMR Serverless

You can configure your EMR Serverless applications to work with the AWS Glue multi-catalog hierarchy. The following example shows how to use EMR-S Spark with the AWS Glue multi-catalog hierarchy.

To learn more about multi-catalog hierarchy, see Working with a multi-catalog hierarchy in AWS Glue Data Catalog with Spark on Amazon EMR.

Using Redshift Managed Storage (RMS) with Iceberg and AWS Glue Data Catalog

The following shows how to configure Spark for integration with an AWS Glue Data Catalog with Iceberg:

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://amzn-s3-demo-bucket/myscript.py",
            "sparkSubmitParameters": "--conf spark.sql.catalog.nfgac_rms =
 org.apache.iceberg.spark.SparkCatalog
             --conf spark.sql.catalog.rms.type=glue
             --conf spark.sql.catalog.rms.glue.id=Glue RMS catalog ID
             --conf spark.sql.defaultCatalog=rms
             --conf
 spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
        }
    }'
```

A sample query from a table in the catalog, following integration:

```
SELECT * FROM my_rms_schema.my_table
```

Using Redshift Managed Storage (RMS) with Iceberg REST API and AWS Glue Data Catalog

The following shows how to configure Spark to work with Iceberg REST catalog:

```
aws emr-serverless start-job-run \
--application-id application-id \
--execution-role-arn job-role-arn \
--job-driver '{
"sparkSubmit": {
"entryPoint": "s3://amzn-s3-demo-bucket/myscript.py",
    "sparkSubmitParameters": "
    --conf spark.sql.catalog.rms=org.apache.iceberg.spark.SparkCatalog
```

```
    --conf spark.sql.catalog.rms.type=rest
    --conf spark.sql.catalog.rms.warehouse=Glue RMS catalog ID
    --conf spark.sql.catalog.rms.uri=Glue endpoint URI/iceberg
    --conf spark.sql.catalog.rms.rest.sigv4-enabled=true
    --conf spark.sql.catalog.rms.rest.signing-name=glue
    --conf
 spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
    }
  }'
```

A sample query from a table in the catalog:

```
SELECT * FROM my_rms_schema.my_table
```

Considerations when using an external metastore

- You can configure databases that are compatible with MariaDB JDBC as your metastore. Examples of these databases are RDS for MariaDB, MySQL, and Amazon Aurora.
- Metastores aren't auto-initialized. If your metastore isn't initialized with a schema for your Hive version, use the Hive Schema Tool.
- EMR Serverless doesn't support Kerberos authentication. You can't use a thrift metastore server with Kerberos authentication with EMR Serverless Spark or Hive jobs.
- You must configure VPC access to use the multi-catalog hierarchy.

Accessing S3 data in another AWS account from EMR Serverless

You can run Amazon EMR Serverless jobs from one AWS account and configure them to access data in Amazon S3 buckets that belong to another AWS account. This page describes how to configure cross-account access to S3 from EMR Serverless.

Jobs that run on EMR Serverless can use an S3 bucket policy or an assumed role to access data in Amazon S3 from a different AWS account.

Prerequisites

To set up cross-account access for Amazon EMR Serverless, you must complete tasks while signed in to two AWS accounts:

- **`AccountA`** – This is the AWS account where you have created an Amazon EMR Serverless application. Before you set up cross-account access, you must have the following ready in this account:
  - An Amazon EMR Serverless application where you want to run jobs.
  - A job execution role that has the required permissions to run jobs in the application. For more information, see Job runtime roles for Amazon EMR Serverless.
- **`AccountB`** – This is the AWS account that contains the S3 bucket that you want your Amazon EMR Serverless jobs to access.

Use an S3 bucket policy to access cross-account S3 data

To access the S3 bucket in account B from account A, attach the following policy to the S3 bucket in account B.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "Example permissions 1",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::AccountA:root"
         },
         "Action": [
            "s3:ListBucket"
         ],
         "Resource": [
            "arn:aws:s3:::bucket_name_in_AccountB"
         ]
      },
      {
         "Sid": "Example permissions 2",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::AccountA:root"
         },
         "Action": [
            "s3:PutObject",
            "s3:GetObject",
            "s3:DeleteObject"
```

```
            ],
            "Resource": [
               "arn:aws:s3:::bucket_name_in_AccountB/*"
            ]
         }
   ]
}
```

For more information about S3 cross-account access with S3 bucket policies, see Example 2: Bucket owner granting cross-account bucket permissions in the *Amazon Simple Storage Service User Guide*.

Use an assumed role to access cross-account S3 data

Another way to set up cross-account access for Amazon EMR Serverless is with the `AssumeRole` action from the AWS Security Token Service (AWS STS). AWS STS is a global web service that lets you request temporary, limited-privilege credentials for users. You can make API calls to EMR Serverless and Amazon S3 with the temporary security credentials that you create with `AssumeRole`.

The following steps illustrate how to use an assumed role to access cross-account S3 data from EMR Serverless:

1. Create an Amazon S3 bucket, *`cross-account-bucket`*, in `AccountB`. For more information, see Creating a bucket in the *Amazon Simple Storage Service User Guide*. If you want to have cross-account access to DynamoDB, you can also create a DynamoDB table in `AccountB`. For more information, see Creating a DynamoDB table in the *Amazon DynamoDB Developer Guide*.
2. Create a `Cross-Account-Role-B` IAM role in `AccountB` that can access the *`cross-account-bucket`*.
   a. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
   b. Choose **Roles** and create a new role: `Cross-Account-Role-B`. For more information about how to create IAM roles, see Creating IAM roles in the IAM User Guide.
   c. Create an IAM policy that specifies the permissions for `Cross-Account-Role-B` to access the *`cross-account-bucket`* S3 bucket, as the following policy statement demonstrates. Then attach the IAM policy to `Cross-Account-Role-B`. For more information, see Creating IAM policies in the *IAM User Guide*.

```
{
```

```
"Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:*",
            "Resource": [
                "arn:aws:s3:::cross-account-bucket",
                "arn:aws:s3:::cross-account-bucket/*"
            ]
        }
    ]
}
```

If you require DynamoDB access, create an IAM policy that specifies permissions to access the cross-account DynamoDB table. Then attach the IAM policy to `Cross-Account-Role-B`. For more information, see Amazon DynamoDB: Allows access to a specific table in the *IAM User Guide*.

The following is a policy to allow access to the DynamoDB table `CrossAccountTable`.

```
{
"Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:MyRegion:AccountB:table/CrossAccountTable"
        }
    ]
}
```

3. Edit the trust relationship for the `Cross-Account-Role-B` role.
   a. To configure the trust relationship for the role, choose the **Trust Relationships** tab in the IAM console for the role `Cross-Account-Role-B` that you created in Step 2.
   b. Select **Edit Trust Relationship**.
   c. Add the following policy document. This allows `Job-Execution-Role-A` in `AccountA` to assume the `Cross-Account-Role-B` role.

```
{
"Version": "2012-10-17",
```

```
  "Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::AccountA:role/Job-Execution-Role-A"
        },
        "Action": "sts:AssumeRole"
    }
  ]
}
```

4. Grant `Job-Execution-Role-A` in `AccountA` the AWS STS `AssumeRole` permission to assume `Cross-Account-Role-B`.

   a. In the IAM console for AWS account `AccountA`, select `Job-Execution-Role-A`.

   b. Add the following policy statement to the `Job-Execution-Role-A` to allow the `AssumeRole` action on the `Cross-Account-Role-B` role.

```
{
"Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Resource": "arn:aws:iam::AccountB:role/Cross-Account-Role-B"
        }
    ]
}
```

Assumed role examples

You can use a single assumed role to access all S3 resources in an account, or with Amazon EMR 6.11 and higher, you can configure multiple IAM roles to assume when you access different cross-account S3 buckets.

Topics

- Access S3 resources with one assumed role
- Access S3 resources with multiple assumed roles

Access S3 resources with one assumed role

> **Note**
>
> When you configure a job to use a single assumed role, all S3 resources throughout the job use that role, including the `entryPoint` script.

If you want to use a single assumed role to access all S3 resources in account B, specify the following configurations:

1. Specify EMRFS configuration `fs.s3.customAWSCredentialsProvider` to `spark.hadoop.fs.s3.customAWSCredentialsProvider=com.amazonaws.emr.AssumeRoleAW`
2. For Spark, use `spark.emr-serverless.driverEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` and `spark.executorEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` to specify the environment variables on driver and executors.
3. For Hive, use `hive.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN`, `tez.am.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN`, and `tez.task.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` to specify the environment variables on Hive driver, Tez application master, and Tez task containers.

The following examples show how to use an assumed role to start an EMR Serverless job run with cross-account access.

Spark

The following example shows how to use an assumed role to start an EMR Serverless Spark job run with cross-account access to S3.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "entrypoint_location",
            "entryPointArguments": [":argument_1:", ":argument_2:"],
```

```
            "sparkSubmitParameters": "--conf spark.executor.cores=4 --conf
 spark.executor.memory=20g --conf spark.driver.cores=4 --conf spark.driver.memory=8g
 --conf spark.executor.instances=1"
        }
    }' \
     --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "spark-defaults",
            "properties": {
                "spark.hadoop.fs.s3.customAWSCredentialsProvider":
 "spark.hadoop.fs.s3.customAWSCredentialsProvider=com.amazonaws.emr.AssumeRoleAWSCredentials
                "spark.emr-serverless.driverEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
 "arn:aws:iam::AccountB:role/Cross-Account-Role-B",
                "spark.executorEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
 "arn:aws:iam::AccountB:role/Cross-Account-Role-B"
            }
        }]
    }'
```

Hive

The following example shows how to use an assumed role to start an EMR Serverless Hive job run with cross-account access to S3.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "hive": {
            "query": "query_location",
            "parameters": "hive_parameters"
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "hive-site",
            "properties": {
                "fs.s3.customAWSCredentialsProvider":
 "com.amazonaws.emr.serverless.credentialsprovider.AssumeRoleAWSCredentialsProvider",
                "hive.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
 "arn:aws:iam::AccountB:role/Cross-Account-Role-B",
                "tez.am.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
 "arn:aws:iam::AccountB:role/Cross-Account-Role-B",
```

```
                "tez.task.emr-
serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
 "arn:aws:iam::AccountB:role/Cross-Account-Role-B"
            }
        }]
    }'
```

Access S3 resources with multiple assumed roles

With EMR Serverless releases 6.11.0 and higher, you can configure multiple IAM roles to assume when you access different cross-account buckets. If you want to access different S3 resources with different assumed roles in account B, use following configurations when you start the job run:

1. Specify EMRFS configuration `fs.s3.customAWSCredentialsProvider` to `com.amazonaws.emr.serverless.credentialsprovider.BucketLevelAssumeRoleCredenti`
2. Specify EMRFS configuration `fs.s3.bucketLevelAssumeRoleMapping` to define the mapping from S3 bucket name to the IAM role in account B to assume. The value should be in format of `bucket1->role1;bucket2->role2`.

For example, you can use `arn:aws:iam::AccountB:role/Cross-Account-Role-B-1` to access bucket `bucket1`, and use `arn:aws:iam::AccountB:role/Cross-Account-Role-B-2` to access bucket `bucket2`. The following examples show how to start an EMR Serverless job run with cross-account access through multiple assumed roles.

Spark

The following example shows how to use multiple assumed roles to create an EMR Serverless Spark job run.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "entrypoint_location",
            "entryPointArguments": [":argument_1:", ":argument_2:"],
            "sparkSubmitParameters": "--conf spark.executor.cores=4 --conf
 spark.executor.memory=20g --conf spark.driver.cores=4 --conf spark.driver.memory=8g
 --conf spark.executor.instances=1"
```

```
        }
    }' \
     --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "spark-defaults",
            "properties": {
                "spark.hadoop.fs.s3.customAWSCredentialsProvider":
 "com.amazonaws.emr.serverless.credentialsprovider.BucketLevelAssumeRoleCredentialsProvider"
                "spark.hadoop.fs.s3.bucketLevelAssumeRoleMapping":
 "bucket1->arn:aws:iam::AccountB:role/Cross-Account-Role-B-1;bucket2-
>arn:aws:iam::AccountB:role/Cross-Account-Role-B-2"
            }
        }]
    }'
```

Hive

The following examples show how to use multiple assumed roles to create an EMR Serverless Hive job run.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "hive": {
            "query": "query_location",
            "parameters": "hive_parameters"
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [{
            "classification": "hive-site",
            "properties": {
                "fs.s3.customAWSCredentialsProvider":
 "com.amazonaws.emr.serverless.credentialsprovider.AssumeRoleAWSCredentialsProvider",
                "fs.s3.bucketLevelAssumeRoleMapping": "bucket1-
>arn:aws:iam::AccountB:role/Cross-Account-Role-B-1;bucket2-
>arn:aws:iam::AccountB:role/Cross-Account-Role-B-2"
            }
        }]
    }'
```

Troubleshooting errors in EMR Serverless

Use the following information to help diagnose and fix common issues you might encounter when working with Amazon EMR Serverless.

Topics

Error: Job failed as account has reached the service limit on the maximum vCPU it can use concurrently.

This error indicates that EMR Serverless couldn't submit the job as the account has exceeded the maximum capacity. Increase the maximum capacity for the account. Check your service limits at EMR Serverless service quotas.

Error: Job failed as application has exceeded maximumCapacity settings.

This error indicates that EMR Serverless couldn't submit the job as the application has exceeded the configured maximum capacity. Increase the maximum capacity for the application.

Error: Job failed due to Worker could not be allocated as the application has exceeded maximumCapacity.

This error indicates that the job couldn't complete. Workers couldn't be allocated because the application has exceeded maximumCapacity settings.

Error: S3 access is denied. Please check S3 access permissions of the job runtime role on the required S3 resources.

This error indicates that your job doesn't have access to your S3 resources. Verify that the job runtime role has permission to access the S3 resources that the job needs to use. To learn more about runtime roles, see Job runtime roles for Amazon EMR Serverless.

Error: ModuleNotFoundError: No module named <module>. Please refer to the user guide on how to use python libraries with EMR Serverless.

This error indicates that a Python module wasn't available for the Spark job. Check that the dependent Python libraries are available to the job. For more information on how to package Python libraries, see Using Python libraries with EMR Serverless.

Error: Could not assume execution role <role name> because it does not exist or is not set up with the required trust relationship.

This error indicates that the job runtime role that you specified for the job doesn't exist, or that the role doesn't have a trust relationship for EMR Serverless permissions. To verify that the IAM role exists and validate that you have set up the role's trust policy properly, see the instructions in Job runtime roles for Amazon EMR Serverless.

Run interactive workloads with EMR Serverless through EMR Studio

With EMR Serverless interactive applications, you can run interactive workloads for Spark with EMR Serverless using notebooks that are hosted in EMR Studio.

Overview

An *interactive application* is an EMR Serverless application that has interactive capabilities enabled. With Amazon EMR Serverless interactive applications, you can execute interactive workloads with Jupyter notebooks that are managed in Amazon EMR Studio. This helps data engineers, data scientists, and data analysts use EMR Studio to run interactive analytics with datasets in data stores such as Amazon S3 and Amazon DynamoDB.

Use cases for interactive applications in EMR Serverless include the following:

- Data engineers use the IDE experience in EMR Studio to create an ETL script. The script ingests data from on-premises, transforms the data for analysis, and stores the data in Amazon S3.
- Data scientists use notebooks to explore datasets and train machine-learning (ML) models to detect anomalies in the datasets.
- Data analysts explore datasets and create scripts that generate daily reports to update applications like business dashboards.

Prerequisites

To use interactive workloads with EMR Serverless, you must meet the following requirements:

- EMR Serverless interactive applications are supported with Amazon EMR 6.14.0 and higher.
- To access your interactive application, execute the workloads that you submit, and run interactive notebooks from EMR Studio, you need specific permissions and roles. For more information, see Required permissions for interactive workloads.

Required permissions for interactive workloads

In addition to the basic permissions that are required to access EMR Serverless, you must configure additional permissions for your IAM identity or role:

To access your interactive application

Set up user and Workspace permissions for EMR Studio. For more information, see Configure EMR Studio user permissions in the *Amazon EMR Management Guide*.

To execute the workloads that you submit with EMR Serverless

Set up a job runtime role. For more information, see Create a job runtime role.

To run the interactive notebooks from EMR Studio

Add the following additional permissions to the IAM policy for the Studio users:

- **`emr-serverless:AccessInteractiveEndpoints`** - Grants permission to access and connect to the interactive application that you specify as `Resource`. This permission is required to attach to an EMR Serverless application from an EMR Studio Workspace.
- **`iam:PassRole`** - Grants permission to access the IAM execution role that you plan to use when you attach to an application. The appropriate `PassRole`permission is required to attach to an EMR Serverless application from an EMR Studio Workspace.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRServerlessInteractiveAccess",
            "Effect": "Allow",
            "Action": "emr-serverless:AccessInteractiveEndpoints",
            "Resource": "arn:aws:emr-serverless:Region:account:/applications/*"
        },
        {
            "Sid": "EMRServerlessRuntimeRoleAccess",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "interactive-execution-role-ARN",
            "Condition": {
                "StringLike": {
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
            }
```

```
        }
    ]
}
```

Configuring interactive applications

Use the following high-level steps to create an EMR Serverless application with interactive capabilities from Amazon EMR Studio in the AWS Management Console.

1. Follow the steps in Getting started with Amazon EMR Serverless to create an application.
2. Then, launch a workspace from EMR Studio and attach to an EMR Serverless application as a compute option. For more information, see the **Interactive workload** tab in Step 2 of the EMR Serverless Getting Started documentation.

When you attach an application to a Studio Workspace, the application start triggers automatically if it's not already running. You can also pre-start the application and keep it ready before you attach it to the Workspace.

Considerations with interactive applications

- EMR Serverless interactive applications are supported with Amazon EMR 6.14.0 and higher.
- EMR Studio is the only client that is integrated with EMR Serverless interactive applications. The following EMR Studio capabilities aren't supported with EMR Serverless interactive applications: Workspace collaboration, SQL Explorer, and programmatic execution of notebooks.
- Interactive applications are only supported for Spark engine.
- Interactive applications support Python 3, PySpark and Spark Scala kernels.
- You can run up to 25 concurrent notebooks on a single interactive application.
- There isn't an endpoint or API interface that supports self-hosted Jupyter notebooks with interactive applications.
- For an optimized startup experience, we recommend that you configure pre-initialized capacity for drivers and executors, and that you pre-start your application. When you pre-start the application, you ensure that it's ready when you want to attach it to your Workspace.

```
aws emr-serverless start-application \
--application-id your-application-id
```

- By default, `autoStopConfig` is enabled for applications. This shuts down the application after 30 minutes of idle time. You can change this configuration as part of your `create-application` or `update-application` request.
- When using an interactive application, we recommend that you configure a pre-intialized capacity of kernels, drivers, and executors to run your notebooks. Each Spark interactive session requires one kernel and one driver, so EMR Serverless maintains a pre-initialized kernel worker for every pre-initialized driver. By default, EMR Serverless maintains a pre-initialized capacity of one kernel worker throughout the entire application even if you don't specify any pre-initialized capacity for drivers. Each kernel worker uses 4 vCPU and 16 GB of memory. For current pricing information, see the Amazon EMR Pricing page.
- You must have sufficient vCPU service quota in your AWS account to run interactive workloads. If you don't run Lake Formation-enabled workloads, we recommend at least 24 vCPU. If you do, we recommend at least 28 vCPU.
- EMR Serverless automatically terminates the kernels from the notebooks if they have been idle for more than 60 minutes. EMR Serverless calculates the kernel idle time from the last activity completed during the notebook session. You can't currently modify the kernel idle timeout setting.
- To enable Lake Formation with interactive workloads, set the configuration `spark.emr-serverless.lakeformation.enabled` to `true` under the `spark-defaults` classification in the `runtime-configuration` object when you create an EMR Serverless application. To learn more about enabling Lake Formation in EMR Serverless, see Enabling Lake Formation in Amazon EMR.

Run interactive workloads with EMR Serverless through an Apache Livy endpoint

With Amazon EMR releases 6.14.0 and higher, you can create and enable an Apache Livy endpoint while creating an EMR Serverless application and run interactive workloads through your self-hosted notebooks or with a custom client. An Apache Livy endpoint offers the following benefits:

- You can securely connect to an Apache Livy endpoint through Jupyter notebooks and manage Apache Spark workloads with Apache Livy's REST interface.
- Use the Apache Livy REST API operations for interactive web applications that use data from Apache Spark workloads.

Prerequisites

To use an Apache Livy endpoint with EMR Serverless, you must meet the following requirements:

- Complete the steps in Getting started with Amazon EMR Serverless.
- To run interactive workloads through Apache Livy endpoints, you need certain permissions and roles. For more information, see Required permissions for interactive workloads.

Required permissions

In addition to the required permissions to access EMR Serverless, you must also add the following permissions to your IAM role to access an Apache Livy endpoint and run applications:

- `emr-serverless:AccessLivyEndpoints` – grants permission to access and connect to the Livy-enabled application that you specify as `Resource`. You need this permission to run the REST API operations available from the Apache Livy endpoint.
- `iam:PassRole` – grants permission to access the IAM execution role while creating the Apache Livy session. EMR Serverless will use this role to execute your workloads.
- `emr-serverless:GetDashboardForJobRun` – grants permission to generate the Spark Live UI and driver log links and provides access to the logs as part of the Apache Livy session results.

```
{
    "Version": "2012-10-17",
    "Statement": [{
            "Sid": "EMRServerlessInteractiveAccess",
            "Effect": "Allow",
            "Action": "emr-serverless:AccessLivyEndpoints",
            "Resource": "arn:aws:emr-serverless:<AWS_REGION>:account:/applications/*"
        },
        {
            "Sid": "EMRServerlessRuntimeRoleAccess",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "execution-role-ARN",
            "Condition": {
                "StringLike": {
                    "iam:PassedToService": "emr-serverless.amazonaws.com"
                }
            }
```

```
        },
        {
            "Sid": "EMRServerlessDashboardAccess",
            "Effect": "Allow",
            "Action": "emr-serverless:GetDashboardForJobRun",
            "Resource": "arn:aws:emr-serverless:<AWS_REGION>:account:/applications/*"
        }
    ]
}
```

Getting started

To create an Apache Livy-enabled application and run it, follow these steps.

1. To create an Apache Livy-enabled application, run the following command.

```
 aws emr-serverless create-application \
--name my-application-name \
--type 'application-type' \
 --release-label <Amazon EMR-release-version>
--interactive-configuration '{"livyEndpointEnabled": true}'
```

2. After EMR Serverless creates your application, start the application to make the Apache Livy endpoint available.

```
 aws emr-serverless start-application \
 --application-id application-id
```

   Use the following command to check whether the status of your application. Once the status becomes STARTED, you can access the Apache Livy endpoint.

```
aws emr-serverless get-application \
--region <AWS_REGION> --application-id >application_id>
```

3. Use the following URL to access the endpoint:

```
https://_<application-id>_.livy.emr-serverless-
services._<AWS_REGION>_.amazonaws.com
```

Once the endpoint is ready, you can submit workloads based on your use case. You must sign every request to the endpoint with the SIGv4 protocol and pass in an authorization header. You can use the following methods to run workloads:

- HTTP client – you must submit your Apache Livy endpoint API operations with a custom HTTP client.
- Sparkmagic kernel – you must locally run the sparkmagic kernel and submit interactive queries with Jupyter notebooks.

HTTP clients

To create an Apache Livy session, you must submit `emr-serverless.session.executionRoleArn` in the `conf` parameter of your request body. The following example is a sample `POST /sessions` request.

```
{
    "kind": "pyspark",
    "heartbeatTimeoutInSecond": 60,
    "conf": {
        "emr-serverless.session.executionRoleArn": "<executionRoleArn>"
    }
}
```

The following table describes all of the available Apache Livy API operations.

API operation	Description
GET /sessions	Returns a list of all of the active interactive sessions.
POST /sessions	Creates a new interactive session via spark or pyspark.
GET /sessions/<*sessionId* >	Returns the session information.
GET /sessions/<*sessionId* >/state	Returns the state of session.
DELETE /sessions/<*sessionId* >	Stops and deletes the session.

API operation	Description
GET /sessions/<*sessionId* >/statements	Returns all the statements in a session.
POST /sessions/<*sessionId* >/statements	Runs a statement in a session.
GET /sessions/<*sessionId* >/statements/<*statementId* >	Returns the details of the specified statement in a session.
POST /sessions/<*sessionId* >/statements/<*statementId* >/cancel	Cancels the specified statement in this session.

Sending requests to the Apache Livy endpoint

You can also send requests directly to the Apache Livy endpoint from an HTTP client. Doing so lets you remotely run code for your use cases outside of a notebook.

Before you can start sending requests to the endpoint, make sure that you've installed the following libraries:

```
pip3 install botocore awscrt requests
```

The following is a sample Python script to send HTTP requests directly to an endpoint:

```
from botocore import crt
import requests
from botocore.awsrequest import AWSRequest
from botocore.credentials import Credentials
import botocore.session
import json, pprint, textwrap

endpoint = 'https://<application_id>.livy.emr-serverless-
services-<AWS_REGION>.amazonaws.com'
headers = {'Content-Type': 'application/json'}

session = botocore.session.Session()
signer = crt.auth.CrtS3SigV4Auth(session.get_credentials(), 'emr-serverless',
 '<AWS_REGION>')


### Create session request
```

```
data = {'kind': 'pyspark', 'heartbeatTimeoutInSecond': 60, 'conf': { 'emr-
serverless.session.executionRoleArn': 'arn:aws:iam::123456789012:role/role1'}}

request = AWSRequest(method='POST', url=endpoint + "/sessions", data=json.dumps(data),
 headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r = requests.post(prepped.url, headers=prepped.headers, data=json.dumps(data))

pprint.pprint(r.json())


### List Sessions Request

request = AWSRequest(method='GET', url=endpoint + "/sessions", headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r2 = requests.get(prepped.url, headers=prepped.headers)
pprint.pprint(r2.json())


### Get session state

session_url = endpoint + r.headers['location']

request = AWSRequest(method='GET', url=session_url, headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()
```

```
r3 = requests.get(prepped.url, headers=prepped.headers)

pprint.pprint(r3.json())


### Submit Statement

data = {
      'code': "1 + 1"
}

statements_url = endpoint + r.headers['location'] + "/statements"

request = AWSRequest(method='POST', url=statements_url, data=json.dumps(data),
 headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r4 = requests.post(prepped.url, headers=prepped.headers, data=json.dumps(data))

pprint.pprint(r4.json())

### Check statements results

specific_statement_url = endpoint + r4.headers['location']

request = AWSRequest(method='GET', url=specific_statement_url, headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r5 = requests.get(prepped.url, headers=prepped.headers)

pprint.pprint(r5.json())


### Delete session
```

```
session_url = endpoint + r.headers['location']

request = AWSRequest(method='DELETE', url=session_url, headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r6 = requests.delete(prepped.url, headers=prepped.headers)

pprint.pprint(r6.json())
```

Sparkmagic kernel

Before you install sparkmagic, make sure that you have configured AWS credentials in the instance in which you want to install sparkmagic

1. Install sparkmagic by following the installation steps. Note that you only need to perform the first four steps.
2. The sparkmagic kernel supports custom authenticators, so you can integrate an authenticator with the sparkmagic kernel so that every request is SIGv4 signed.
3. Install the EMR Serverless custom authenticator.

   ```
   pip install emr-serverless-customauth
   ```

4. Now provide the path to the custom authenticator and the Apache Livy endpoint URL in the sparkmagic configuration json file. Use the following command to open the configuration file.

   ```
   vim ~/.sparkmagic/config.json
   ```

   The following is a sample `config.json` file.

   ```
   {
   "kernel_python_credentials" : {
       "username": "",
       "password": "",
   ```

```
    "url": "https://<application-id>.livy.emr-serverless-
services.<AWS_REGION>.amazonaws.com",
    "auth": "Custom_Auth"
  },

  "kernel_scala_credentials" : {
    "username": "",
    "password": "",
    "url": "https://<application-id>.livy.emr-serverless-
services.<AWS_REGION>.amazonaws.com",
    "auth": "Custom_Auth"
  },
  "authenticators": {
    "None": "sparkmagic.auth.customauth.Authenticator",
    "Basic_Access": "sparkmagic.auth.basic.Basic",
    "Custom_Auth":
 "emr_serverless_customauth.customauthenticator.EMRServerlessCustomSigV4Signer"
  },
  "livy_session_startup_timeout_seconds": 600,
  "ignore_ssl_errors": false
}
```

5. Start Jupyter lab. It should use the custom authentication that you set up in the last step.
6. You can then run the following notebook commands and your code to get started.

```
%%info //Returns the information about the current sessions.
```

```
%%configure -f //Configure information specific to a session. We supply
 executionRoleArn in this example. Change it for your use case.
{
    "driverMemory": "4g",
    "conf": {
          "emr-serverless.session.executionRoleArn":
 "arn:aws:iam::123456789012:role/JobExecutionRole"
    }
}
```

```
<your code>//Run your code to start the session
```

Internally, each instruction calls each of the Apache Livy API operations through the configured Apache Livy endpoint URL. You can then write your instructions according to your use case.

Considerations

Consider the following considerations when running interactive workloads through Apache Livy endpoints.

- EMR Serverless maintains session-level isolation using the caller principal. The caller principal that creates the session is the only one that can access that session. For more granular isolation, you can configure a source identity when you assume credentials. In this case, EMR Serverless enforces session-level isolation based on both the caller principal and the source identity. For more information about source identity, see Monitor and control actions taken with assumed roles.
- Apache Livy endpoints are supported with EMR Serverless releases 6.14.0 and higher.
- Apache Livy endpoints are supported only for the Apache Spark engine.
- Apache Livy endpoints support Scala Spark and PySpark.
- By default, `autoStopConfig` is enabled in your applications. This means that applications shut down after 15 minutes of being idle. You can change this configuration as part of your `create-application` or `update-application` request.
- You can run up to 25 concurrent sessions on a single Apache Livy endpoint-enabled application.
- For the best startup experience, we recommend that you configure pre-initialized capacity for drivers and executors.
- You must manually start your application before connecting to the Apache Livy endpoint.
- You must have sufficient vCPU service quota in your AWS account to run interactive workloads with the Apache Livy endpoint. We recommend at least 24 vCPU.
- The default Apache Livy session timeout is 1 hour. If you don't run statements one hour, then Apache Livy deletes the session and releases the driver and executors. You can't change this configuration.
- Only active sessions can interact with an Apache Livy endpoint. Once the session finishes, cancels, or terminates, you can't access it through the Apache Livy endpoint.

Logging and monitoring

Monitoring is an important part of maintaining the reliability, availability, and performance of EMR Serverless applications and jobs. You should collect monitoring data from all of the parts of your EMR Serverless solutions so that you can more easily debug a multipoint failure if one occurs.

Topics

Storing logs

To monitor your job progress on EMR Serverless and troubleshoot job failures, you can choose how EMR Serverless stores and serves application logs. When you submit a job run, you can specify managed storage, Amazon S3, and Amazon CloudWatch as your logging options.

With CloudWatch, you can specify the log types and log locations that you want to use, or accept the default types and locations. For more information on CloudWatch logs, see the section called "Amazon CloudWatch". With managed storage and S3 logging, the following table shows the log locations and UI availability that you can expect if you choose managed storage, Amazon S3 buckets, or both.

Option	Event logs	Container logs	Application UI
Managed storage	Stored in managed storage	Stored in managed storage	Supported
Both managed storage and S3 bucket	Stored in both places	Stored in S3 bucket	Supported

Option	Event logs	Container logs	Application UI
Amazon S3 bucket	Stored in S3 bucket	Stored in S3 bucket	Not supported[1]

[1] We recommend that you keep the **Managed storage** option selected. Otherwise, you can't use the built-in application UIs.

Logging for EMR Serverless with managed storage

By default, EMR Serverless stores application logs securely in Amazon EMR managed storage for a maximum of 30 days.

> **Note**
>
> If you turn off the default option, Amazon EMR can't troubleshoot your jobs on your behalf.

To turn off this option from EMR Studio, deselect the **Allow AWS to retain logs for 30 days** check box in the **Additional settings** section of the **Submit job** page.

To turn off this option from the AWS CLI, use the `managedPersistenceMonitoringConfiguration` configuration when you submit a job run.

```
{
    "monitoringConfiguration": {
        "managedPersistenceMonitoringConfiguration": {
            "enabled": false
        }
    }
}
```

Logging for EMR Serverless with Amazon S3 buckets

Before your jobs can send log data to Amazon S3, you must include the following permissions in the permissions policy for the job runtime role. Replace *amzn-s3-demo-logging-bucket* with the name of your logging bucket.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-logging-bucket/*"
            ]
        }
    ]
}
```

To set up an Amazon S3 bucket to store logs from the AWS CLI, use the `s3MonitoringConfiguration` configuration when you start a job run. To do this, provide the following `--configuration-overrides` in the configuration.

```
{
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-logging-bucket/logs/"
        }
    }
}
```

For batch jobs that don't have retries enabled, EMR Serverless sends the logs to the following path:

```
'/applications/<applicationId>/jobs/<jobId>'
```

EMR Serverless releases 7.1.0 and higher support retry attempts for streaming jobs and batch jobs. If you run a job with retries enabled, EMR Serverless automatically adds an attempt number to the log path prefix, so you can better distinguish and track logs.

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'
```

Logging for EMR Serverless with Amazon CloudWatch

When you submit a job to an EMR Serverless application, you can choose Amazon CloudWatch as an option to store your application logs. This allows you to use CloudWatch log analysis features

such as CloudWatch Logs Insights and Live Tail. You can also stream logs from CloudWatch to other systems such as OpenSearch for further analysis.

EMR Serverless provides real-time logging for driver logs. You can view the logs in real time with the CloudWatch live tail capability, or through CloudWatch CLI tail commands.

By default, CloudWatch logging is disabled for EMR Serverless. To enable it, see the configuration in AWS CLI.

> **Note**
>
> Amazon CloudWatch publishes logs in real time, so it incurs more resources from workers. If you choose a low worker capacity, the impact to your job run time might increase. If you enable CloudWatch logging, we recommend that you choose a greater worker capacity. It's also possible that log publication could throttle if the transactions per second (TPS) rate is too low for `PutLogEvents`. The CloudWatch throttling configuration is global to all services, including EMR Serverless. For more information, see How do I determine throttling in my CloudWatch logs? on *AWS re:post*.

Required permissions for logging with CloudWatch

Before your jobs can send log data to Amazon CloudWatch, you must include the following permissions in the permissions policy for the job runtime role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups"
            ],
            "Resource": [
                "arn:aws:logs:AWS Region:111122223333:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
```

```
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws:logs:AWS Region:111122223333:log-group:my-log-group-name:*"
            ]
        }
    ]
}
```

AWS CLI

To set up Amazon CloudWatch to store logs for EMR Serverless from the AWS CLI, use the `cloudWatchLoggingConfiguration` configuration when you start a job run. To do this, provide the following configuration overrides. Optionally, you can also provide a log group name, log stream prefix name, log types, and an encryption key ARN.

If you don't specify the optional values, then CloudWatch publishes the logs to a default log group `/aws/emr-serverless`, with the default log stream `/applications/applicationId/jobs/jobId/worker-type`.

EMR Serverless releases 7.1.0 and higher support retry attempts for streaming jobs and batch jobs. If you enabled retries for a job, EMR Serverless automatically adds an attempt number to the log path prefix, so you can better distinguish and track logs.

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/worker-type'
```

The following shows the minimum configuration that is required to turn on Amazon CloudWatch logging with the default settings for EMR Serverless:

```
{
    "monitoringConfiguration": {
        "cloudWatchLoggingConfiguration": {
            "enabled": true
         }
     }
}
```

The following example shows all of the required and optional configurations that you can specify when you turn on Amazon CloudWatch logging for EMR Serverless. The supported `logTypes` values are also listed below this example.

```
{
    "monitoringConfiguration": {
        "cloudWatchLoggingConfiguration": {
            "enabled": true, // Required
            "logGroupName": "Example_logGroup", // Optional
            "logStreamNamePrefix": "Example_logStream", // Optional
            "encryptionKeyArn": "key-arn", // Optional
            "logTypes": {
                "SPARK_DRIVER": ["stdout", "stderr"] //List of values
             }
         }
     }
}
```

By default, EMR Serverless publishes only the driver stdout and stderr logs to CloudWatch. If you want other logs, then you can specify a container role and corresponding log types with the `logTypes` field.

The following list shows the supported worker types that you can specify for the `logTypes` configuration:

Spark

- `SPARK_DRIVER : ["STDERR", "STDOUT"]`
- `SPARK_EXECUTOR : ["STDERR", "STDOUT"]`

Hive

- `HIVE_DRIVER : ["STDERR", "STDOUT", "HIVE_LOG", "TEZ_AM"]`
- `TEZ_TASK : ["STDERR", "STDOUT", "SYSTEM_LOGS"]`

Rotating logs

Amazon EMR Serverless can rotate Spark application logs and event logs. Log rotation helps with the issue of long running jobs generating large log files that can take up all of your disk space. Rotating logs helps you save disk storage and reduces the amount of job failures because you have no more space left on your disk.

Log rotation is enabled by default and is available only for Spark jobs.

Spark event logs

> **Note**
>
> Spark event log rotation is available across all Amazon EMR release labels.

Instead of generating a single event log file, EMR Serverless rotates the event log at a regular time interval and removes the older event log files. Rotating logs doesn't affect the logs uploaded to the S3 bucket.

Spark application logs

> **Note**
>
> Spark application log rotation is available across all Amazon EMR release labels.

EMR Serverless also rotates the spark application logs for drivers and executors, such as `stdout` and `stderr` files. You can access the latest log files by choosing the log links in Studio by using the Spark History Server and Live UI links. Log files are the truncated versions of the latest logs. To see the older rotated logs, you must specify an Amazon S3 location when storing logs. See Logging for EMR Serverless with Amazon S3 buckets for more information.

You can find the latest log files at the following location. EMR Serverless refreshes the files every 15 seconds. These files can range from 0 MB to 128 MB.

```
<example-S3-logUri>/applications/<application-id>/jobs/<job-id>/SPARK_DRIVER/stderr.gz
```

The following location contains the older rotated files. Each file is 128 MB.

```
<example-S3-logUri>/applications/<application-id>/jobs/<job-id>/SPARK_DRIVER/archived/
stderr_<index>.gz
```

The same behavior applies to Spark executors as well. This change is only applicable to S3 logging. Log rotation doesn't introduce any changes to log streams uploaded to Amazon CloudWatch.

EMR Serverless releases 7.1.0 and higher support retry attempts for streaming and batch jobs. If you enabled retry attempts with your job, EMR Serverless adds a prefix to the log path for such jobs so you can better track and distinguish the logs from one another. This path contains all rotated logs.

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'.
```

Encrypting logs

Encrypting EMR Serverless logs with managed storage

To encrypt logs in managed storage with your own KMS key, use the `managedPersistenceMonitoringConfiguration` configuration when you submit a job run.

```
{
    "monitoringConfiguration": {
        "managedPersistenceMonitoringConfiguration" : {
            "encryptionKeyArn": "key-arn"
        }
    }
}
```

Encrypting EMR Serverless logs with Amazon S3 buckets

To encrypt logs in your Amazon S3 bucket with your own KMS key, use the `s3MonitoringConfiguration` configuration when you submit a job run.

```
{
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-logging-bucket/logs/",
            "encryptionKeyArn": "key-arn"
        }
    }
}
```

Encrypting EMR Serverless logs with Amazon CloudWatch

To encrypt logs in Amazon CloudWatch with your own KMS key, use the `cloudWatchLoggingConfiguration` configuration when you submit a job run.

```
{
    "monitoringConfiguration": {
        "cloudWatchLoggingConfiguration": {
            "enabled": true,
            "encryptionKeyArn": "key-arn"
         }
     }
}
```

Required permissions for log encryption

In this section

Required user permissions

The user who submits the job or views the logs or the application UIs must have permissions to use the key. You can specify the permissions in either the KMS key policy or the IAM policy for the user, group, or role. If the user who submits the job lacks the KMS key permissions, EMR Serverless rejects the job run submission.

Example key policy

The following key policy provides the permissions to `kms:GenerateDataKey` and `kms:Decrypt`:

```
{
    "Effect": "Allow",
    "Principal":{
       "AWS": "arn:aws:iam::111122223333:user/user-name"
     },
     "Action": [
       "kms:GenerateDataKey",
       "kms:Decrypt"
      ],
     "Resource": "*"
 }
```

Example IAM policy

The following IAM policy provides the permissions to `kms:GenerateDataKey` and `kms:Decrypt`:

```
{
 "Version": "2012-10-17",
  "Statement": {
     "Effect": "Allow",
     "Action": [
           "kms:GenerateDataKey",
           "kms:Decrypt"
      ],
     "Resource": "key-arn"
   }
}
```

To launch the Spark or Tez UI, you must give your users, groups, or roles permissions to access the `emr-serverless:GetDashboardForJobRun` API as follows:

```
{
 "Version": "2012-10-17",
  "Statement": {
     "Effect": "Allow",
     "Action": [
           "emr-serverless:GetDashboardForJobRun"
      ]
   }
}
```

Encryption key permissions for Amazon S3 and managed storage

When you encrypt logs with your own encryption key either in managed storage or in your S3 buckets, you must configure KMS key permissions as follows.

The `emr-serverless.amazonaws.com` principal must have the following permissions in the policy for the KMS key:

```
{
    "Effect": "Allow",
    "Principal":{
       "Service": "emr-serverless.amazonaws.com"
     },
     "Action": [
       "kms:Decrypt",
```

```
        "kms:GenerateDataKey"
       ],
      "Resource": "*"
      "Condition": {
        "StringLike": {
          "aws:SourceArn": "arn:aws:emr-serverless:region:aws-account-id:/
applications/application-id"
        }
      }
 }
```

As a security best practice, we recommend that you add an `aws:SourceArn` condition key to the KMS key policy. The IAM global condition key `aws:SourceArn` helps ensure that EMR Serverless uses the KMS key only for an application ARN.

The job runtime role must have the following permissions in its IAM policy:

```
{
 "Version": "2012-10-17",
  "Statement": {
     "Effect": "Allow",
     "Action": [
           "kms:GenerateDataKey",
           "kms:Decrypt"
       ],
     "Resource": "key-arn"
   }
}
```

Encryption key permissions for Amazon CloudWatch

To associate the KMS key ARN to your log group, use the following IAM policy for the job runtime role.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "logs:AssociateKmsKey"
        ],
        "Resource": [
```

```
            "arn:aws:logs:AWS Region:111122223333:log-group:my-log-group-name:*"
        ]
    }
}
```

Configure the KMS key policy to grant KMS permissions to Amazon CloudWatch:

```
{
    "Version": "2012-10-17",
    "Id": "key-default-1",
    "Statement":
     {
        "Effect": "Allow",
        "Principal": {
            "Service": "logs.AWS Region.amazonaws.com"
        },
        "Action": [
            "kms:Decrypt",
            "kms:GenerateDataKey",
        ],
        "Resource": "*",
        "Condition": {
            "ArnLike": {
                "kms:EncryptionContext:aws:logs:arn": "arn:aws:logs:AWS
 Region:111122223333:*"
            }
        }
     }
}
```

Configure Apache Log4j2 properties for Amazon EMR Serverless

This page describes how to configure custom Apache Log4j 2.x properties for EMR Serverless jobs at `StartJobRun`. If you want to configure Log4j classifications at the application level, see Default application configuration for EMR Serverless.

Configure Spark Log4j2 properties for Amazon EMR Serverless

With Amazon EMR releases 6.8.0 and higher, you can customize Apache Log4j 2.x properties to specify fine-grained log configurations. This simplifies troubleshooting of your Spark jobs on

EMR Serverless. To configure these properties, use the `spark-driver-log4j2` and `spark-executor-log4j2` classifications.

Topics

Log4j2 classifications for Spark

To customize the Spark log configurations, use the following classifications with `applicationConfiguration`. To configure the Log4j 2.x properties, use the following `properties`.

`spark-driver-log4j2`

This classification sets the values in the `log4j2.properties` file for the driver.

`spark-executor-log4j2`

This classification sets the values in the `log4j2.properties` file for the executor.

Log4j2 configuration example for Spark

The following example shows how to submit a Spark job with `applicationConfiguration` to customize Log4j2 configurations for the Spark driver and executor.

To configure Log4j classifications at the application level instead of when you submit the job, see Default application configuration for EMR Serverless.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
            "entryPointArguments": ["1"],
```

```
            "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi --conf
 spark.executor.cores=4 --conf spark.executor.memory=20g --conf spark.driver.cores=4 --
conf spark.driver.memory=8g --conf spark.executor.instances=1"
        }
    }'
    --configuration-overrides '{
        "applicationConfiguration": [
             {
                "classification": "spark-driver-log4j2",
                "properties": {
                    "rootLogger.level":"error", // will only display Spark error logs
                    "logger.IdentifierForClass.name": "classpath for setting logger",
                    "logger.IdentifierForClass.level": "info"

                }
            },
            {
                "classification": "spark-executor-log4j2",
                "properties": {
                    "rootLogger.level":"error", // will only display Spark error logs
                    "logger.IdentifierForClass.name": "classpath for setting logger",
                    "logger.IdentifierForClass.level": "info"
                }
            }
       ]
    }'
```

Log4j2 in sample Spark jobs

The following code samples demonstrate how to create a Spark application while you initialize a custom Log4j2 configuration for the application.

Python

Example - Using Log4j2 for a Spark job with Python

```
import os
import sys

from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

app_name = "PySparkApp"
```

```
if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .appName(app_name)\
        .getOrCreate()

    spark.sparkContext._conf.getAll()
    sc = spark.sparkContext
    log4jLogger = sc._jvm.org.apache.log4j
    LOGGER = log4jLogger.LogManager.getLogger(app_name)

    LOGGER.info("pyspark script logger info")
    LOGGER.warn("pyspark script logger warn")
    LOGGER.error("pyspark script logger error")

    // your code here

    spark.stop()
```

To customize Log4j2 for the driver when you execute a Spark job, you can use the following configuration:

```
{
   "classification": "spark-driver-log4j2",
      "properties": {
          "rootLogger.level":"error", // only display Spark error logs
          "logger.PySparkApp.level": "info",
          "logger.PySparkApp.name": "PySparkApp"
      }
}
```

Scala

Example - Using Log4j2 for a Spark job with Scala

```
import org.apache.log4j.Logger
import org.apache.spark.sql.SparkSession

object ExampleClass {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
    .builder
```

```
    .appName(this.getClass.getName)
    .getOrCreate()

    val logger = Logger.getLogger(this.getClass);
    logger.info("script logging info logs")
    logger.warn("script logging warn logs")
    logger.error("script logging error logs")

// your code here
    spark.stop()
  }
}
```

To customize Log4j2 for the driver when you execute a Spark job, you can use the following configuration:

```
{
   "classification": "spark-driver-log4j2",
      "properties": {
          "rootLogger.level":"error", // only display Spark error logs
          "logger.ExampleClass.level": "info",
          "logger.ExampleClass.name": "ExampleClass"
      }
}
```

Log4j2 considerations for Spark

The following Log4j2.x properties are not configurable for Spark processes:

- `rootLogger.appenderRef.stdout.ref`
- `appender.console.type`
- `appender.console.name`
- `appender.console.target`
- `appender.console.layout.type`
- `appender.console.layout.pattern`

For detailed information about the Log4j2.x properties that you can configure, see the `log4j2.properties.template` file on GitHub.

Monitoring EMR Serverless

This section covers the ways that you can monitor your Amazon EMR Serverless applications and jobs.

Topics

Monitoring EMR Serverless applications and jobs

With Amazon CloudWatch metrics for EMR Serverless, you can receive 1-minute CloudWatch metrics and access CloudWatch dashboards to view near-real-time operations and performance of your EMR Serverless applications.

EMR Serverless sends metrics to CloudWatch every minute. EMR Serverless emits these metrics at the application level as well as the job, worker-type, and capacity-allocation-type levels.

To get started, use the EMR Serverless CloudWatch dashboard template provided in the EMR Serverless GitHub repository and deploy it.

> **Note**
>
> EMR Serverless interactive workloads have only application-level monitoring enabled, and have a new worker type dimension, `Spark_Kernel`. To monitor and debug your interactive workloads, you can view the logs and Apache Spark UI from within your EMR Studio Workspace.

The table below describes the EMR Serverless dimensions available within the `AWS/EMRServerless` namespace.

Dimensions for EMR Serverless metrics

Dimension	Description	
`ApplicationId`	Filters for all metrics of an EMR Serverless application.	
`JobId`	Filters for all metrics of an EMR Serverless job run.	
`WorkerType`	Filters for all metrics of a given worker type. For example, you can filter for `SPARK_DRIVER` and `SPARK_EXECUTORS` for Spark jobs.	
`CapacityAllocationType`	Filters for all metrics of a given capacity allocation type. For example, you can filter for `PreInitCapacity` for pre-initialized capacity and `OnDemandCapacity` for everything else.	

Application-level monitoring

You can monitor capacity usage at the EMR Serverless application level with Amazon CloudWatch metrics. You can also set up a single view to monitor application capacity usage in a CloudWatch dashboard.

EMR Serverless application metrics

Metric	Description	Primary dimension	Secondary dimension
`CPUAllocated`	The total numbers of vCPUs allocated.	`ApplicationId`	`ApplicationId ,WorkerTyp`

Metric	Description	Primary dimension	Secondary dimension
			e ,CapacityA llocationType
IdleWorkerCount	The number of total workers idle.	ApplicationId	Applicati onId ,WorkerTyp e ,CapacityA llocationType
MaxCPUAllowed	The maximum CPU allowed for the application.	ApplicationId	N/A
MaxMemory Allowed	The maximum memory in GB allowed for the application.	ApplicationId	N/A
MaxStorag eAllowed	The maximum storage in GB allowed for the application.	ApplicationId	N/A
MemoryAllocated	The total memory in GB allocated.	ApplicationId	Applicati onId ,WorkerTyp e ,CapacityA llocationType
PendingCr eationWor kerCount	The number of total workers pending creation.	ApplicationId	Applicati onId ,WorkerTyp e ,CapacityA llocationType
RunningWo rkerCount	The number of total workers in use by the application.	ApplicationId	Applicati onId ,WorkerTyp e ,CapacityA llocationType

Metric	Description	Primary dimension	Secondary dimension
StorageAllocated	The total disk storage in GB allocated.	ApplicationId	ApplicationId ,WorkerType ,CapacityAllocationType
TotalWorkerCount	The number of total workers available.	ApplicationId	ApplicationId ,WorkerType ,CapacityAllocationType

Job-level monitoring

Amazon EMR Serverless sends the following job-level metrics to Amazon CloudWatch every one minute. You can view the metric values for aggregate job runs by job run state. The unit for each of the metrics is *count*.

EMR Serverless job-level metrics

Metric	Description	Primary dimension
SubmittedJobs	The number of jobs in a Submitted state.	ApplicationId
PendingJobs	The number of jobs in a Pending state.	ApplicationId
ScheduledJobs	The number of jobs in a Scheduled state.	ApplicationId
RunningJobs	The number of jobs in a Running state.	ApplicationId
SuccessJobs	The number of jobs in a Success state.	ApplicationId

Metric	Description	Primary dimension
`FailedJobs`	The number of jobs in a Failed state.	`ApplicationId`
`CancellingJobs`	The number of jobs in a Cancelling state.	`ApplicationId`
`CancelledJobs`	The number of jobs in a Cancelled state.	`ApplicationId`

You can monitor engine-specific metrics for both running and completed EMR Serverless jobs with engine-specific application UIs. When you view the UI for a running job, you see the live application UI with real-time updates. When you view the UI for a completed job, you see the persistent app UI.

Running jobs

For your running EMR Serverless jobs, you can view a real-time interface that provides engine-specific metrics. You can use either the Apache Spark UI or the Hive Tez UI to monitor and debug your jobs. To access these UIs, use the EMR Studio console or request a secure URL endpoint with the AWS Command Line Interface.

Completed jobs

For your completed EMR Serverless jobs, you can use the Spark History Server or the Persistent Hive Tez UI to view jobs details, stages, tasks, and metrics for Spark or Hive jobs runs. To access these UIs, use the EMR Studio console, or request a secure URL endpoint with the AWS Command Line Interface.

Job worker-level monitoring

Amazon EMR Serverless sends the following job worker level metrics that are available in the `AWS/EMRServerless` namespace and `Job Worker Metrics` metric group to Amazon CloudWatch. EMR Serverless collects data points from individual workers during job runs at the job level, worker-type, and the capacity-allocation-type level. You can use `ApplicationId` as a dimension to monitor multiple jobs that belong to the same application.

EMR Serverless job worker-level metrics

Metric	Description	Unit	Primary dimension	Secondary dimension
WorkerCpuAllocated	The total numbers of vCPU cores allocated for workers in a job run.	None	JobId	ApplicationId , WorkerType , and CapacityAllocationType
WorkerCpuUsed	The total numbers of vCPU cores utilized by workers in a job run.	None	JobId	ApplicationId , WorkerType , and CapacityAllocationType
WorkerMemoryAllocated	The total memory in GB allocated for workers in a job run.	Gigabytes (GB)	JobId	ApplicationId , WorkerType , and CapacityAllocationType
WorkerMemoryUsed	The total memory in GB utilized by workers in a job run.	Gigabytes (GB)	JobId	ApplicationId , WorkerType , and CapacityAllocationType
WorkerEphemeralStorageAllocated	The number of bytes of ephemeral storage allocated for	Gigabytes (GB)	JobId	ApplicationId , WorkerType , and CapacityA

Metric	Description	Unit	Primary dimension	Secondary dimension
	workers in a job run.			llocation Type
WorkerEphemeralStorageUsed	The number of bytes of ephemeral storage used by workers in a job run.	Gigabytes (GB)	JobId	ApplicationId , WorkerType , and CapacityAllocation Type
WorkerStorageReadBytes	The number of bytes read from storage by workers in a job run.	Bytes	JobId	ApplicationId , WorkerType , and CapacityAllocation Type
WorkerStorageWriteBytes	The number of bytes written to storage from workers in a job run.	Bytes	JobId	ApplicationId , WorkerType , and CapacityAllocation Type

The steps below describe how to view the various types of metrics.

Console

To access your application UI with the console

1. Navigate to your EMR Serverless application on the EMR Studio with the instructions in Getting started from the console.
2. To view engine-specific application UIs and logs for a running job:

   a. Choose a job with a RUNNING status.

b. Select the job on the **Application details** page, or navigate to the **Job details** page for your job.

c. Under the **Display UI** dropdown menu, choose either **Spark UI** or **Hive Tez UI** to navigate to the application UI for your job type.

d. To view Spark engine logs, navigate to the **Executors** tab in the Spark UI, and choose the **Logs** link for the driver. To view Hive engine logs, choose the **Logs** link for the appropriate DAG in the Hive Tez UI.

3. To view engine-specific application UIs and logs for a completed job:

a. Choose a job with a SUCCESS status.

b. Select the job on your application's **Application details** page or navigate to the job's **Job details** page.

c. Under the **Display UI** dropdown menu, choose either **Spark History Server** or **Persistent Hive Tez UI** to navigate to the application UI for your job type.

d. To view Spark engine logs, navigate to the **Executors** tab in the Spark UI, and choose the **Logs** link for the driver. To view Hive engine logs, choose the **Logs** link for the appropriate DAG in the Hive Tez UI.

AWS CLI

To access your application UI with the AWS CLI

- To generate a URL that you can use to access your application UI for both running and completed jobs, call the `GetDashboardForJobRun` API.

```
aws emr-serverless get-dashboard-for-job-run /
--application-id <application-id> /
--job-run-id <job-id>
```

The URL that you generate is valid for one hour.

Monitor Spark metrics with Amazon Managed Service for Prometheus

With Amazon EMR releases 7.1.0 and higher, you can integrate EMR Serverless with Amazon Managed Service for Prometheus to collect Apache Spark metrics for EMR Serverless jobs and

applications. This integration is available when you submit a job or create an application using either the AWS console, the EMR Serverless API, or the AWS CLI.

Prerequisites

Before you can deliver your Spark metrics to Amazon Managed Service for Prometheus, you must complete the following prerequisites.

- Create an Amazon Managed Service for Prometheus workspace. This workspace serves as an ingestion endpoint. Make a note of the URL displayed for **Endpoint - remote write URL**. You'll need to specify the URL when you create your EMR Serverless application.
- To grant access of your jobs to Amazon Managed Service for Prometheus for monitoring purposes, add the following policy to your job execution role.

```
{
    "Sid": "AccessToPrometheus",
    "Effect": "Allow",
    "Action": ["aps:RemoteWrite"],
    "Resource": "arn:aws:aps:<AWS_REGION>:<AWS_ACCOUNT_ID>:workspace/<WORKSPACE_ID>"
}
```

Setup

To use the AWS console to create an application that's integrated with Amazon Managed Service for Prometheus

1. See Getting started with Amazon EMR Serverless to create an application.
2. While you're creating an application, choose **Use custom settings**, and then configure your application by specifying the information into the fields you want to configure.
3. Under **Application logs and metrics**, choose **Deliver engine metrics to Amazon Managed Service for Prometheus**, and then specify your remote write URL.
4. Specify any other configuration settings you want, and then choose **Create and start application**.

Use the AWS CLI or EMR Serverless API

You can also use the AWS CLI or EMR Serverless API to integrate your EMR Serverless application with Amazon Managed Service for Prometheus when you're running the `create-application` or the `start-job-run` commands.

create-application

```
aws emr-serverless create-application \
--release-label emr-7.1.0 \
--type "SPARK" \
--monitoring-configuration '{
    "prometheusMonitoringConfiguration": {
        "remoteWriteUrl": "https://aps-workspaces.<AWS_REGION>.amazonaws.com/
workspaces/<WORKSPACE_ID>/api/v1/remote_write"
    }
}'
```

start-job-run

```
aws emr-serverless start-job-run \
--application-id <APPPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--job-driver '{
    "sparkSubmit": {
        "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",
        "entryPointArguments": ["10000"],
        "sparkSubmitParameters": "--conf spark.dynamicAllocation.maxExecutors=10"
    }
}' \
--configuration-overrides '{
     "monitoringConfiguration": {
        "prometheusMonitoringConfiguration": {
            "remoteWriteUrl": "https://aps-workspaces.<AWS_REGION>.amazonaws.com/
workspaces/<WORKSPACE_ID>/api/v1/remote_write"
        }
    }
}'
```

Including `prometheusMonitoringConfiguration` in your command indicates that EMR Serverless must run the Spark job with an agent that collects the Spark metrics and writes them to your `remoteWriteUrl` endpoint for Amazon Managed Service for Prometheus. You can then

use the Spark metrics in Amazon Managed Service for Prometheus for visualization, alerts, and analysis.

Advanced configuration properties

EMR Serverless uses a component within Spark named `PrometheusServlet` to collect Spark metrics and translates performance data into data that's compatible with Amazon Managed Service for Prometheus. By default, EMR Serverless sets default values in Spark and parses driver and executor metrics when you submit a job using `PrometheusMonitoringConfiguration`.

The following table describes all of the properties you can configure when submitting a Spark job that sends metrics to Amazon Managed Service for Prometheus.

Spark property	Default value	Description
`spark.metrics.conf.*.sink.prometheusServlet.class`	org.apache.spark.metrics.sink.PrometheusServlet	The class that Spark uses to send metrics to Amazon Managed Service for Prometheus. To override the default behavior, specify your own custom class.
`spark.metrics.conf.*.source.jvm.class`	org.apache.spark.metrics.source.JvmSource	The class Spark uses to collect and send crucial metrics from the underlying Java virtual machine. To stop collecting JVM metrics, disable this property by setting it to an empty string, such as "". To override the default behavior, specify your own custom class.
`spark.metrics.conf.driver.sink.prometheusServlet.path`	/metrics/prometheus	The distinct URL that Amazon Managed Service for Prometheus uses to collect metrics from the driver. To override the default behavior,

Spark property	Default value	Description
		specify your own path. To stop collecting driver metrics, disable this property by setting it to an empty string, such as "".
`spark.metrics.conf.executor.sink.prometheusServlet.path`	/metrics/executor/prometheus	The distinct URL that Amazon Managed Service for Prometheus uses to collect metrics from the executor. To override the default behavior, specify your own path. To stop collecting executor metrics, disable this property by setting it to an empty string, such as "".

For more information about the Spark metrics, see Apache Spark metrics.

Considerations and limitations

When using Amazon Managed Service for Prometheus to collect metrics from EMR Serverless, consider the following considerations and limitations.

- Support for using Amazon Managed Service for Prometheus with EMR Serverless is available only in the AWS Regions where Amazon Managed Service for Prometheus is generally available.
- Running the agent to collect Spark metrics on Amazon Managed Service for Prometheus requires more resources from workers. If you choose a smaller worker size, such as one vCPU worker, your job run time might increase.
- Support for using Amazon Managed Service for Prometheus with EMR Serverless is available only for Amazon EMR releases 7.1.0 and higher.
- Amazon Managed Service for Prometheus must be deployed in the same account where you run EMR Serverless in order to collect metrics.

EMR Serverless usage metrics

You can use Amazon CloudWatch usage metrics to provide visibility into the resources that your account uses. Use these metrics to visualize your service usage on CloudWatch graphs and dashboards.

EMR Serverless usage metrics correspond to Service Quotas. You can configure alarms that alert you when your usage approaches a service quota. For more information, see Service Quotas and Amazon CloudWatch alarms in the *Service Quotas User Guide*.

For more information about EMR Serverless service quotas, see Endpoints and quotas for EMR Serverless.

Service quota usage metrics for EMR Serverless

EMR Serverless publishes the following service quota usage metrics in the `AWS/Usage` namespace.

Metric	Description
`ResourceCount`	The total number of the specified resource that is running on your account. The resource is defined by the dimensions that are associate d with the metric.

Dimensions for EMR Serverless service quota usage metrics

You can use the following dimensions to refine the usage metrics that EMR Serverless publishes.

Dimension	Value	Description
`Service`	EMR Serverless	The name of the AWS service that contains the resource.
`Type`	Resource	The type of entity that EMR Serverless is reporting.
`Resource`	vCPU	The type of resource that EMR Serverless is tracking.

Dimension	Value	Description
`Class`	None	The class of resource that EMR Serverless is tracking.

Automating EMR Serverless with Amazon EventBridge

You can use Amazon EventBridge to automate your AWS services and respond automatically to system events, such as application availability issues or resource changes. EventBridge delivers a near real-time stream of system events that describe changes in your AWS resources. You can write simple rules to indicate which events are of interest to you, and what automated actions to take when an event matches a rule. With EventBridge, you can automatically:

- Invoke an AWS Lambda function
- Relay an event to Amazon Kinesis Data Streams
- Activate an AWS Step Functions state machine
- Notify an Amazon SNS topic or an Amazon SQS queue

For example, when you use EventBridge with EMR Serverless, you can activate an AWS Lambda function when an ETL job succeed or notify an Amazon SNS topic when an ETL job fails.

EMR Serverless emits four kinds of events:

- Application state change events – Events that emit every state change of an application. For more information about application states, see Application states.
- Job run state change events – Events that emit every state change of a job run. For more information about, see Job run states.
- Job run retry events – Events that emit every retry of a job run from Amazon EMR Serverless releases 7.1.0 and higher.
- Job resource utilization update events – Events that emit resource utilization updates for a job run at close to 30-minute intervals.

Sample EMR Serverless EventBridge events

Events reported by EMR Serverless have a value of `aws.emr-serverless` assigned to `source`, as in the following examples.

Application state change event

The following example event shows an application in the `CREATING` state.

```
{
    "version": "0",
    "id": "9fd3cf79-1ff1-b633-4dd9-34508dc1e660",
    "detail-type": "EMR Serverless Application State Change",
    "source": "aws.emr-serverless",
    "account": "123456789012",
    "time": "2022-05-31T21:16:31Z",
    "region": "us-east-1",
    "resources": [],
    "detail": {
        "applicationId": "00f1cbsc6anuij25",
        "applicationName": "3965ad00-8fba-4932-a6c8-ded32786fd42",
        "arn": "arn:aws:emr-serverless:us-east-1:111122223333:/
applications/00f1cbsc6anuij25",
        "releaseLabel": "emr-6.6.0",
        "state": "CREATING",
        "type": "HIVE",
        "createdAt": "2022-05-31T21:16:31.547953Z",
        "updatedAt": "2022-05-31T21:16:31.547970Z",
        "autoStopConfig": {
            "enabled": true,
            "idleTimeout": 15
        },
        "autoStartConfig": {
            "enabled": true
        }
    }
}
```

Job run state change event

The following example event shows a job run that moves from the `SCHEDULED` state to the `RUNNING` state.

```
{
    "version": "0",
    "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",
    "detail-type": "EMR Serverless Job Run State Change",
    "source": "aws.emr-serverless",
    "account": "123456789012",
    "time": "2022-05-31T21:07:42Z",
    "region": "us-east-1",
    "resources": [],
    "detail": {
        "jobRunId": "00f1cbn5g4bb0c01",
        "applicationId": "00f1982r1uukb925",
        "arn": "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/00f1982r1uukb925/jobruns/00f1cbn5g4bb0c01",
        "releaseLabel": "emr-6.6.0",
        "state": "RUNNING",
        "previousState": "SCHEDULED",
        "createdBy": "arn:aws:sts::123456789012:assumed-role/
TestRole-402dcef3ad14993c15d28263f64381e4cda34775/6622b6233b6d42f59c25dd2637346242",
        "updatedAt": "2022-05-31T21:07:42.299487Z",
        "createdAt": "2022-05-31T21:07:25.325900Z"
    }
}
```

Job run retry event

The following is an example of a job run retry event.

```
{
    "version": "0",
    "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",
    "detail-type": "EMR Serverless Job Run Retry",
    "source": "aws.emr-serverless",
    "account": "123456789012",
    "time": "2022-05-31T21:07:42Z",
    "region": "us-east-1",
    "resources": [],
    "detail": {
        "jobRunId": "00f1cbn5g4bb0c01",
        "applicationId": "00f1982r1uukb925",
        "arn": "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/00f1982r1uukb925/jobruns/00f1cbn5g4bb0c01",
        "releaseLabel": "emr-6.6.0",
```

```
        "createdBy": "arn:aws:sts::123456789012:assumed-role/
TestRole-402dcef3ad14993c15d28263f64381e4cda34775/6622b6233b6d42f59c25dd2637346242",
        "updatedAt": "2022-05-31T21:07:42.299487Z",
        "createdAt": "2022-05-31T21:07:25.325900Z",
        //Attempt Details
        "previousAttempt": 1,
        "previousAttemptState": "FAILED",
        "previousAttemptCreatedAt": "2022-05-31T21:07:25.325900Z",
        "previousAttemptEndedAt": "2022-05-31T21:07:30.325900Z",
        "newAttempt": 2,
        "newAttemptCreatedAt": "2022-05-31T21:07:30.325900Z"
    }
}
```

Job Resource Utilization Update

The following example event shows the final resource utilization update for a job that moved to a terminal state after running.

```
{
    "version": "0",
    "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",
    "detail-type": "EMR Serverless Job Resource Utilization Update",
    "source": "aws.emr-serverless",
    "account": "123456789012",
    "time": "2022-05-31T21:07:42Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:emr-serverless:us-east-1:123456789012:/applications/00f1982r1uukb925/
jobruns/00f1cbn5g4bb0c01"
    ],
    "detail": {
        "applicationId": "00f1982r1uukb925",
        "jobRunId": "00f1cbn5g4bb0c01",
        "attempt": 1,
        "mode": "BATCH",
        "createdAt": "2022-05-31T21:07:25.325900Z",
        "startedAt": "2022-05-31T21:07:26.123Z",
        "calculatedFrom": "2022-05-31T21:07:42.299487Z",
        "calculatedTo": "2022-05-31T21:07:30.325900Z",
        "resourceUtilizationFinal": true,
        "resourceUtilizationForInterval": {
            "vCPUHour": 0.023,
```

```
            "memoryGBHour": 0.114,
            "storageGBHour": 0.228
        },
        "billedResourceUtilizationForInterval": {
            "vCPUHour": 0.067,
            "memoryGBHour": 0.333,
            "storageGBHour": 0
        },
        "totalResourceUtilization": {
            "vCPUHour": 0.023,
            "memoryGBHour": 0.114,
            "storageGBHour": 0.228
        },
        "totalBilledResourceUtilization": {
            "vCPUHour": 0.067,
            "memoryGBHour": 0.333,
            "storageGBHour": 0
        }
    }
}
```

The **startedAt** field will only be present in the event if the job had moved to a running state.

Tagging resources

You can assign your own metadata to each resource using tags to help you manage your EMR Serverless resources. This section provides an overview of the tag functions and shows you how to create tags.

Topics

What is a tag?

A tag is a label that you assign to an AWS resource. Each tag consists of a key and a value, both of which you define. Tags enable you to categorize your AWS resources by attributes such as purpose, owner, and environment. When you have many resources of the same type, you can quickly identify a specific resource based on the tags you've assigned to it. For example, you can define a set of tags for your Amazon EMR Serverless applications to help you track each application's owner and stack level. We recommend that you devise a consistent set of tag keys for each resource type.

Tags are not automatically assigned to your resources. After you add a tag to a resource, you can modify a tag's value or remove the tag from the resource at any time. Tags do not have any semantic meaning to Amazon EMR Serverless and are interpreted strictly as strings of characters. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the earlier value.

If you use IAM, you can control which users in your AWS account have permission to manage tags. For tag-based access control policy examples, see Policies for tag-based access control.

Tagging your resources

You can tag new or existing applications and job runs. If you're using the Amazon EMR Serverless API, the AWS CLI, or an AWS SDK, you can apply tags to new resources using the `tags` parameter on the relevant API action. You can apply tags to existing resources using the `TagResource` API action.

You can use some resource-creating actions to specify tags for a resource when the resource is created. In this case, if tags cannot be applied while the resource is being created, the resource fails to be created. This mechanism ensures that resources you intended to tag on creation are either created with specified tags or not created at all. If you tag resources at the time of creation, you do not need to run custom tagging scripts after creating a resource.

The following table describes the Amazon EMR Serverless resources that can be tagged.

Resource	Supports tags	Supports tag propagation	Supports tagging on creation (Amazon EMR Serverless API, AWS CLI, and AWS SDK)	API for creation (tags can be added during creation)
Application	Yes	No. Tags associated with an application do not propagate to job runs submitted to that application.	Yes	`CreateApplication`
Job run	Yes	No	Yes	`StartJobRun`

Tagging limitations

The following basic limitations apply to tags:

- Each resource can have a maximum of 50 user-created tags.
- For each resource, each tag key must be unique, and each tag key can have only one value.
- The maximum key length is 128 Unicode characters in UTF-8.
- The maximum value length is 256 Unicode characters in UTF-8.

- Allowed characters are letters, numbers, spaces representable in UTF-8, and the following characters: _ . : / = + - @.
- A tag key cannot be an empty string. A tag value can be an empty string, but not null.
- Tag keys and values are case sensitive.
- Do not use `AWS:` or any upper or lowercase combination of such as a prefix for either keys or values. These are reserved only for AWS use.

Working with tags using the AWS CLI and the Amazon EMR Serverless API

Use the following AWS CLI commands or Amazon EMR Serverless API operations to add, update, list, and delete the tags for your resources.

Resource	Supports tags	Supports tag propagation
Add or overwrite one or more tags	`tag-resource`	`TagResource`
List tags for a resource	`list-tags-for-resource`	`ListTagsForResource`
Delete one or more tags	`untag-resource`	`UntagResource`

The following examples show how to tag or untag resources using the AWS CLI.

Tag an existing application

The following command tags an existing application.

```
aws emr-serverless tag-resource --resource-arn resource_ARN --tags team=devs
```

Untag an existing application

The following command deletes a tag from an existing application.

```
aws emr-serverless untag-resource --resource-arn resource_ARN --tag-keys tag_key
```

List tags for a resource

The following command lists the tags associated with an existing resource.

```
aws emr-serverless list-tags-for-resource --resource-arn resource_ARN
```

Tutorials for EMR Serverless

This section describes common use cases when you work with EMR Serverless applications. This includes a variety of tools including Hudi and Iceberg for working on large data sets and using Python and Python libraries to submit Spark jobs.

Topics

Using Java 17 with Amazon EMR Serverless

With Amazon EMR releases 6.11.0 and higher, you can configure EMR Serverless Spark jobs to use Java 17 runtime for the Java Virtual Machine (JVM). Use one of the following methods to configure Spark with Java 17.

`JAVA_HOME`

To override the JVM setting for EMR Serverless 6.11.0 and higher, you can supply the `JAVA_HOME` setting to its `spark.emr-serverless.driverEnv` and `spark.executorEnv` environment classifications.

x86_64

Set the required properties to specify Java 17 as the `JAVA_HOME` configuration for the Spark driver and executors:

```
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-
corretto.x86_64/
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.x86_64/
```

arm_64

Set the required properties to specify Java 17 as the `JAVA_HOME` configuration for the Spark driver and executors:

```
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-
corretto.aarch64/
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.aarch64/
```

`spark-defaults`

Alternatively, you can specify Java 17 in the `spark-defaults` classification to override the JVM setting for EMR Serverless 6.11.0 and higher.

x86_64

Specify Java 17 in the `spark-defaults` classification:

```
{
"applicationConfiguration": [
         {
            "classification": "spark-defaults",
            "properties": {
               "spark.emr-serverless.driverEnv.JAVA_HOME" : "/usr/lib/jvm/java-17-
amazon-corretto.x86_64/",
               "spark.executorEnv.JAVA_HOME": "/usr/lib/jvm/java-17-amazon-
corretto.x86_64/"
            }
         }
      ]
}
```

arm_64

Specify Java 17 in the `spark-defaults` classification:

```
{
```

```
"applicationConfiguration": [
         {
            "classification": "spark-defaults",
            "properties": {
               "spark.emr-serverless.driverEnv.JAVA_HOME" : "/usr/lib/jvm/java-17-
amazon-corretto.aarch64/",
               "spark.executorEnv.JAVA_HOME": "/usr/lib/jvm/java-17-amazon-
corretto.aarch64/"
            }
         }
      ]
}
```

Using Apache Hudi with EMR Serverless

This section describes using Apache Hudi with EMR Serverless applications. Hudi is a data-management framework that makes data processing more simple.

To use Apache Hudi with EMR Serverless applications

1. Set the required Spark properties in the corresponding Spark job run.

   ```
   spark.jars=/usr/lib/hudi/hudi-spark-bundle.jar
   spark.serializer=org.apache.spark.serializer.KryoSerializer
   ```

2. To sync a Hudi table to the configured catalog, designate either the AWS Glue Data Catalog as your metastore, or configure an external metastore. EMR Serverless supports `hms` as the sync mode for Hive tables for Hudi workloads. EMR Serverless activates this property as a default. To learn more about how to set up your metastore, see Metastore configuration for EMR Serverless.

   > **Important**
   >
   > EMR Serverless doesn't support `HIVEQL` or `JDBC` as sync mode options for Hive tables to handle Hudi workloads. To learn more, see Sync modes.

   When you use the AWS Glue Data Catalog as your metastore, you can specify the following configuration properties for your Hudi job.

```
--conf spark.jars=/usr/lib/hudi/hudi-spark-bundle.jar,
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer,
--conf
 spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSG
```

To learn more about Apache Hudi releases of Amazon EMR, see Hudi release history.

Using Apache Iceberg with EMR Serverless

This section describes how to use Apache Iceberg with EMR Serverless applications. Apache Iceberg is a table format that helps working with large data sets in data lakes.

To use Apache Iceberg with EMR Serverless applications

1. Set the required Spark properties in the corresponding Spark job run.

   ```
   spark.jars=/usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar
   ```

2. Designate either the AWS Glue Data Catalog as your metastore or configure an external metastore. To learn more about setting up your metastore, see Metastore configuration for EMR Serverless.

   Configure the metastore properties that you want to use for Iceberg. For example, if you want to use the AWS Glue Data Catalog, set the following properties in the application configuration.

   ```
   spark.sql.catalog.dev.warehouse=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/
   spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
   spark.sql.catalog.dev=org.apache.iceberg.spark.SparkCatalog
   spark.sql.catalog.dev.catalog-impl=org.apache.iceberg.aws.glue.GlueCatalog
   spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGl
   ```

   When you use the AWS Glue Data Catalog as your metastore, you can specify the following configuration properties for your Iceberg job.

   ```
   --conf spark.jars=/usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar,
   --conf
    spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,
   ```

```
--conf spark.sql.catalog.dev=org.apache.iceberg.spark.SparkCatalog,
--conf spark.sql.catalog.dev.catalog-impl=org.apache.iceberg.aws.glue.GlueCatalog,
--conf spark.sql.catalog.dev.warehouse=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/
--conf
 spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSG
```

To learn more about Apache Iceberg releases of Amazon EMR, see Iceberg release history.

Using Python libraries with EMR Serverless

When you run PySpark jobs on Amazon EMR Serverless applications, you can package various Python libraries as dependencies. To do this, you can use native Python features, build a virtual environment, or directly configure your PySpark jobs to use Python libraries. This page covers each approach.

Using native Python features

When you set the following configuration, you can use PySpark to upload Python files (`.py`), zipped Python packages (`.zip`), and Egg files (`.egg`) to Spark executors.

```
--conf spark.submit.pyFiles=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/<.py|.egg|.zip
 file>
```

For more details about how to use Python virtual environments for PySpark jobs, see Using PySpark Native Features.

Building a Python virtual environment

To package multiple Python libraries for a PySpark job, you can create isolated Python virtual environments.

1. To build the Python virtual environment, use the following commands. The example shown installs the packages `scipy` and `matplotlib` into a virtual environment package and copies the archive to an Amazon S3 location.

   > **Important**
   >
   > You must run the following commands in a similar Amazon Linux 2 environment with the same version of Python as you use in EMR Serverless, that is, Python 3.7.10 for

Amazon EMR release 6.6.0. You can find an example Dockerfile in the EMR Serverless Samples GitHub repository.

```
# initialize a python virtual environment
python3 -m venv pyspark_venvsource
source pyspark_venvsource/bin/activate

# optionally, ensure pip is up-to-date
pip3 install --upgrade pip

# install the python packages
pip3 install scipy
pip3 install matplotlib

# package the virtual environment into an archive
pip3 install venv-pack
venv-pack -f -o pyspark_venv.tar.gz

# copy the archive to an S3 location
aws s3 cp pyspark_venv.tar.gz s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/

# optionally, remove the virtual environment directory
rm -fr pyspark_venvsource
```

2. Submit the Spark job with your properties set to use the Python virtual environment.

```
--conf spark.archives=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/
pyspark_venv.tar.gz#environment
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=./environment/bin/
python
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=./environment/bin/python
--conf spark.executorEnv.PYSPARK_PYTHON=./environment/bin/python
```

Note that if you don't override the original Python binary, the second configuration in the preceding sequence of settings will be `--conf spark.executorEnv.PYSPARK_PYTHON=python`.

For more on how to use Python virtual environments for PySpark jobs, see Using Virtualenv. For more examples of how to submit Spark jobs, see Using Spark configurations when you run EMR Serverless jobs.

Configuring PySpark jobs to use Python libraries

With Amazon EMR releases 6.12.0 and higher, you can directly configure EMR Serverless PySpark jobs to use popular data science Python libraries like pandas, NumPy, and PyArrow without any additional setup.

The following examples show how to package each Python library for a PySpark job.

NumPy

NumPy is a Python library for scientific computing that offers multidimensional arrays and operations for math, sorting, random simulation, and basic statistics. To use NumPy, run the following command:

```
import numpy
```

pandas

pandas is a Python library that is built on top of NumPy. The pandas library provides datas scientists with DataFrame data structures and data analysis tools. To use pandas, run the following command:

```
import pandas
```

PyArrow

PyArrow is a Python library that manages in-memory columnar data for improved job performance. PyArrow is based on the Apache Arrow cross-language development specification, which is a standard way to represent and exchange data in a columnar format. To use PyArrow, run the following command:

```
import pyarrow
```

Using different Python versions with EMR Serverless

In addition to the use case in Using Python libraries with EMR Serverless, you can also use Python virtual environments to work with different Python versions than the version packaged in the Amazon EMR release for your Amazon EMR Serverless application. To do this, you must build a Python virtual environment with the Python version you want to use.

To submit a job from a Python virtual environment

1. Build your virtual environment with the commands in the following example. This example installs Python 3.9.9 into a virtual environment package and copies the archive to an Amazon S3 location.

> **Important**
>
> If you use Amazon EMR releases 7.0.0 and higher, you must run your commands in an Amazon Linux 2023 environment similar to the one you use for your EMR Serverless applications.
> If you use release 6.15.0 or lower, you must run the following commands in a similar Amazon Linux 2 environment.

```
# install Python 3.9.9 and activate the venv
yum install -y gcc openssl-devel bzip2-devel libffi-devel tar gzip wget make
wget https://www.python.org/ftp/python/3.9.9/Python-3.9.9.tgz && \
tar xzf Python-3.9.9.tgz && cd Python-3.9.9 && \
./configure --enable-optimizations && \
make altinstall

# create python venv with Python 3.9.9
python3.9 -m venv pyspark_venv_python_3.9.9 --copies
source pyspark_venv_python_3.9.9/bin/activate

# copy system python3 libraries to venv
cp -r /usr/local/lib/python3.9/* ./pyspark_venv_python_3.9.9/lib/python3.9/

# package venv to archive.
# **Note** that you have to supply --python-prefix option
# to make sure python starts with the path where your
# copied libraries are present.
# Copying the python binary to the "environment" directory.
pip3 install venv-pack
venv-pack -f -o pyspark_venv_python_3.9.9.tar.gz --python-prefix /home/hadoop/
environment

# stage the archive in S3
aws s3 cp pyspark_venv_python_3.9.9.tar.gz s3://<path>
```

```
# optionally, remove the virtual environment directory
rm -fr pyspark_venv_python_3.9.9
```

2. Set your properties to use the Python virtual environment and submit the Spark job.

```
# note that the archive suffix "environment" is the same as the directory where you
 copied the Python binary.
--conf spark.archives=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/
pyspark_venv_python_3.9.9.tar.gz#environment
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=./environment/bin/
python
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=./environment/bin/python
--conf spark.executorEnv.PYSPARK_PYTHON=./environment/bin/python
```

For more on how to use Python virtual environments for PySpark jobs, see Using Virtualenv. For more examples of how to submit Spark jobs, see Using Spark configurations when you run EMR Serverless jobs.

Using Delta Lake OSS with EMR Serverless

Amazon EMR versions 6.9.0 and higher

> **Note**
>
> Amazon EMR 7.0.0 and higher uses Delta Lake 3.0.0, which renames the `delta-core.jar` file to `delta-spark.jar`. If you use Amazon EMR 7.0.0 or higher, make sure to specify `delta-spark.jar` in your configurations.

Amazon EMR 6.9.0 and higher includes Delta Lake, so you no longer have to package Delta Lake yourself or provide the `--packages` flag with your EMR Serverless jobs.

1. When you submit EMR Serverless jobs, make sure that you have the following configuration properties and include the following parameters in the `sparkSubmitParameters` field.

```
--conf spark.jars=/usr/share/aws/delta/lib/delta-core.jar,/usr/share/aws/delta/lib/
delta-storage.jar
    --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
```

```
    --conf
 spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
```

2. Create a local `delta_sample.py` to test creating and reading a Delta table.

```
# delta_sample.py
    from pyspark.sql import SparkSession

    import uuid

    url = "s3://amzn-s3-demo-bucket/delta-lake/output/%s/" % str(uuid.uuid4())
    spark = SparkSession.builder.appName("DeltaSample").getOrCreate()

    ## creates a Delta table and outputs to target S3 bucket
    spark.range(5).write.format("delta").save(url)

    ## reads a Delta table and outputs to target S3 bucket
    spark.read.format("delta").load(url).show
```

3. Using the AWS CLI, upload the `delta_sample.py` file to your Amazon S3 bucket. Then use the `start-job-run` command to submit a job to an existing EMR Serverless application.

```
aws s3 cp delta_sample.py s3://amzn-s3-demo-bucket/code/

    aws emr-serverless start-job-run \
        --application-id application-id \
        --execution-role-arn job-role-arn \
        --name emr-delta \
        --job-driver '{
            "sparkSubmit": {
                "entryPoint": "s3://amzn-s3-demo-bucket/code/delta_sample.py",
                "sparkSubmitParameters": "--conf spark.jars=/usr/share/
aws/delta/lib/delta-core.jar,/usr/share/aws/delta/lib/delta-storage.jar --
conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf
 spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
            }
        }'
```

To use Python libraries with Delta Lake, you can add the `delta-core` library by packaging it as a dependency or by using it as a custom image.

Alternatively, you can use the `SparkContext.addPyFile` to add the Python libraries from the `delta-core` JAR file:

```
import glob
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
spark.sparkContext.addPyFile(glob.glob("/usr/share/aws/delta/lib/delta-core_*.jar")[0])
```

Amazon EMR versions 6.8.0 and lower

If you're using Amazon EMR 6.8.0 or lower, follow these steps to use Delta Lake OSS with your EMR Serverless applications.

1. To build an open source version of Delta Lake that's compatible with the version of Spark on your Amazon EMR Serverless application, navigate to the Delta GitHub and follow the instructions.
2. Upload the Delta Lake libraries to an Amazon S3 bucket in your AWS account.
3. When you submit EMR Serverless jobs in the application configuration, include the Delta Lake JAR files that are now in your bucket.

   ```
   --conf spark.jars=s3://amzn-s3-demo-bucket/jars/delta-core_2.12-1.1.0.jar
   ```

4. To ensure that you can read to and write from a Delta table, run a sample PySpark test.

   ```
   from pyspark import SparkConf, SparkContext
       from pyspark.sql import HiveContext, SparkSession

       import uuid

       conf = SparkConf()
       sc = SparkContext(conf=conf)
       sqlContext = HiveContext(sc)

       url = "s3://amzn-s3-demo-bucket/delta-lake/output/1.0.1/%s/" %
    str(uuid.uuid4())

       ## creates a Delta table and outputs to target S3 bucket
       session.range(5).write.format("delta").save(url)

       ## reads a Delta table and outputs to target S3 bucket
   ```

```
    session.read.format("delta").load(url).show
```

Submitting EMR Serverless jobs from Airflow

The Amazon Provider in Apache Airflow provides EMR Serverless operators. For more information about operators, see Amazon EMR Serverless Operators in the Apache Airflow documentation.

You can use `EmrServerlessCreateApplicationOperator` to create a Spark or Hive application. You can also use `EmrServerlessStartJobOperator` to start one or more jobs with the your new application.

To use the operator with Amazon Managed Workflows for Apache Airflow (MWAA) with Airflow 2.2.2, add the following line to your `requirements.txt` file and update your MWAA environment to use the new file.

```
apache-airflow-providers-amazon==6.0.0
boto3>=1.23.9
```

Note that EMR Serverless support was added to release 5.0.0 of the Amazon provider. Release 6.0.0 is the last version compatible with Airflow 2.2.2. You can use later versions with Airflow 2.4.3 on MWAA.

The following abbreviated example shows how to create an application, run multiple Spark jobs, and then stop the application. A full example is available in the EMR Serverless Samples GitHub repository. For additional details of `sparkSubmit` configuration, see Using Spark configurations when you run EMR Serverless jobs.

```
from datetime import datetime

from airflow import DAG
from airflow.providers.amazon.aws.operators.emr import (
    EmrServerlessCreateApplicationOperator,
    EmrServerlessStartJobOperator,
    EmrServerlessDeleteApplicationOperator,
)

# Replace these with your correct values
JOB_ROLE_ARN = "arn:aws:iam::account-id:role/emr_serverless_default_role"
S3_LOGS_BUCKET = "amzn-s3-demo-bucket"
```

```
DEFAULT_MONITORING_CONFIG = {
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {"logUri": f"s3://amzn-s3-demo-bucket/logs/"}
    },
}

with DAG(
    dag_id="example_endtoend_emr_serverless_job",
    schedule_interval=None,
    start_date=datetime(2021, 1, 1),
    tags=["example"],
    catchup=False,
) as dag:
    create_app = EmrServerlessCreateApplicationOperator(
        task_id="create_spark_app",
        job_type="SPARK",
        release_label="emr-6.7.0",
        config={"name": "airflow-test"},
    )

    application_id = create_app.output

    job1 = EmrServerlessStartJobOperator(
        task_id="start_job_1",
        application_id=application_id,
        execution_role_arn=JOB_ROLE_ARN,
        job_driver={
            "sparkSubmit": {
                "entryPoint": "local:///usr/lib/spark/examples/src/main/python/
pi_fail.py",
            }
        },
        configuration_overrides=DEFAULT_MONITORING_CONFIG,
    )

    job2 = EmrServerlessStartJobOperator(
        task_id="start_job_2",
        application_id=application_id,
        execution_role_arn=JOB_ROLE_ARN,
        job_driver={
            "sparkSubmit": {
                "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",
                "entryPointArguments": ["1000"]
            }
```

```
        },
        configuration_overrides=DEFAULT_MONITORING_CONFIG,
    )

    delete_app = EmrServerlessDeleteApplicationOperator(
        task_id="delete_app",
        application_id=application_id,
        trigger_rule="all_done",
    )

    (create_app >> [job1, job2] >> delete_app)
```

Using Hive user-defined functions with EMR Serverless

Hive user-defined functions (UDFs) let you create custom functions to process records or groups of records. In this tutorial, you'll use a sample UDF with a pre-existing Amazon EMR Serverless application to run a job that outputs a query result. To learn how to set up an application, see Getting started with Amazon EMR Serverless.

To use a UDF with EMR Serverless

1. Navigate to the GitHub for a sample UDF. Clone the repo and switch to the git branch that you want to use. Update the `maven-compiler-plugin` in the `pom.xml` file of the repository to have a source. Also update the target java version configuration to `1.8`. Run `mvn package -DskipTests` to create the JAR file that contains your sample UDFs.
2. After you create the JAR file, upload it to your S3 bucket with the following command.

   ```
   aws s3 cp brickhouse-0.8.2-JS.jar s3://amzn-s3-demo-bucket/jars/
   ```

3. Create an example file to use one of the sample UDF functions. Save this query as `udf_example.q` and upload it to your S3 bucket.

   ```
   add jar s3://amzn-s3-demo-bucket/jars/brickhouse-0.8.2-JS.jar;
   CREATE TEMPORARY FUNCTION from_json AS 'brickhouse.udf.json.FromJsonUDF';
   select from_json('{"key1":[0,1,2], "key2":[3,4,5,6], "key3":[7,8,9]}', map("",
    array(cast(0 as int))));
   select from_json('{"key1":[0,1,2], "key2":[3,4,5,6], "key3":[7,8,9]}', map("",
    array(cast(0 as int))))["key1"][2];
   ```

4. Submit the following Hive job.

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
        "query": "s3://amzn-s3-demo-bucket/queries/udf_example.q",
        "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/
emr-serverless-hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://'$BUCKET'/
emr-serverless-hive/warehouse"
    }
}' --configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "hive-site",
        "properties": {
            "hive.driver.cores": "2",
            "hive.driver.memory": "6G"
        }
    }],
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/logs/"
        }
    }
}'
```

5. Use the `get-job-run` command to check your job's state. Wait for the state to change to SUCCESS.

```
aws emr-serverless get-job-run --application-id application-id --job-run-id job-id
```

6. Download the output files with the following command.

```
aws s3 cp --recursive s3://amzn-s3-demo-bucket/logs/applications/application-id/
jobs/job-id/HIVE_DRIVER/ .
```

The `stdout.gz` file resembles the following.

```
{"key1":[0,1,2],"key2":[3,4,5,6],"key3":[7,8,9]}
2
```

Using custom images with EMR Serverless

Topics

Use a custom Python version

You can build a custom image to use a different version of Python. To use Python version 3.10 for Spark jobs, for example, run the following command:

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

# install python 3
RUN yum install -y gcc openssl-devel bzip2-devel libffi-devel tar gzip wget make
RUN wget https://www.python.org/ftp/python/3.10.0/Python-3.10.0.tgz && \
tar xzf Python-3.10.0.tgz && cd Python-3.10.0 && \
./configure --enable-optimizations && \
make altinstall

# EMRS will run the image as hadoop
USER hadoop:hadoop
```

Before you submit the Spark job, set your properties to use the Python virtual environment, as follows.

```
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=/usr/local/bin/python3.10
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=/usr/local/bin/python3.10
--conf spark.executorEnv.PYSPARK_PYTHON=/usr/local/bin/python3.10
```

Use a custom Java version

The following example demonstrates how to build a custom image to use Java 11 for your Spark jobs.

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

# install JDK 11
RUN sudo amazon-linux-extras install java-openjdk11

# EMRS will run the image as hadoop
USER hadoop:hadoop
```

Before you submit the Spark job, set Spark properties to use Java 11, as follows.

```
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-11-
openjdk-11.0.16.0.8-1.amzn2.0.1.x86_64
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-11-
openjdk-11.0.16.0.8-
```

Build a data science image

The following example shows how to include common, data science Python packages, such as Pandas and NumPy.

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

# python packages
RUN pip3 install boto3 pandas numpy
RUN pip3 install -U scikit-learn==0.23.2 scipy
RUN pip3 install sk-dist
RUN pip3 install xgboost

# EMR Serverless will run the image as hadoop
USER hadoop:hadoop
```

Processing geospatial data with Apache Sedona

The following example shows how to build an image to include Apache Sedona for geospatial processing.

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

RUN yum install -y wget
RUN wget https://repo1.maven.org/maven2/org/apache/sedona/sedona-core-3.0_2.12/1.3.0-
incubating/sedona-core-3.0_2.12-1.3.0-incubating.jar -P /usr/lib/spark/jars/
RUN pip3 install apache-sedona

# EMRS will run the image as hadoop
USER hadoop:hadoop
```

Licensing information for using custom images

You can build custom images with EMR Serverless to perform specific tasks or to use specific versions of a software package. Modification and distribution of custom images can be subject to rules and licensing terms. The licensing text appears in the subsection that follows.

Licensing that applies to custom images

Copyright Amazon.com and its affiliates; all rights reserved. This software is AWS Content under AWS Customer Agreement and may not be distributed without permission. In addition to the permissions in AWS Intellectual Property License, the AWS Licensor grants you these additional permissions:

Create, Copy, and Use Derivatives of the AWS Content is permitted provided that the following conditions are met:

- *You do not modify the AWS Content itself, and any Derivatives are strictly the result of Your addition of new content.*
- *Internal reproductions must retain the above copyright notice.*
- *External distribution, in source or binary form, with or without modification, is not permitted under the terms of this license.*

For more information about using custom images, see Using custom images with EMR Serverless.

Using Amazon Redshift integration for Apache Spark on Amazon EMR Serverless

With Amazon EMR release 6.9.0 and later, every release image includes a connector between Apache Spark and Amazon Redshift. With this connector, you can use Spark on Amazon EMR Serverless to process data stored in Amazon Redshift. The integration is based on the `spark-redshift` open-source connector. For Amazon EMR Serverless, the Amazon Redshift integration for Apache Spark is included as a native integration.

Topics

Launching a Spark application with the Amazon Redshift integration for Apache Spark

To use the integration with EMR Serverless 6.9.0, you must pass the required Spark-Redshift dependencies with your Spark job. Use `--jars` to include Redshift connector related libraries. To see other file locations supported by the `--jars` option, see the Advanced Dependency Management section of the Apache Spark documentation.

- `spark-redshift.jar`
- `spark-avro.jar`
- `RedshiftJDBC.jar`
- `minimal-json.jar`

Amazon EMR releases 6.10.0 and higher don't require the `minimal-json.jar` dependency, and automatically install the other dependencies to each cluster by default. The following examples show how to launch a Spark application with the Amazon Redshift integration for Apache Spark.

Amazon EMR 6.10.0 +

Launch a Spark job on Amazon EMR Serverless with the Amazon Redshift integration for Apache Spark on EMR Serverless release 6.10.0 and higher.

```
spark-submit my_script.py
```

Amazon EMR 6.9.0

To launch a Spark job on Amazon EMR Serverless with the Amazon Redshift integration for Apache Spark on EMR Serverless release 6.9.0, use the `--jars` option as shown in the following example. Note that the paths listed with the `--jars` option are the default paths for the JAR files.

```
--jars
    /usr/share/aws/redshift/jdbc/RedshiftJDBC.jar,
    /usr/share/aws/redshift/spark-redshift/lib/spark-redshift.jar,
    /usr/share/aws/redshift/spark-redshift/lib/spark-avro.jar,
    /usr/share/aws/redshift/spark-redshift/lib/minimal-json.jar
```

```
spark-submit \
  --jars /usr/share/aws/redshift/jdbc/RedshiftJDBC.jar,/usr/share/aws/redshift/
spark-redshift/lib/spark-redshift.jar,/usr/share/aws/redshift/spark-redshift/lib/
spark-avro.jar,/usr/share/aws/redshift/spark-redshift/lib/minimal-json.jar \
  my_script.py
```

Authenticating with the Amazon Redshift integration for Apache Spark

Use AWS Secrets Manager to retrieve credentials and connect to Amazon Redshift

You can securely authenticate to Amazon Redshift by storing the credentials in Secrets Manager and have the Spark job call the `GetSecretValue` API to fetch it:

```
from pyspark.sql import SQLContextimport boto3

sc = # existing SparkContext
sql_context = SQLContext(sc)
```

```
secretsmanager_client = boto3.client('secretsmanager',
 region_name=os.getenv('AWS_REGION'))
secret_manager_response = secretsmanager_client.get_secret_value(
    SecretId='string',
    VersionId='string',
    VersionStage='string'
)
username = # get username from secret_manager_response
password = # get password from secret_manager_response
url = "jdbc:redshift://redshifthost:5439/database?user=" + username + "&password="
 + password

# Access to Redshift cluster using Spark
```

Authenticate to Amazon Redshift with a JDBC driver

Set username and password inside the JDBC URL

You can authenticate a Spark job to an Amazon Redshift cluster by specifying the Amazon Redshift database name and password in the JDBC URL.

> **Note**
>
> If you pass the database credentials in the URL, anyone who has access to the URL can also access the credentials. This method isn't generally recommended because it's not a secure option.

If security isn't a concern for your application, you can use the following format to set the username and password in the JDBC URL:

```
jdbc:redshift://redshifthost:5439/database?user=username&password=password
```

Use IAM based authentication with Amazon EMR Serverless job execution role

Starting with Amazon EMR Serverless release 6.9.0, the Amazon Redshift JDBC driver 2.1 or higher is packaged into the environment. With JDBC driver 2.1 and higher, you can specify the JDBC URL and not include the raw username and password.

Instead, you can specify `jdbc:redshift:iam://` scheme. This commands the JDBC driver to use your EMR Serverless job execution role to fetch the credentials automatically. See Configure a JDBC or ODBC connection to use IAM credentials in the *Amazon Redshift Management Guide* for more information. An example of this URL is:

```
jdbc:redshift:iam://examplecluster.abc123xyz789.us-west-2.redshift.amazonaws.com:5439/
dev
```

The following permissions are required for your job execution role when the provided conditions are met:

Permission	Conditions when required for job execution role
`redshift:GetClusterCredentials`	Required for JDBC driver to fetch the credentials from Amazon Redshift
`redshift:DescribeCluster`	Required if you specify the Amazon Redshift cluster and AWS Region in the JDBC URL instead of endpoint
`redshift-serverless:GetCredentials`	Required for JDBC driver to fetch the credentials from Amazon Redshift Serverless
`redshift-serverless:GetWorkgroup`	Required if you are using Amazon Redshift Serverless and you are specifying the URL in terms of workgroup name and Region

Connecting to Amazon Redshift within a different VPC

When you set up a provisioned Amazon Redshift cluster or Amazon Redshift Serverless workgroup under a VPC, you must configure VPC connectivity for your Amazon EMR Serverless application to access to the resources. For more information on how to configure VPC connectivity on an EMR Serverless application, see Configuring VPC access for EMR Serverless applications to connect to data.

- If your provisioned Amazon Redshift cluster or Amazon Redshift Serverless workgroup is publicly accessible, you can specify one or more private subnets that have a NAT gateway attached when you create EMR Serverless applications.
- If your provisioned Amazon Redshift cluster or Amazon Redshift Serverless workgroup isn't publicly accessible, you must create an Amazon Redshift managed VPC endpoint for your

Amazon Redshift cluster as described in Configuring VPC access for EMR Serverless applications to connect to data. Alternatively, you can create your Amazon Redshift Serverless workgroup as described in Connecting to Amazon Redshift Serverless in the *Amazon Redshift Management Guide*. You must associate your cluster or your subgroup to the private subnets that you specify when you create your EMR Serverless application.

> **Note**
>
> If you use IAM based authentication, and your private subnets for the EMR Serverless application don't have a NAT gateway attached, then you must also create a VPC endpoint on those subnets for Amazon Redshift or Amazon Redshift Serverless. This way, the JDBC driver can fetch the credentials.

Reading and writing from and to Amazon Redshift

The following code examples use PySpark to read and write sample data from and to an Amazon Redshift database with a data source API and with SparkSQL.

Data source API

Use PySpark to read and write sample data from and to an Amazon Redshift database with data source API.

```
import boto3
from pyspark.sql import SQLContext

sc = # existing SparkContext
sql_context = SQLContext(sc)

url = "jdbc:redshift:iam://redshifthost:5439/database"
aws_iam_role_arn = "arn:aws:iam::account-id:role/role-name"

df = sql_context.read \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", url) \
    .option("dbtable", "table-name") \
    .option("tempdir", "s3://path/for/temp/data") \
    .option("aws_iam_role", "aws-iam-role-arn") \
    .load()
```

```
df.write \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", url) \
    .option("dbtable", "table-name-copy") \
    .option("tempdir", "s3://path/for/temp/data") \
    .option("aws_iam_role", "aws-iam-role-arn") \
    .mode("error") \
    .save()
```

SparkSQL

Use PySpark to read and write sample data from and to an Amazon Redshift database with SparkSQL.

```
import boto3
import json
import sys
import os
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .enableHiveSupport() \
    .getOrCreate()

url = "jdbc:redshift:iam://redshifthost:5439/database"
aws_iam_role_arn = "arn:aws:iam::account-id:role/role-name"

bucket = "s3://path/for/temp/data"
tableName = "table-name" # Redshift table name

s = f"""CREATE TABLE IF NOT EXISTS {table-name} (country string, data string)
    USING io.github.spark_redshift_community.spark.redshift
    OPTIONS (dbtable '{table-name}', tempdir '{bucket}', url '{url}', aws_iam_role
 '{aws-iam-role-arn}' ); """

spark.sql(s)

columns = ["country" ,"data"]
data = [("test-country","test-data")]
df = spark.sparkContext.parallelize(data).toDF(columns)
```

```
# Insert data into table
df.write.insertInto(table-name, overwrite=False)
df = spark.sql(f"SELECT * FROM {table-name}")
df.show()
```

Considerations and limitations when using the Spark connector

- We recommend that you turn on SSL for the JDBC connection from Spark on Amazon EMR to Amazon Redshift.
- We recommend that you manage the credentials for the Amazon Redshift cluster in AWS Secrets Manager as a best practice. See Using AWS Secrets Manager to retrieve credentials for connecting to Amazon Redshift for an example.
- We recommend that you pass an IAM role with the parameter `aws_iam_role` for the Amazon Redshift authentication parameter.
- The parameter `tempformat` currently doesn't support the Parquet format.
- The `tempdir` URI points to an Amazon S3 location. This temp directory isn't cleaned up automatically and therefore could add additional cost.
- Consider the following recommendations for Amazon Redshift:
  - We recommend that you block public access to the Amazon Redshift cluster.
  - We recommend that you turn on Amazon Redshift audit logging.
  - We recommend that you turn on Amazon Redshift at-rest encryption.
- Consider the following recommendations for Amazon S3:
  - We recommend that you block public access to Amazon S3 buckets.
  - We recommend that you use Amazon S3 server-side encryption to encrypt the Amazon S3 buckets used.
  - We recommend that you use Amazon S3 lifecycle policies to define the retention rules for the Amazon S3 bucket.
  - Amazon EMR always verifies code imported from open-source into the image. For security, we don't support the following authentication methods from Spark to Amazon S3:
    - Setting AWS access keys in the `hadoop-env` configuration classification
    - Encoding AWS access keys in the `tempdir` URI

For more information on using the connector and its supported parameters, see the following resources:

- Amazon Redshift integration for Apache Spark in the *Amazon Redshift Management Guide*
- The `spark-redshift` community repository on Github

Connecting to DynamoDB with Amazon EMR Serverless

In this tutorial, you upload a subset of data from the United States Board on Geographic Names to an Amazon S3 bucket and then use Hive or Spark on Amazon EMR Serverless to copy the data to an Amazon DynamoDB table that you can query.

Step 1: Upload data to an Amazon S3 bucket

To create an Amazon S3 bucket, follow the instructions in Creating a bucket in the *Amazon Simple Storage Service Console User Guide*. Replace references to *`amzn-s3-demo-bucket`* with the name of your newly created bucket. Now your EMR Serverless application is ready to run jobs.

1. Download the sample data archive `features.zip` with the following command.

```
wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
features.zip
```

2. Extract the `features.txt` file from the archive and view the first the few lines in the file:

```
unzip features.zip
head features.txt
```

   The result should look similar to the following.

```
1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794
875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7
1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10
26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681
1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605
1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558
1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024
533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0
829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671
```

```
541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
```

The fields in each line here indicate a unique identifier, name, type of natural feature, state, latitude in degrees, longitude in degrees, and height in feet.

3. Upload your data to Amazon S3

```
aws s3 cp features.txt s3://amzn-s3-demo-bucket/features/
```

Step 2: Create a Hive table

Use Apache Spark or Hive to create a new Hive table that contains the uploaded data in Amazon S3.

Spark

To create a Hive table with Spark, run the following command.

```
import org.apache.spark.sql.SparkSession

val sparkSession = SparkSession.builder().enableHiveSupport().getOrCreate()

sparkSession.sql("CREATE TABLE hive_features \
    (feature_id BIGINT, \
    feature_name STRING, \
    feature_class STRING, \
    state_alpha STRING, \
    prim_lat_dec DOUBLE, \
    prim_long_dec DOUBLE, \
    elev_in_ft BIGINT) \
    ROW FORMAT DELIMITED \
    FIELDS TERMINATED BY '|' \
    LINES TERMINATED BY '\n' \
    LOCATION 's3://amzn-s3-demo-bucket/features';")
```

You now have a populated Hive table with data from the `features.txt` file. To verify that your data is in the table, run a Spark SQL query as shown in the following example.

```
sparkSession.sql(
    "SELECT state_alpha, COUNT(*) FROM hive_features GROUP BY state_alpha;")
```

Hive

To create a Hive table with Hive, run the following command.

```
CREATE TABLE hive_features
    (feature_id             BIGINT,
    feature_name            STRING ,
    feature_class           STRING ,
    state_alpha             STRING,
    prim_lat_dec            DOUBLE ,
    prim_long_dec           DOUBLE ,
    elev_in_ft              BIGINT)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '|'
    LINES TERMINATED BY '\n'
    LOCATION 's3://amzn-s3-demo-bucket/features';
```

You now have a Hive table that contains data from the `features.txt` file. To verify that your data is in the table, run a HiveQL query, as shown in the following example.

```
SELECT state_alpha, COUNT(*) FROM hive_features GROUP BY state_alpha;
```

Step 3: Copy data to DynamoDB

Use Spark or Hive to copy data to a new DynamoDB table.

Spark

To copy data from the Hive table that you created in the previous step to DynamoDB, follow **Steps 1-3** in Copy data to DynamoDB. This creates a new DynamoDB table called `Features`. You can then read data directly from the text file and copy it to your DynamoDB table, as the following example shows.

```
import com.amazonaws.services.dynamodbv2.model.AttributeValue
import org.apache.hadoop.dynamodb.DynamoDBItemWritable
import org.apache.hadoop.dynamodb.read.DynamoDBInputFormat
import org.apache.hadoop.io.Text
import org.apache.hadoop.mapred.JobConf
import org.apache.spark.SparkContext

import scala.collection.JavaConverters._
```

```
object EmrServerlessDynamoDbTest {

    def main(args: Array[String]): Unit = {

        jobConf.set("dynamodb.input.tableName", "Features")
        jobConf.set("dynamodb.output.tableName", "Features")
        jobConf.set("dynamodb.region", "region")

        jobConf.set("mapred.output.format.class",
 "org.apache.hadoop.dynamodb.write.DynamoDBOutputFormat")
        jobConf.set("mapred.input.format.class",
 "org.apache.hadoop.dynamodb.read.DynamoDBInputFormat")

        val rdd = sc.textFile("s3://amzn-s3-demo-bucket/ddb-connector/")
            .map(row => {
                val line = row.split("\\|")
                val item = new DynamoDBItemWritable()

                val elevInFt = if (line.length > 6) {
                    new AttributeValue().withN(line(6))
                } else {
                    new AttributeValue().withNULL(true)
                }

                item.setItem(Map(
                    "feature_id" -> new AttributeValue().withN(line(0)),
                    "feature_name" -> new AttributeValue(line(1)),
                    "feature_class" -> new AttributeValue(line(2)),
                    "state_alpha" -> new AttributeValue(line(3)),
                    "prim_lat_dec" -> new AttributeValue().withN(line(4)),
                    "prim_long_dec" -> new AttributeValue().withN(line(5)),
                    "elev_in_ft" -> elevInFt)
                    .asJava)
                (new Text(""), item)
        })
        rdd.saveAsHadoopDataset(jobConf)
    }
}
```

Hive

To copy data from the Hive table that you created in the previous step to DynamoDB, follow the instructions in Copy data to DynamoDB.

Step 4: Query data from DynamoDB

Use Spark or Hive to query your DynamoDB table.

Spark

To query data from the DynamoDB table that you created in the previous step, you can use either Spark SQL or the Spark MapReduce API.

Example – Query your DynamoDB table with Spark SQL

The following Spark SQL query returns a list of all the feature types in alphabetical order.

```
val dataFrame = sparkSession.sql("SELECT DISTINCT feature_class \
    FROM ddb_features \
    ORDER BY feature_class;")
```

The following Spark SQL query returns a list of all lakes that begin with the letter *M*.

```
val dataFrame = sparkSession.sql("SELECT feature_name, state_alpha \
    FROM ddb_features \
    WHERE feature_class = 'Lake' \
    AND feature_name LIKE 'M%' \
    ORDER BY feature_name;")
```

The following Spark SQL query returns a list of all states with at least three features that are higher than one mile.

```
val dataFrame = sparkSession.dql("SELECT state_alpha, feature_class, COUNT(*) \
    FROM ddb_features \
    WHERE elev_in_ft > 5280 \
    GROUP by state_alpha, feature_class \
    HAVING COUNT(*) >= 3 \
    ORDER BY state_alpha, feature_class;")
```

Example – Query your DynamoDB table with the Spark MapReduce API

The following MapReduce query returns a list of all the feature types in alphabetical order.

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
 classOf[DynamoDBItemWritable])
    .map(pair => (pair._1, pair._2.getItem))
```

```
    .map(pair => pair._2.get("feature_class").getS)
    .distinct()
    .sortBy(value => value)
    .toDF("feature_class")
```

The following MapReduce query returns a list of all lakes that begin with the letter *M*.

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
 classOf[DynamoDBItemWritable])
    .map(pair => (pair._1, pair._2.getItem))
    .filter(pair => "Lake".equals(pair._2.get("feature_class").getS))
    .filter(pair => pair._2.get("feature_name").getS.startsWith("M"))
    .map(pair => (pair._2.get("feature_name").getS,
 pair._2.get("state_alpha").getS))
    .sortBy(_._1)
    .toDF("feature_name", "state_alpha")
```

The following MapReduce query returns a list of all states with at least three features that are higher than one mile.

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
 classOf[DynamoDBItemWritable])
    .map(pair => pair._2.getItem)
    .filter(pair => pair.get("elev_in_ft").getN != null)
    .filter(pair => Integer.parseInt(pair.get("elev_in_ft").getN) > 5280)
    .groupBy(pair => (pair.get("state_alpha").getS, pair.get("feature_class").getS))
    .filter(pair => pair._2.size >= 3)
    .map(pair => (pair._1._1, pair._1._2, pair._2.size))
    .sortBy(pair => (pair._1, pair._2))
    .toDF("state_alpha", "feature_class", "count")
```

Hive

To query data from the DynamoDB table that you created in the previous step, follow the instructions in Query the data in the DynamoDB table.

Setting up cross-account access

To set up cross-account access for EMR Serverless, complete the following steps. In the example, `AccountA` is the account where you created your Amazon EMR Serverless application, and `AccountB` is the account where your Amazon DynamoDB is located.

1. Create a DynamoDB table in `AccountB`. For more information, see Step 1: Create a table.
2. Create a `Cross-Account-Role-B` IAM role in `AccountB` that can access the DynamoDB table.

   a. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.

   b. Choose **Roles**, and create a new role called `Cross-Account-Role-B`. For more information on how to create IAM roles, see Creating IAM roles in the *a user Guide*.

   c. Create an IAM policy that grants permissions to access the cross-account DynamoDB table. Then attach the IAM policy to `Cross-Account-Role-B`.

      The following is a policy that grants access to a DynamoDB table `CrossAccountTable`.

```
{"Version": "2012-10-17",
    "Statement": [
        {"Effect": "Allow",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:region:AccountB:table/
CrossAccountTable"
        }
    ]
}
```

   d. Edit the trust relationship for the `Cross-Account-Role-B` role.

      To configure the trust relationship for the role, choose the **Trust Relationships** tab in the IAM console for the role that you created in *Step 2: Cross-Account-Role-B*.

      Select **Edit Trust Relationship** and then add the following policy document. This document allows `Job-Execution-Role-A` in `AccountA` to assume this `Cross-Account-Role-B` role.

```
{"Version": "2012-10-17",
  "Statement": [
    {"Effect": "Allow",
      "Principal": {"AWS": "arn:aws:iam::AccountA:role/Job-Execution-Role-A"
      },
      "Action": "sts:AssumeRole"
    }
  ]
```

```
}
```

e. Grant `Job-Execution-Role-A` in `AccountA` with `- STS Assume role` permissions to assume `Cross-Account-Role-B`.

In the IAM console for AWS account `AccountA`, select `Job-Execution-Role-A`. Add the following policy statement to the `Job-Execution-Role-A` to allow the `AssumeRole` action on the `Cross-Account-Role-B` role.

```
{"Version": "2012-10-17",
    "Statement": [
        {"Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Resource": "arn:aws:iam::AccountB:role/Cross-Account-Role-B"
        }
    ]
}
```

f. Set the `dynamodb.customAWSCredentialsProvider` property with value as `com.amazonaws.emr.AssumeRoleAWSCredentialsProvider` in core-site classification. Set the environment variable `ASSUME_ROLE_CREDENTIALS_ROLE_ARN` with the ARN value of `Cross-Account-Role-B`.

3. Run Spark or Hive job using `Job-Execution-Role-A`.

Considerations

Note these behaviors and limitations when you use the DynamoDB connector with Apache Spark or Apache Hive.

Considerations when using the DynamoDB connector with Apache Spark

- Spark SQL doesn't support the creation of a Hive table with the storage-handler option. For more information, see Specifying storage format for Hive tables in the Apache Spark documentation.
- Spark SQL doesn't support the `STORED BY` operation with storage handler. If you want to interact with a DynamoDB table through an external Hive table, use Hive to create the table first.
- To translate a query to a DynamoDB query, the DynamoDB connector uses *predicate pushdown*. Predicate pushdown filters data by a column that is mapped to the partition key of a DynamoDB

table. Predicate pushdown only operates when you use the connector with Spark SQL, and not with the MapReduce API.

Considerations when using the DynamoDB connector with Apache Hive

Tuning the maximum number of mappers

- If you use the SELECT query to read data from an external Hive table that maps to DynamoDB, the number of map tasks on EMR Serverless is calculated as the total read throughput configured for the DynamoDB table, divided by the throughput per map task. The default throughput per map task is 100.
- The Hive job can use the number of map tasks beyond the maximum number of containers configured per EMR Serverless application, depending upon the read throughput configured for DynamoDB. Also, a long-running Hive query can consume all of the provisioned read capacity of the DynamoDB table. This negatively impacts other users.
- You can use the `dynamodb.max.map.tasks` property to set an upper limit for map tasks. You can also use this property to tune the amount of data read by each map task based on the task container size.
- You can set the `dynamodb.max.map.tasks`property at Hive query level, or in the `hive-site` classification of the **start-job-run** command. This value must be equal to or greater than 1. When Hive processes your query, the resulting Hive job uses no more than the values of `dynamodb.max.map.tasks` when it reads from the DynamoDB table.

Tuning the write throughput per task

- Write throughput per task on EMR Serverless is calculated as the total write throughput that is configured for a DynamoDB table, divided by the value of the `mapreduce.job.maps` property. For Hive, the default value of this property is 2. Thusthe first two tasks in the final stage of Hive job can consume all of the write throughput . This leads to throttling of writes of other tasks in the same job or other jobs.
- To avoid write throttling, you can set the value of `mapreduce.job.maps` property based on the number of tasks in the final stage or the write throughput that you want to allocate per task. Set this property in the `mapred-site` classification of the **start-job-run** command on EMR Serverless.

Security

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon EMR Serverless, see AWS services in scope by compliance program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon EMR Serverless. The topics in this chapter show you how to configure Amazon EMR Serverless and use other AWS services to meet your security and compliance objectives.

Topics

- Infrastructure security in Amazon EMR Serverless
- Configuration and vulnerability analysis in Amazon EMR Serverless

Security best practices for Amazon EMR Serverless

Amazon EMR Serverless provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Apply principle of least privilege

EMR Serverless provides a granular access policy for applications using IAM roles, such as execution roles. We recommend that execution roles be granted only the minimum set of privileges required by the job, such as covering your application and access to log destination. We also recommend auditing the jobs for permissions on a regular basis and upon any change to application code.

Isolate untrusted application code

EMR Serverless creates full network isolation between jobs belonging to different EMR Serverless applications. In cases where job-level isolation is desired, consider isolating jobs into different EMR Serverless applications.

Role-based access control (RBAC) permissions

Administrators should strictly control Role-based access control (RBAC) permissions for EMR Serverless applcations.

Data protection

The AWS shared responsibility model applies to data protection in Amazon EMR Serverless. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the AWS Security Blog.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- Use Amazon EMR Serverless encryption options to encrypt data at rest and in transit.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Amazon EMR Serverless or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Amazon EMR Serverless or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

Encryption at rest

Data encryption helps prevent unauthorized users from reading data on a cluster and associated data storage systems. This includes data saved to persistent media, known as data at rest, and data that may be intercepted as it travels the network, known as data in transit.

Data encryption requires keys and certificates. You can choose from several options, including keys managed by AWS Key Management Service, keys managed by Amazon S3, and keys and certificates from custom providers that you supply. When using AWS KMS as your key provider, charges apply for the storage and use of encryption keys. For more information, see AWS KMS pricing.

Before you specify encryption options, decide on the key and certificate management systems you want to use. Then create the keys and certificates for the custom providers that you specify as part of encryption settings.

Encryption at rest for EMRFS data in Amazon S3

Each EMR Serverless application uses a specific release version, which includes EMRFS (EMR File System). Amazon S3 encryption works with EMR File System (EMRFS) objects read from and written to Amazon S3. You can specify Amazon S3 server-side encryption (SSE) or client-side encryption (CSE) as the **Default encryption mode** when you enable encryption at rest. Optionally, you can specify different encryption methods for individual buckets using **Per bucket encryption overrides**. Regardless of whether Amazon S3 encryption is enabled, Transport Layer Security (TLS) encrypts the EMRFS objects in transit between EMR cluster nodes and Amazon S3. If you use Amazon S3 CSE with customer-managed keys, your execution role used to run jobs in an EMR Serverless application must have access to the key. For in-depth information about Amazon S3 encryption, see Protecting data using encryption in the Amazon Simple Storage Service Developer Guide.

> **Note**
>
> When you use AWS KMS, charges apply for the storage and use of encryption keys. For more information, see AWS KMS pricing.

Amazon S3 server-side encryption

When you set up Amazon S3 server-side encryption, Amazon S3 encrypts data at the object level as it writes the data to disk and decrypts the data when it is accessed. For more information about SSE, see Protecting data using server-side encryption in the Amazon Simple Storage Service Developer Guide.

You can choose between two different key management systems when you specify SSE in Amazon EMR Serverless:

- **SSE-S3** - Amazon S3 manages keys for you. No additional setup is required on EMR Serverless.
- **SSE-KMS** - You use an AWS KMS key to set up with policies suitable for EMR Serverless. No additional setup is required on EMR Serverless.

To use AWS KMS encryption for data that you write to Amazon S3, you have two options when you use the `StartJobRun` API. You can either enable encrytion for everything that you write to Amazon S3, or you can enable encryption for data that you write to a specific bucket. For more information about the `StartJobRun` API, see the EMR Serverless API Reference.

To turn on AWS KMS encryption for all data that you write to Amazon S3, use the following commands when you call the `StartJobRun` API.

```
--conf spark.hadoop.fs.s3.enableServerSideEncryption=true
--conf spark.hadoop.fs.s3.serverSideEncryption.kms.keyId=<kms_id>
```

To turn on AWS KMS encryption for data that you write to a specific bucket, use the following commands when you call the `StartJobRun` API.

```
--conf spark.hadoop.fs.s3.bucket.<amzn-s3-demo-bucket1>.enableServerSideEncryption=true
--conf spark.hadoop.fs.s3.bucket.<amzn-s3-demo-
bucket1>.serverSideEncryption.kms.keyId=<kms-id>
```

SSE with customer-provided keys (SSE-C) is not available for use with EMR Serverless.

Amazon S3 client-side encryption

With Amazon S3 client-side encryption, the Amazon S3 encryption and decryption takes place in the EMRFS client available on every Amazon EMR release. Objects are encrypted before being uploaded to Amazon S3 and decrypted after they are downloaded. The provider you specify supplies the encryption key that the client uses. The client can use keys provided by AWS KMS (CSE-KMS) or a custom Java class that provides the client-side root key (CSE-C). The encryption specifics are slightly different between CSE-KMS and CSE-C, depending on the specified provider and the metadata of the object being decrypted or encrypted. If you use Amazon S3 CSE with customer-managed keys, your execution role used to run jobs in an EMR Serverless application must have access to the key. Additional KMS charges may apply. For more information about these differences, see Protecting data using client-side encryption in the Amazon Simple Storage Service Developer Guide.

Local disk encryption

Data stored in ephemeral storage is encrypted with service owned keys using industry standard AES-256 cryptographic algorithm.

Key management

You can configure KMS to automatically rotate your KMS keys. This rotates your keys once a year while saving old keys indefinitely so that your data can still be decrypted. For additional information, see Rotating customer master keys.

Encryption in transit

The following application-specific encryption features are available with Amazon EMR Serverless:

- Spark
  - By default, communication between Spark drivers and executors is authenticated and internal. RPC communication between drivers and executors is encrypted.
- Hive
  - Communication between the AWS Glue metastore and EMR Serverless applications happens via TLS.

You should allow only encrypted connections over HTTPS (TLS) using the aws:SecureTransport condition on Amazon S3 bucket IAM policies.

Identity and Access Management (IAM) in Amazon EMR Serverless

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon EMR Serverless resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Troubleshooting Amazon EMR Serverless identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Amazon EMR Serverless.

Service user – If you use the Amazon EMR Serverless service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Amazon EMR Serverless features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Amazon EMR Serverless, see Troubleshooting Amazon EMR Serverless identity and access.

Service administrator – If you're in charge of Amazon EMR Serverless resources at your company, you probably have full access to Amazon EMR Serverless. It's your job to determine which Amazon EMR Serverless features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Amazon EMR Serverless, see Identity and Access Management (IAM) in Amazon EMR Serverless.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Amazon EMR Serverless. To view example Amazon EMR Serverless identity-based policies that you can use in IAM, see Sample identity-based policies for EMR Serverless.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For

information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – 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. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific

resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached

to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How EMR Serverless works with IAM

Before you use IAM to manage access to Amazon EMR Serverless, learn what IAM features are available to use with Amazon EMR Serverless.

IAM features you can use with EMR Serverless

IAM feature	Amazon EMR Serverless support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	No
ACLs	No
ABAC (tags in policies)	Yes

IAM feature	Amazon EMR Serverless support
Temporary credentials	Yes
Principal permissions	Yes
Service roles	No
Service-linked roles	Yes

To get a high-level view of how EMR Serverless and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for EMR Serverless

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Sample identity-based policies for EMR Serverless

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless.

Resource-based policies within EMR Serverless

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that

support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for EMR Serverless

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of EMR Serverless actions, see Actions, resources, and condition keys for Amazon EMR Serverless in the *Service Authorization Reference*.

Policy actions in EMR Serverless use the following prefix before the action.

```
emr-serverless
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "emr-serverless:action1",
      "emr-serverless:action2"
         ]
```

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless.

Policy resources for EMR Serverless

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Amazon EMR Serverless resource types and their ARNs, see Resources defined by Amazon EMR Serverless in the *Service Authorization Reference*. To learn which actions you can specify the ARN of each resource, see Actions, resources, and condition keys for Amazon EMR Serverless.

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless.

Policy condition keys for EMR Serverless

Supports service-specific policy condition keys	No

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Amazon EMR Serverless condition keys and to learn which actions and resources you can use a condition key, see Actions, resources, and condition keys for Amazon EMR Serverless in the *Service Authorization Reference*.

All Amazon EC2 actions support the `aws:RequestedRegion` and `ec2:Region` condition keys. For more information, see Example: Restricting access to a specific region.

Access control lists (ACLs) in EMR Serverless

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with EMR Serverless

Supports ABAC (tags in policies)	Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using Temporary credentials with EMR Serverless

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for EMR Serverless

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for EMR Serverless

Supports service roles	No

Service-linked roles for EMR Serverless

Supports service-linked roles	Yes

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Using service-linked roles for EMR Serverless

Amazon EMR Serverless uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to EMR Serverless. Service-linked roles are predefined by EMR Serverless and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up EMR Serverless easier because you don't have to manually add the necessary permissions. EMR Serverless defines the permissions of its service-linked roles, and unless defined otherwise, only EMR Serverless can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your EMR Serverless resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for EMR Serverless

EMR Serverless uses the service-linked role named **AWSServiceRoleForAmazonEMRServerless** to enable it to call AWS APIs on your behalf.

The AWSServiceRoleForAmazonEMRServerless service-linked role trusts the following services to assume the role:

- `ops.emr-serverless.amazonaws.com`

The role permissions policy named `AmazonEMRServerlessServiceRolePolicy` allows EMR Serverless to complete the following actions on the specified resources.

> **Note**
>
> Managed policy contents change, so the policy shown here might be out of date. View the most up-to-date policy AmazonEMRServerlessServiceRolePolicy in the AWS Management Console.

- Action: `ec2:CreateNetworkInterface`
- Action: `ec2:DeleteNetworkInterface`
- Action: `ec2:DescribeNetworkInterfaces`
- Action: `ec2:DescribeSecurityGroups`
- Action: `ec2:DescribeSubnets`
- Action: `ec2:DescribeVpcs`
- Action: `ec2:DescribeDhcpOptions`
- Action: `ec2:DescribeRouteTables`
- Action: `cloudwatch:PutMetricData`

The following is the full `AmazonEMRServerlessServiceRolePolicy` policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EC2PolicyStatement",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs",
                "ec2:DescribeDhcpOptions",
                "ec2:DescribeRouteTables"
            ],
            "Resource": "*"
        },
        {
            "Sid": "CloudWatchPolicyStatement",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": [
                "*"
            ],
            "Condition": {
                "StringEquals": {
                    "cloudwatch:namespace": [
                        "AWS/EMRServerless",
                        "AWS/Usage"
                    ]
                }
            }
        }
    ]
}
```

The following trust policy is attached to this role to allow the EMR Serverless principal to assume this role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "ops.emr-serverless.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for EMR Serverless

You don't need to manually create a service-linked role. When you create a new EMR Serverless application in the AWS Management Console (using EMR Studio), the AWS CLI, or the AWS API, EMR Serverless creates the service-linked role for you. You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role.

To create the AWSServiceRoleForAmazonEMRServerless service-linked role using IAM

Add the following statement to the permissions policy for the IAM entity that needs to create the service-linked role.

```
{
    "Effect": "Allow",
    "Action": [
        "iam:CreateServiceLinkedRole"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/
AWSServiceRoleForAmazonEMRServerless*",
    "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-
serverless.amazonaws.com"}}
}
```

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create a new EMR Serverless application, EMR Serverless creates the service-linked role for you again.

You can also use the IAM console to create a service-linked role with the **EMR Serverless** use case. In the AWS CLI or the AWS API, create a service-linked role with the `ops.emr-serverless.amazonaws.com` service name. For more information, see Creating a service-linked role in the *IAM User Guide*. If you delete this service-linked role, you can use this same process to create the role again.

Editing a service-linked role for EMR Serverless

EMR Serverless does not allow you to edit the AWSServiceRoleForAmazonEMRServerless service-linked role because various entities might reference the role. You can't edit the AWS-owned IAM policy that the EMR Serverless service-linked role uses, as it contains all the necessary permissions EMR Serverless needs. However, you can edit the description of the role using IAM.

To edit the description of the AWSServiceRoleForAmazonEMRServerless service-linked role using IAM

Add the following statement to the permissions policy for the IAM entity that needs to edit the description of a service-linked role.

```
{
    "Effect": "Allow",
    "Action": [
        "iam: UpdateRoleDescription"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/
AWSServiceRoleForAmazonEMRServerless*",
    "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-
serverless.amazonaws.com"}}
}
```

For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for EMR Serverless

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. This is so you don’t have an unused entity that is not actively monitored

or maintained. However, you must delete all EMR Serverless applications in all Regions before you can delete the service-linked role.

> **Note**
>
> If the EMR Serverless service is using the role when you try to delete the resources associated with the role, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To delete the AWSServiceRoleForAmazonEMRServerless service-linked role using IAM

Add the following statement to the permissions policy for the IAM entity that needs to delete a service-linked role.

```
{
    "Effect": "Allow",
    "Action": [
        "iam:DeleteServiceLinkedRole",
        "iam:GetServiceLinkedRoleDeletionStatus"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/
AWSServiceRoleForAmazonEMRServerless*",
    "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-
serverless.amazonaws.com"}}
}
```

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForAmazonEMRServerless service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Supported Regions for EMR Serverless service-linked roles

EMR Serverless supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Job runtime roles for Amazon EMR Serverless

You can specify IAM role permissions that a EMR Serverless job run can assume when calling other services on your behalf. This includes access to Amazon S3 for any data sources, targets, as well as other AWS resources like Amazon Redshift clusters and DynamoDB tables. To learn more about how to create a role, see Create a job runtime role.

Sample runtime policies

You can attach a runtime policy, such as the following, to a job runtime role. The following job runtime policy allows:

- Read access to Amazon S3 buckets with EMR samples.
- Full access to S3 buckets.
- Create and read access to AWS Glue Data Catalog.

To add access to other AWS resources like DynamoDB, you'll need to include permissions for them in the policy when creating the runtime role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadAccessForEMRSamples",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*.elasticmapreduce",
                "arn:aws:s3:::*.elasticmapreduce/*"
            ]
        },
        {
            "Sid": "FullAccessToS3Bucket",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
```

```
                "s3:ListBucket",
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket",
                "arn:aws:s3:::amzn-s3-demo-bucket/*"
            ]
        },
        {
            "Sid": "GlueCreateAndReadDataCatalog",
            "Effect": "Allow",
            "Action": [
                "glue:GetDatabase",
                "glue:CreateDatabase",
                "glue:GetDataBases",
                "glue:CreateTable",
                "glue:GetTable",
                "glue:UpdateTable",
                "glue:DeleteTable",
                "glue:GetTables",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:GetUserDefinedFunctions"
            ],
            "Resource": ["*"]
        }
    ]
}
```

Pass role privileges

You can attach IAM permissions policies to the a user's role to allow the user to pass only approved roles. This allows administrators to control which users can pass specific job runtime roles to EMR Serverless jobs. To learn more about setting permissions, see Granting a user permissions to pass a role to an AWS service.

The following is an example policy that allows passing a job runtime role to the EMR Serverless service principal.

```
{
     "Effect": "Allow",
```

```
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::1234567890:role/JobRuntimeRoleForEMRServerless",
         "Condition": {
                 "StringLike": {
                     "iam:PassedToService": "emr-serverless.amazonaws.com"
                 }
             }
}
```

Managed permission policies associated with runtime roles

When you submit job runs to EMR serverless through the EMR Studio console, there is a step where you choose a **Runtime role** to associate with your application. There are underlying managed policies associated with each selection in the console that are important to be aware of. The three selections are the following:

1. **All buckets** – When you choose this, it specifies the AmazonS3FullAccess AWS managed policy, which provides full access to all buckets.
2. **Specific buckets** – This specifies the Amazon resource name (ARN) identifier of each bucket that you choose. There isn't an underlying managed policy included.
3. **None** – No managed-policy permissions are included.

We recommend adding specific buckets. If you choose all buckets, keep in mind that it sets full access for all buckets.

User access policy examples for EMR Serverless

You can set up fine-grained policies for your users depending on the actions you want each user to perform when interacting with EMR Serverless applications. The following policies are examples that might help in setting up the right permissions for your users. This section focuses only on EMR Serverless policies. For samples of EMR Studio user policies, see Configure EMR Studio user permissions. For information about how to attach policies to IAM users (principals), see Managing IAM policies in the IAM User Guide.

Power user policy

To grant all the required actions for EMR Serverless, create and attach a `AmazonEMRServerlessFullAccess` policy to the required IAM user, role, or group.

The following is a sample policy that allows power users to create and modify EMR Serverless applications, as well as perform other actions like submitting and debugging jobs. It reveals all the actions that EMR Serverless requires for other services.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRServerlessActions",
            "Effect": "Allow",
            "Action": [
                "emr-serverless:CreateApplication",
                "emr-serverless:UpdateApplication",
                "emr-serverless:DeleteApplication",
                "emr-serverless:ListApplications",
                "emr-serverless:GetApplication",
                "emr-serverless:StartApplication",
                "emr-serverless:StopApplication",
                "emr-serverless:StartJobRun",
                "emr-serverless:CancelJobRun",
                "emr-serverless:ListJobRuns",
                "emr-serverless:GetJobRun"
            ],
            "Resource": "*"
        }
    ]
}
```

When you enable network connectivity to your VPC, EMR Serverless applications create Amazon EC2 elastic network interfaces (ENIs) to communicate with VPC resources. The following policy ensures that any new EC2 ENIs are only created in the context of EMR Serverless applications.

> **Note**
>
> We strongly recommend setting this policy to ensure that users cannot create EC2 ENIs except in the context of launching EMR Serverless applications.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Sid": "AllowEC2ENICreationWithEMRTags",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:network-interface/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:CalledViaLast": "ops.emr-serverless.amazonaws.com"
                }
            }
        }
}
```

If you want to restrict EMR Serverless access to certain subnets, you can tag each subnet with a tag condition. This IAM policy ensures that EMR Serverless applications can only create EC2 ENIs within allowed subnets.

```
{
    "Sid": "AllowEC2ENICreationInSubnetAndSecurityGroupWithEMRTags",
    "Effect": "Allow",
    "Action": [
        "ec2:CreateNetworkInterface"
    ],
    "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:ResourceTag/KEY": "VALUE"
        }
    }
}
```

> **Important**
>
> If you're an Administrator or power user creating your first application, you must configure your permission policies to allow you to create a EMR Serverless service-linked role. To learn more, see Using service-linked roles for EMR Serverless.

The following IAM policy permits you to create a EMR Serverless service-linked role for your account.

```
{
   "Sid":"AllowEMRServerlessServiceLinkedRoleCreation",
   "Effect":"Allow",
   "Action":"iam:CreateServiceLinkedRole",
   "Resource":"arn:aws:iam::account-id:role/aws-service-role/ops.emr-
serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless"
}
```

Data engineer policy

This following is a sample policy that allows users read-only permissions on EMR Serverless applications, as well as the the ability to submit and debug jobs. Keep in mind that because this policy does not explicitly deny actions, a different policy statement may still be used to grant access to specified actions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRServerlessActions",
            "Effect": "Allow",
            "Action": [
                "emr-serverless:ListApplications",
                "emr-serverless:GetApplication",
                "emr-serverless:StartApplication",
                "emr-serverless:StartJobRun",
                "emr-serverless:CancelJobRun",
                "emr-serverless:ListJobRuns",
                "emr-serverless:GetJobRun"
            ],
            "Resource": "*"
```

```
        }
    ]
}
```

Using tags for access control

You can use tag conditions for fine-grained access control. For example, you can restrict users from one team such that they're only able to submit jobs to EMR Serverless applications tagged with their team name.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRServerlessActions",
            "Effect": "Allow",
            "Action": [
                "emr-serverless:ListApplications",
                "emr-serverless:GetApplication",
                "emr-serverless:StartApplication",
                "emr-serverless:StartJobRun",
                "emr-serverless:CancelJobRun",
                "emr-serverless:ListJobRuns",
                "emr-serverless:GetJobRun"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Team": "team-name"
                }
            }
        }
    ]
}
```

Policies for tag-based access control

You can use conditions in your identity-based policy to control access to applications and job runs based on tags.

The following examples demonstrate different scenarios and ways to use condition operators with EMR Serverless condition keys. These IAM policy statements are intended for demonstration

purposes only and should not be used in production environments. There are multiple ways to combine policy statements to grant and deny permissions according to your requirements. For more information about planning and testing IAM policies, see the IAM user Guide.

> **Important**
>
> Explicitly denying permission for tagging actions is an important consideration. This prevents users from tagging a resource and thereby granting themselves permissions that you did not intend to grant. If tagging actions for a resource are not denied, a user can modify tags and circumvent the intention of the tag-based policies. For an example of a policy that denies tagging actions, see Deny access to add and remove tags.

The examples below demonstrate identity-based permissions policies that are used to control the actions that are allowed with EMR Serverless applications.

Allow actions only on resources with specific tag values

In the following policy example, the `StringEquals` condition operator tries to match `dev` with the value for the tag department. If the tag department hasn't been added to the application, or doesn't contain the value `dev`, the policy doesn't apply, and the actions aren't allowed by this policy. If no other policy statements allow the actions, the user can only work with applications that have this tag with this value.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:GetApplication"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "emr-serverless:ResourceTag/department": "dev"
        }
      }
    }
  ]
```

```
}
```

You can also specify multiple tag values using a condition operator. For example, to allow actions on applications where the `department` tag contains the value `dev` or `test`, you could replace the condition block in the earlier example with the following.

```
"Condition": {
        "StringEquals": {
          "emr-serverless:ResourceTag/department": ["dev", "test"]
        }
      }
```

Require tagging when a resource is created

In the example below, the tag needs to be applied when creating the application.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CreateApplication"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "emr-serverless:RequestTag/department": "dev"
        }
      }
    }
  ]
}
```

The following policy statement allows a user to create an application only if the application has a `department` tag, which can contain any value.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Action": [
        "emr-serverless:CreateApplication"
      ],
      "Resource": "*",
      "Condition": {
        "Null": {
          "emr-serverless:RequestTag/department": "false"
        }
      }
    }
  ]
}
```

Deny access to add and remove tags

This policy prevents a user from adding or removing tags on EMR Serverless applications with a `department` tag whose value is not `dev`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "emr-serverless:TagResource",
        "emr-serverless:UntagResource"
      ],
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "emr-serverless:ResourceTag/department": "dev"
        }
      }
    }
  ]
}
```

Identity-based policy examples for EMR Serverless

By default, users and roles don't have permission to create or modify Amazon EMR Serverless resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the

resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Amazon EMR Serverless, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon EMR Serverless in the *Service Authorization Reference*.

Topics

Policy best practices

> **Note**
>
> EMR Serverless doesn't support managed policies, so the first practice listed below doesn't apply.

Identity-based policies determine whether someone can create, access, or delete Amazon EMR Serverless resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
```

```
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon EMR Serverless updates to AWS managed policies

View details about updates to AWS managed policies for Amazon EMR Serverless since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Amazon EMR Serverless Document history page.

Change	Description	Date
AmazonEMRServerlessServiceRolePolicy – Update to an existing policy	Amazon EMR Serverless added the new `Sid` `CloudWatchPolicyStatement` and `EC2PolicyStatement` to the AmazonEMRServerlessServiceRolePolicy policy.	January 25, 2024

Change	Description	Date
AmazonEMRServerless sServiceRolePolicy – Update to an existing policy	Amazon EMR Serverless added new permissions to allow Amazon EMR Serverless to publish aggregated account metrics for vCPU usage in the `"AWS/Usage"` namespace.	April 20, 2023
Amazon EMR Serverless started tracking changes	Amazon EMR Serverless started tracking changes for its AWS managed policies.	April 20, 2023

Troubleshooting Amazon EMR Serverless identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon EMR Serverless and IAM.

Topics

- I am not authorized to perform an action in Amazon EMR Serverless
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Amazon EMR Serverless resources

I am not authorized to perform an action in Amazon EMR Serverless

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` user tries to use the console to view details about a fictional *`my-example-widget`* resource but does not have the fictional `emr-serverless:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: emr-
serverless:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *my-example-widget* resource using the `emr-serverless:GetWidget` action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon EMR Serverless.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon EMR Serverless. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Amazon EMR Serverless resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon EMR Serverless supports these features, see Identity and Access Management (IAM) in Amazon EMR Serverless.

- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Using EMR Serverless with AWS Lake Formation for fine-grained access control

Overview

With Amazon EMR releases 7.2.0 and higher, you can leverage AWS Lake Formation to apply fine-grained access controls on Data Catalog tables that are backed by S3. This capability lets you configure table, row, column, and cell level access controls for read queries within your Amazon EMR Serverless Spark jobs. To configure fine-grained access control for Apache Spark batch jobs and interactive sessions, use EMR Studio. See the following sections to learn more about Lake Formation and how to use it with EMR Serverless.

Using Amazon EMR Serverless with AWS Lake Formation incurs additional charges. For more information, see Amazon EMR pricing.

How EMR Serverless works with AWS Lake Formation

Using EMR Serverless with Lake Formation lets you enforce a layer of permissions on each Spark job to apply Lake Formation permissions control when EMR Serverless executes jobs. EMR Serverless uses Spark resource profiles to create two profiles to effectively execute jobs. The user profile executes user-supplied code, while the system profile enforces Lake Formation policies. For more information, see What is AWS Lake Formation and Considerations and limitations.

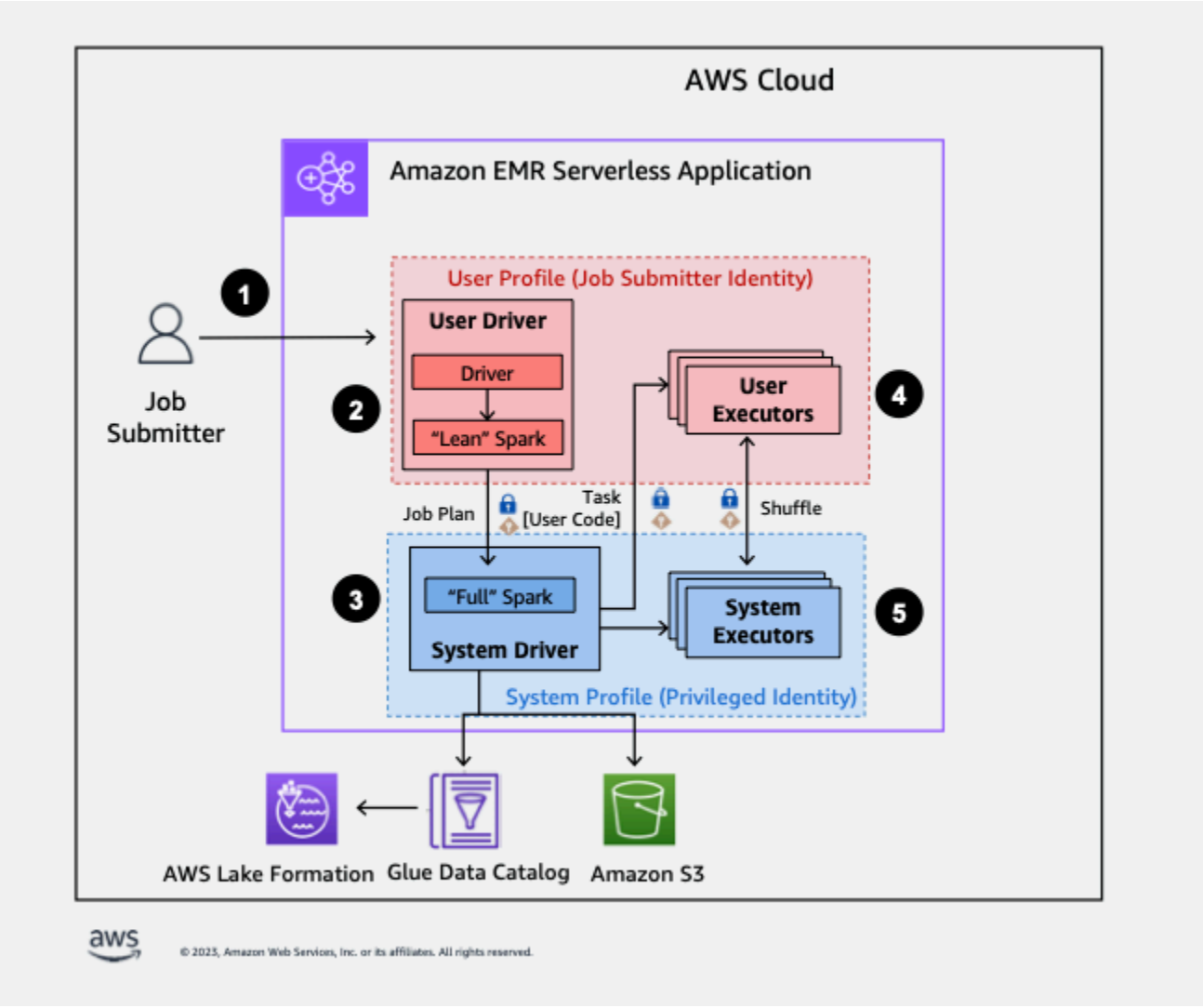
When you use pre-initialized capacity with Lake Formation, we recommend that you have a minimum of two Spark drivers. Each Lake Formation-enabled job utilizes two Spark drivers, one for the user profile and one for the system profile. For the best performance, you should use double the number of drivers for Lake Formation-enabled jobs compared to if you don't use Lake Formation.

When you run Spark jobs on EMR Serverless, you must also consider the impact of dynamic allocation on resource management and cluster performance. The configuration `spark.dynamicAllocation.maxExecutors` of the maximum number of executors per resource profile applies to both user and system executors. If you configure that number to be equal to the maximum allowed number of executors, your job run might get stuck because of one type of executor that uses all available resources, which prevents the other executor when you run jobs jobs.

So you don't run out of resources, EMR Serverless sets the default maximum number of executors per resource profile to 90% of the `spark.dynamicAllocation.maxExecutors` value. You can override this configuration when you specify `spark.dynamicAllocation.maxExecutorsRatio` with a value between 0 and 1. Additionally, you can also configure the following properties to optimize resource allocation and overall performance:

- `spark.dynamicAllocation.cachedExecutorIdleTimeout`
- `spark.dynamicAllocation.shuffleTracking.timeout`
- `spark.cleaner.periodicGC.interval`

The following is a high-level overview of how EMR Serverless gets access to data protected by Lake Formation security policies.



1. A user submits Spark job to an AWS Lake Formation-enabled EMR Serverless application.
2. EMR Serverless sends the job to a user driver and runs the job in the user profile. The user driver runs a lean version of Spark that has no ability to launch tasks, request executors, access S3 or the Glue Catalog. It builds a job plan.
3. EMR Serverless sets up a second driver called the system driver and runs it in the system profile (with a privileged identity). EMR Serverless sets up an encrypted TLS channel between the two drivers for communication. The user driver uses the channel to send the job plans to the system driver. The system driver does not run user-submitted code. It runs full Spark and communicates with S3, and the Data Catalog for data access. It request executors and compiles the Job Plan into a sequence of execution stages.
4. EMR Serverless then runs the stages on executors with the user driver or system driver. User code in any stage is run exclusively on user profile executors.

5. Stages that read data from Data Catalog tables protected by AWS Lake Formation or those that apply security filters are delegated to system executors.

Enabling Lake Formation in Amazon EMR

To enable Lake Formation, you must set `spark.emr-serverless.lakeformation.enabled` to `true` under `spark-defaults` classification for the runtime-configuration parameter when creating an EMR Serverless application.

```
aws emr-serverless create-application \
    --release-label emr-7.6.0 \
    --runtime-configuration '{
     "classification": "spark-defaults",
     "properties": {
      "spark.emr-serverless.lakeformation.enabled": "true"
      }
    }' \
    --type "SPARK"
```

You can also enable Lake Formation when you create a new application in EMR Studio. Choose **Use Lake Formation for fine-grained access control**, available under **Additional configurations**.

Inter-worker encryption is enabled by default when you use Lake Formation with EMR Serverless, so you don't have to explicitly enable inter-worker encryption again.

Enabling Lake Formation for Spark jobs

To enable Lake Formation for individual Spark jobs, set `spark.emr-serverless.lakeformation.enabled` to true when using `spark-submit`.

```
--conf spark.emr-serverless.lakeformation.enabled=true
```

Job runtime role IAM permissions

Lake Formation permissions control access to AWS Glue Data Catalog resources, Amazon S3 locations, and the underlying data at those locations. IAM permissions control access to the Lake Formation and AWS Glue APIs and resources. Although you might have the Lake Formation permission to access a table in the Data Catalog (SELECT), your operation fails if you don’t have the IAM permission on the `glue:Get*` API operation.

The following is an example policy of how to provide IAM permissions to access a script in S3, uploading logs to S3, AWS Glue API permissions, and permission to access Lake Formation.

```
{
"Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ScriptAccess",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
            "arn:aws:s3:::*.amzn-s3-demo-bucket/scripts",
            "arn:aws:s3:::*.amzn-s3-demo-bucket/*" ]
        },
        {
         "Sid": "LoggingAccess",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": [
               "arn:aws:s3:::amzn-s3-demo-bucket/logs/*"
            ]
        },
        {
            "Sid": "GlueCatalogAccess",
            "Effect": "Allow",
            "Action": [
                 "glue:Get*",
                 "glue:Create*",
                 "glue:Update*"
            ],
            "Resource": ["*"]
        },
        {
            "Sid": "LakeFormationAccess",
            "Effect": "Allow",
            "Action": [
             "lakeformation:GetDataAccess"
             ],
```

```
            "Resource": ["*"]
        }
    ]
}
```

Setting up Lake Formation permissions for job runtime role

First, register the location of your Hive table with Lake Formation. Then create permissions for your job runtime role on your desired table. For more details about Lake Formation, see What is AWS Lake Formation? in the *AWS Lake Formation Developer Guide*.

After you set up the Lake Formation permissions, you can submit Spark jobs on Amazon EMR Serverless. For more information about Spark jobs, see Spark examples.

Submitting a job run

After you finish setting up the Lake Formation grants, you can submit Spark jobs on EMR Serverless. To run Iceberg jobs, you must provide the following `spark-submit` properties.

```
--conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog
--conf spark.sql.catalog.spark_catalog.warehouse=<S3_DATA_LOCATION>
--conf spark.sql.catalog.spark_catalog.glue.account-id=<ACCOUNT_ID>
--conf spark.sql.catalog.spark_catalog.client.region=<REGION>
--conf spark.sql.catalog.spark_catalog.glue.endpoint=https://
glue.<REGION>.amazonaws.com
```

Open-table format support

Amazon EMR release 7.2.0 includes support for fine-grained access control based on Lake Formation. EMR Serverless supports Hive and Iceberg table types. The following table describes all of the supported operations.

Operations	Hive	Iceberg
DDL commands	With IAM role permissions only	With IAM role permissions only
Incremental queries	Not applicable	Fully supported

Operations	Hive	Iceberg
Time travel queries	Not applicable to this table format	Fully supported
Metadata tables	Not applicable to this table format	Supported, but certain tables are hidden. See considerations and limitations for more information.
`DML INSERT`	With IAM permissions only	With IAM permissions only
DML UPDATE	Not applicable to this table format	With IAM permissions only
`DML DELETE`	Not applicable to this table format	With IAM permissions only
Read operations	Fully supported	Fully supported
Stored procedures	Not applicable	Supported with the exceptions of `register_table` and `migrate`. See considerations and limitations for more information.

Debugging jobs

> **Note**
>
> With this feature, you can view **stdout** and **stderr** logs for the system profile workers that may contain sensitive, unfiltered information. The following permission should be used only for accessing non-production data. For applications created for use with production jobs, we strongly recommend that you add these permissions only to administrators or users with elevated data access.

With EMR-7.3.0 and later, EMR Serverless is enabling self-debugging capability for Lake Formation-enabled batch jobs. To do so, use the new parameter **accessSystemProfileLogs** in the GetDashboardForJobRun API. If **accessSystemProfileLogs** is set to **true**, you can view the **stdout** and **stderr** logs for the system profile workers, which can be used for debugging a Lake Formation-enabled EMR Serverless batch job.

```
aws emr-serverless get-dashboard-for-job-run \
  --application-id application-id
  --job-run-id job-run-id
  --access-system-profile-logs
```

Required permissions

The principal who wants to debug Lake Formation-enabled batch jobs using **GetDashboardForJobRun** must have the following additional permissions:

```
{
    "Sid": "AccessSystemProfileLogs",
    "Effect": "Allow",
    "Action": [
        "emr-serverless:GetDashboardForJobRun",
        "emr-serverless:AccessSystemProfileLogs",
        "glue:GetDatabases",
        "glue:SearchTables"
    ],
    "Resource": [
        "arn:aws:emr-serverless:region:account-id:/applications/applicationId/
jobruns/jobid",
        "arn:aws:glue:region:account-id:catalog",
        "arn:aws:glue:region:account-id:database/*",
        "arn:aws:glue:region:account-id:table/*/*"
    ]
}
```

Considerations

System profile logs for debugging are visible for jobs that access databases or tables in Lake Formation within the same account as the job. They are not visible in the following scenarios:

- If the data catalog managed using Lake Formation permissions has cross-account databases and tables

- If the data catalog managed using Lake Formation permissions has resource links

Considerations and limitations

Consider the following considerations and limitations when you use Lake Formation with EMR Serverless.

> **Note**
>
> When you enable Lake Formation for a Spark job on EMR Serverless, the job launches a system driver and a user driver. If you specified pre-initialized capacity at launch, the drivers provision from the pre-initialized capacity, and the number of system drivers is equal to the number of user drivers that you specify. If you choose On Demand capacity, EMR Serverless launches a system driver in addition to a user driver. To estimate the costs associated with your EMR Serverless with Lake Formation job, use the AWS Pricing Calculator.

Amazon EMR Serverless with Lake Formation is available in all supported EMR Serverless Regions except AWS GovCloud (US-East) and AWS GovCloud (US-West).

- Amazon EMR Serverless supports fine-grained access control via Lake Formation only for Apache Hive and Apache Iceberg tables. Apache Hive formats include Parquet, ORC, and xSV.
- Lake Formation-enabled applications don't support usage of customized EMR Serverless images.
- You can't turn off `DynamicResourceAllocation` for Lake Formation jobs.
- You can only use Lake Formation with Spark jobs.
- EMR Serverless with Lake Formation only supports a single Spark session throughout a job.
- EMR Serverless with Lake Formation only supports cross-account table queries shared through resource links.
- The following aren't supported:
  - Resilient distributed datasets (RDD)
  - Spark streaming
  - Write with Lake Formation granted permissions
  - Access control for nested columns
- EMR Serverless blocks functionalities that might undermine the complete isolation of system driver, including the following:

- UDTs, HiveUDFs, and any user-defined function that involves custom classes
- Custom data sources
- Supply of additional jars for Spark extension, connector, or metastore
- `ANALYZE TABLE` command

- To enforce access controls, `EXPLAIN PLAN` and DDL operations such as `DESCRIBE TABLE` don't expose restricted information.
- EMR Serverless restricts access to system driver Spark logs on Lake Formation-enabled applications. Since the system driver runs with more access, events and logs that the system driver generates can include sensitive information. To prevent unauthorized users or code from accessing this sensitive data, EMR Serverless disabled access to system driver logs. For troubleshooting, contact AWS support.
- If you registered a table location with Lake Formation, the data access path goes through the Lake Formation stored credentials regardless of the IAM permission for the EMR Serverless job runtime role. If you misconfigure the role registered with table location, jobs submitted that use the role with S3 IAM permission to the table location will fail.
- Writing to a Lake Formation table uses IAM permission rather than Lake Formation granted permissions. If your job runtime role has the necessary S3 permissions, you can use it to run write operations.

The following are considerations and limitations when using Apache Iceberg:

- You can only use Apache Iceberg with session catalog and not arbitrarily named catalogs.
- Iceberg tables that are registered in Lake Formation only support the metadata tables `history`, `metadata_log_entries`, `snapshots`, `files`, `manifests`, and `refs`. Amazon EMR hides the columns that might have sensitive data, such as `partitions`, `path`, and `summaries`. This limitation doesn't apply to Iceberg tables that aren't registered in Lake Formation.
- Tables that you don't register in Lake Formation support all Iceberg stored procedures. The `register_table` and `migrate` procedures aren't supported for any tables.
- We recommend that you use Iceberg DataFrameWriterV2 instead of V1.

Troubleshooting

See the following sections for troubleshooting solutions.

Logging

EMR Serverless uses Spark resources profiles to split job execution. EMR Serverless uses the user profile to run the code you supplied, while the system profile enforces Lake Formation policies. You can access the logs for the tasks ran as the user profile.

For more information about debugging Lake Formation-enabled jobs, see Debugging jobs.

Live UI and Spark History Server

The Live UI and the Spark History Server have all Spark events generated from the user profile and redacted events generated from the system driver.

You can see all of the tasks from both the user and system drivers in the **Executors** tab. However, log links are available only for the user profile. Also, some information is redacted from Live UI, such as the number of output records.

Job failed with insufficient Lake Formation permissions

Make sure that your job runtime role has the permissions to run SELECT and DESCRIBE on the table that you are accessing.

Job with RDD execution failed

EMR Serverless currently doesn't support resilient distributed dataset (RDD) operations on Lake Formation-enabled jobs.

Unable to access data files in Amazon S3

Make sure you have registered the location of the data lake in Lake Formation.

Security validation exception

EMR Serverless detected a security validation error. Contact AWS support for assistance.

Sharing AWS Glue Data Catalog and tables across accounts

You can share databases and tables across accounts and still use Lake Formation. For more information, see Cross-account data sharing in Lake Formation and How do I share AWS Glue Data Catalog and tables cross-account using AWS Lake Formation?.

Inter-worker encryption

With Amazon EMR versions 6.15.0 and higher, you can enable mutual-TLS encrypted communication between workers in your Spark job runs. When enabled, EMR Serverless automatically generates and distributes a unique certificate for each worker provisioned under your job runs. When these workers communicate to exchange control messages or transfer shuffle data, they establish a mutual TLS connection and use the configured certificates to verify the identity of each other. If a worker is unable to verify another certificate, the TLS handshake fails, and EMR Serverless aborts the connection between them.

If you're using Lake Formation with EMR Serverless, mutual-TLS encryption is enabled by default.

Enabling mutual-TLS encryption on EMR Serverless

To enable mutual TLS encryption on your spark application, set `spark.ssl.internode.enabled` to true when creating EMR Serverless application. If you're using the AWS console to create an EMR Serverless application, choose **Use custom settings**, then expand **Application configuration**, and enter your `runtimeConfiguration`.

```
aws emr-serverless create-application \
--release-label emr-6.15.0 \
--runtime-configuration '{
  "classification": "spark-defaults",
  "properties": {"spark.ssl.internode.enabled": "true"}
}' \
--type "SPARK"
```

If you want to enable mutual TLS encryption for individual spark job runs, set `spark.ssl.internode.enabled` to true when using `spark-submit`.

```
--conf spark.ssl.internode.enabled=true
```

Secrets Manager for data protection with EMR Serverless

AWS Secrets Manager is a secret storage service that you can use to protect database credentials, API keys, and other secret information. Then in your code, you can replace hardcoded credentials with an API call to Secrets Manager. This helps ensure that the secret can't be compromised by someone examining your code, because the secret isn't there. For an overview, see the AWS Secrets Manager User Guide.

Secrets Manager encrypts secrets using AWS Key Management Service keys. For more information, see Secret encryption and decryption in the *AWS Secrets Manager User Guide*.

You can configure Secrets Manager to automatically rotate secrets for you according to a schedule that you specify. This enables you to replace long-term secrets with short-term ones, which helps to significantly reduce the risk of compromise. For more information, see Rotate AWS Secrets Manager secrets in the *AWS Secrets Manager User Guide*.

Amazon EMR Serverless integrates with AWS Secrets Manager so that you can store your data in Secrets Manager and use the secret ID in your configurations.

How EMR Serverless uses secrets

When you store your data in Secrets Manager and use the secret ID in your configurations for EMR Serverless, you don't pass sensitive configuration data to EMR Serverless in plain text and expose it to external APIs. If you indicate that a key-value pair contains a secret ID for a secret that you stored in Secrets Manager, EMR Serverless retrieves the secret when it sends configuration data to workers for running jobs.

To indicate that a key-value pair for a configuration contains a reference to a secret stored in Secrets Manager, add the `EMR.secret@` annotation to the configuration value. For any configuration property with secret Id annotation, EMR Serverless calls Secrets Manager and resolves the secret at the time of job execution.

How to create a secret

To create a secret, follow the steps in Create an AWS Secrets Manager secret in the *AWS Secrets Manager User Guide*. In **Step 3**, choose the **Plaintext** field to enter your sensitive value.

Provide a secret in a configuration classification

The following examples show how to provide a secret in a configuration classification at `StartJobRun`. If you want to configure classifications for Secrets Manager at the application level, see Default application configuration for EMR Serverless.

In the examples, replace *`SecretName`* with the name of the secret to retrieve. Include the hyphen, followed by the six characters that Secrets Manager adds to the end of the secret ARN. For more information, see How to create a secret.

In this section

- Specify secret references - Spark
- Specify secret references - Hive

Specify secret references - Spark

Example – Specify secret references in the external Hive metastore configuration for Spark

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://amzn-s3-demo-bucket/scripts/spark-jdbc.py",
            "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
            --conf
 spark.hadoop.javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
            --conf spark.hadoop.javax.jdo.option.ConnectionUserName=connection-user-
name
            --conf
 spark.hadoop.javax.jdo.option.ConnectionPassword=EMR.secret@SecretName
            --conf spark.hadoop.javax.jdo.option.ConnectionURL=jdbc:mysql://db-host:db-
port/db-name
            --conf spark.driver.cores=2
            --conf spark.executor.memory=10G
            --conf spark.driver.memory=6G
            --conf spark.executor.cores=4"
        }
    }' \
    --configuration-overrides '{
        "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
        }
    }
}'
```

Example – Specify secret references for the external Hive metastore configuration in the `spark-defaults` classification

```
{
```

```
        "classification": "spark-defaults",
        "properties": {

 "spark.hadoop.javax.jdo.option.ConnectionDriverName":"org.mariadb.jdbc.Driver"
            "spark.hadoop.javax.jdo.option.ConnectionURL":"jdbc:mysql://db-host:db-
port/db-name"
            "spark.hadoop.javax.jdo.option.ConnectionUserName":"connection-user-name"
            "spark.hadoop.javax.jdo.option.ConnectionPassword":
 "EMR.secret@SecretName",
        }
    }
```

Specify secret references - Hive

Example – Specify secret references in the external Hive metastore configuration for Hive

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
    --job-driver '{
        "hive": {
        "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-query.ql",
        "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/scratch
                    --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/
emr-serverless-hive/hive/warehouse
                    --hiveconf javax.jdo.option.ConnectionUserName=username
                    --hiveconf
 javax.jdo.option.ConnectionPassword=EMR.secret@SecretName
                    --hiveconf
 hive.metastore.client.factory.class=org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreCli
                    --hiveconf
 javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
                    --hiveconf javax.jdo.option.ConnectionURL=jdbc:mysql://db-host:db-
port/db-name"
        }
    }' \
    --configuration-overrides '{
        "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket"
        }
    }
```

```
}'
```

Example – Specify secret references for the external Hive metastore configuration in the `hive-site` classification

```
{
    "classification": "hive-site",
    "properties": {
        "hive.metastore.client.factory.class":
 "org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClientFactory",
        "javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver",
        "javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-port/db-name",
        "javax.jdo.option.ConnectionUserName": "username",
        "javax.jdo.option.ConnectionPassword": "EMR.secret@SecretName"
    }
}
```

Grant access for EMR Serverless to retrieve the secret

To allow EMR Serverless to retrieve the secret value from Secrets Manager, add the following policy statement to your secret when you create it. You must create your secret with the customer-managed KMS key for EMR Serverless to read the secret value. For more information, see Permissions for the KMS key in the *AWS Secrets Manager User Guide*.

In the following policy, replace *`applicationId`* with the ID for your application.

Resource policy for the secret

You must include the following permissions in the resource policy for the secret in AWS Secrets Manager to allow EMR Serverless to retrieve secret values. To ensure that only a specific application can retrieve this secret, you can optionally specify the EMR Serverless application ID as a condition in the policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue",
```

```
        "secretsmanager:DescribeSecret"
      ],
      "Principal": {
        "Service": [
          "emr-serverless.amazonaws.com"
        ]
      },
      "Resource": [
        "*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceArn": "arn:aws:emr-serverless:AWS Region:aws_account_id:/
applications/applicationId"
        }
      }
    }
  ]
}
```

Create your secret with the following policy for the customer-managed AWS Key Management Service (AWS KMS) key:

Policy for customer-managed AWS KMS key

```
{
    "Sid": "Allow EMR Serverless to use the key for decrypting secrets",
    "Effect": "Allow",
    "Principal": {
        "Service": [
            "emr-serverless.amazonaws.com"
        ]
    },
    "Action": [
        "kms:Decrypt",
        "kms:DescribeKey"
    ],
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "secretsmanager.AWS Region.amazonaws.com"
        }
    }
```

```
}
```

Rotating the secret

Rotation is when you periodically update a secret. You can configure AWS Secrets Manager to automatically rotate the secret for you on a schedule that you specify. This way, you can replace long-term secrets with short-term ones. This helps to reduce the risk of compromise. EMR Serverless retrieves the secret value from an annotated configuration when the job transitions to a running state. If you or a process updates the secret value in Secrets Manager, you must submit a new job so that the job can fetch the updated value.

> **Note**
>
> Jobs that are already in a running state can't fetch an updated secret value. This might result in job failure.

Using Amazon S3 Access Grants with EMR Serverless

S3 Access Grants overview for EMR Serverless

With Amazon EMR releases 6.15.0 and higher, Amazon S3 Access Grants provide a scalable access control solution that you can use to augment access to your Amazon S3 data from EMR Serverless. If you have a complex or large permission configuration for your S3 data, you can use Access Grants to scale S3 data permissions for users, roles, and applications.

Use S3 Access Grants to augment access to Amazon S3 data beyond the permissions granted by the runtime role or the IAM roles that are attached to the identities with access to your EMR Serverless application.

For more information, see Managing access with S3 Access Grants for Amazon EMR in the *Amazon EMR Management Guide* and Managing access with S3 Access Grants in the *Amazon Simple Storage Service User Guide*.

This section describes how to launch an EMR Serverless application that uses S3 Access Grants to provide access to data in Amazon S3. For steps to use S3 Access Grants with other Amazon EMR deployments, see the following documentation:

- Using S3 Access Grants with Amazon EMR

- Using S3 Access Grants with Amazon EMR on EKS

Launch an EMR Serverless application with S3 Access Grants for data management

You can enable S3 Access Grants on EMR Serverless and launch a Spark application. When your application makes a request for S3 data, Amazon S3 provides temporary credentials that are scoped to the specific bucket, prefix, or object.

1. Set up a job execution role for your EMR Serverless application. Include the required IAM permissions that you need to run Spark jobs and use S3 Access Grants, `s3:GetDataAccess` and `s3:GetAccessGrantsInstanceForPrefix`:

```
{
    "Effect": "Allow",
    "Action": [
    "s3:GetDataAccess",
    "s3:GetAccessGrantsInstanceForPrefix"
    ],
    "Resource": [     //LIST ALL INSTANCE ARNS THAT THE ROLE IS ALLOWED TO QUERY
         "arn:aws_partition:s3:Region:account-id1:access-grants/default",
         "arn:aws_partition:s3:Region:account-id2:access-grants/default"
    ]
}
```

> **Note**
>
> If you specify IAM roles for job execution that have additional permissions to access S3 directly, then users will be able to access the data permitted by the role even if they don't have permission from S3 Access Grants.

2. Launch your EMR Serverless application with an Amazon EMR release label of 6.15.0 or higher and the `spark-defaults` classification, as the following example shows. Replace the values in *`red text`* with the appropriate values for your usage scenario.

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
```

```
        "sparkSubmit": {
            "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
            "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket1/
wordcount_output"],
            "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf
 spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g
 --conf spark.executor.instances=1"
        }
    }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "spark-defaults",
        "properties": {
          "spark.hadoop.fs.s3.s3AccessGrants.enabled": "true",
          "spark.hadoop.fs.s3.s3AccessGrants.fallbackToIAM": "false"
         }
      }]
}'
```

S3 Access Grants considerations with EMR Serverless

For important support, compatibility, and behavioral information when you use Amazon S3 Access Grants with EMR Serverless, see S3 Access Grants considerations with Amazon EMR in the *Amazon EMR Management Guide*.

Logging Amazon EMR Serverless API calls using AWS CloudTrail

Amazon EMR Serverless is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in EMR Serverless. CloudTrail captures all API calls for EMR Serverless as events. The calls captured include calls from the EMR Serverless console and code calls to the EMR Serverless API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for EMR Serverless. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to EMR Serverless, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

EMR Serverless information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in EMR Serverless, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for EMR Serverless, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

All EMR Serverless actions are logged by CloudTrail and are documented in the EMR Serverless API Reference. For example, calls to the `CreateApplication`, `StartJobRun` and `CancelJobRun` actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Understanding EMR Serverless log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateApplication` action.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE:admin",
        "arn": "arn:aws:sts::012345678910:assumed-role/Admin/admin",
        "accountId": "012345678910",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDACKCEVSQ6C2EXAMPLE",
                "arn": "arn:aws:iam::012345678910:role/Admin",
                "accountId": "012345678910",
                "userName": "Admin"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2022-06-01T23:46:52Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2022-06-01T23:49:28Z",
    "eventSource": "emr-serverless.amazonaws.com",
    "eventName": "CreateApplication",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.0",
    "userAgent": "PostmanRuntime/7.26.10",
    "requestParameters": {
        "name": "my-serverless-application",
```

```
        "releaseLabel": "emr-6.6",
        "type": "SPARK",
        "clientToken": "0a1b234c-de56-7890-1234-567890123456"
    },
    "responseElements": {
        "name": "my-serverless-application",
        "applicationId": "1234567890abcdef0",
        "arn": "arn:aws:emr-serverless:us-west-2:555555555555:/
applications/1234567890abcdef0"
    },
    "requestID": "890b8639-e51f-11e7-b038-EXAMPLE",
    "eventID": "874f89fa-70fc-4798-bc00-EXAMPLE",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "012345678910",
    "eventCategory": "Management"
}
```


Compliance validation for Amazon EMR Serverless

The security and compliance of EMR Serverless is assessed by third-party auditors as part of multiple AWS compliance programs, including the following:

- System and Organization Controls (SOC)
- Payment Card Industry Data Security Standard (PCI DSS)
- Federal Risk and Authorization Management Program (FedRAMP) Moderate
- Health Insurance Portability and Accountability Act (HIPAA)

AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using EMR Serverless is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use

of EMR Serverless is subject to compliance with standards like HIPAA, PCI, or FedRAMP Moderate, AWS provides resources to help:

- Security and Compliance Quick Start Guides that discuss architectural considerations and steps for deploying security- and compliance-focused baseline environments on AWS.
- AWS Customer Compliance Guides can help you understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- AWS Config can be used to assess how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Compliance Resources is a collection of workbooks and guides might apply to your industry and location.
- AWS Security Hub provides you with a comprehensive view of your security state within AWS and helps you check your compliance with security industry standards and best practices.
- AWS Audit Manager – this AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon EMR Serverless

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Amazon EMR Serverless offers integration with Amazon S3 through EMRFS to help support your data resiliency and backup needs.

Infrastructure security in Amazon EMR Serverless

As a managed service, Amazon EMR is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To

design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Amazon EMR through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Configuration and vulnerability analysis in Amazon EMR Serverless

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- Compliance validation for Amazon EMR Serverless
- Shared Responsibility Model
- Amazon Web Services: Overview of Security Processes (whitepaper)

Endpoints and quotas for EMR Serverless

Service endpoints

To connect programmatically to an AWS service, you use an *endpoint*. An endpoint is the URL of the entry point for an AWS web service. In addition to the standard AWS endpoints, some AWS services offer FIPS endpoints in selected Regions. The following table lists the service endpoints for EMR Serverless. For more information, see AWS service endpoints.

Region name	Region	Endpoint	Protocol
US East (Ohio)	`us-east-2` (limited to the following Availability Zones: `use2-az1`, `use2-az2`, and `use2-az3`)	`emr-serverless.us-east-2.amazonaws.com`	HTTPS

US East (N. Virginia)	`us-east-1` (limited to the following Availability Zones: `use1-az1`, `use1-az2`, `use1-az4`, `use1-az5`, and `use1-az6`)	`emr-serverless.us-east-1.amazonaws.com`
		
		`emr-serverless-fips.us-east-1.amazonaws.com`
			HTTPS

US West (N. California)	`us-west-1`	`emr-serverless.us-west-1.amazonaws.com`	HTTPS
US West (Oregon)	`us-west-2`	`emr-serverless.us-west-2.amazonaws.com`	HTTPS

Region name	Region	Endpoint	Protocol
		emr-serverless-fips.us-west-2.amazonaws.com	
Africa (Cape Town)	af-south-1	emr-serverless.af-south-1.amazonaws.com	HTTPS
Asia Pacific (Hong Kong)	ap-east-1	emr-serverless.ap-east-1.amazonaws.com	HTTPS
Asia Pacific (Jakarta)	ap-southeast-3	emr-serverless.ap-southeast-3.amazonaws.com	HTTPS
Asia Pacific (Mumbai)	ap-south-1	emr-serverless.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (Osaka)	ap-northeast-3	emr-serverless.ap-northeast-3.amazonaws.com	HTTPS

Region name	Region	Endpoint	Protocol
Asia Pacific (Seoul)	ap-northeast-2	emr-serverless.ap-northeast-2.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	emr-serverless.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	emr-serverless.ap-southeast-2.amazonaws.com	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	emr-serverless.ap-northeast-1.amazonaws.com	HTTPS
Canada (Central)	ca-central-1 (limited to the following Availability Zones: cac1-az1 and cac1-az2)	emr-serverless.ca-central-1.amazonaws.com	HTTPS
Europe (Frankfurt)	eu-central-1	emr-serverless.eu-central-1.amazonaws.com	HTTPS

Region name	Region	Endpoint	Protocol
Europe (Ireland)	eu-west-1	emr-serverless.eu-west-1.amazonaws.com	HTTPS
Europe (London)	eu-west-2	emr-serverless.eu-west-2.amazonaws.com	HTTPS
Europe (Milan)	eu-south-1	emr-serverless.eu-south-1.amazonaws.com	HTTPS
Europe (Paris)	eu-west-3	emr-serverless.eu-west-3.amazonaws.com	HTTPS
Europe (Spain)	eu-south-2	emr-serverless.eu-south-2.amazonaws.com	HTTPS
Europe (Stockholm)	eu-north-1	emr-serverless.eu-north-1.amazonaws.com	HTTPS
Middle East (Bahrain)	me-south-1	emr-serverless.me-south-1.amazonaws.com	HTTPS

Region name	Region	Endpoint	Protocol
Middle East (UAE)	me-central-1	emr-serverless.me-central-1.amazonaws.com	HTTPS
South America (São Paulo)	sa-east-1	emr-serverless.sa-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-East)	us-gov-east-1	emr-serverless.us-gov-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-West)	us-gov-west-1	emr-serverless.us-gov-west-1.amazonaws.com	HTTPS

Service quotas

Service quotas, also known as *limits*, are the maximum number of service resources or operations that your AWS account can use. EMR Serverless collects service quota usage metrics every minute and publishes them in the `AWS/Usage` namespace.

> **Note**
>
> New AWS accounts might have initial lower quotas that can increase over time. Amazon EMR Serverless monitors account usage within each AWS Region, and then automatically increases the quotas based on your usage.

The following table lists the service quotas for EMR Serverless. For more information, see AWS service quotas.

Name	Default limit	Adjustable?	Description
Max concurrent vCPUs per account	16	Yes	The maximum number of vCPUs that can concurrently run for the account in the current AWS Region.
Max Queued Jobs Per Account	2000	Yes	The maximum number of queued jobs for the account in the current AWS Region.

API limits

The following describes the API limits per Region for your AWS account.

Resource	Default quota
ListApplications	10 transactions per second. Burst of 50 transactions per second.
CreateApplication	1 transaction per second. Burst of 25 transactions per second.
DeleteApplication	1 transaction per second. Burst of 25 transactions per second.
GetApplication	10 transactions per second. Burst of 50 transactions per second.
UpdateApplication	1 transaction per second. Burst of 25 transactions per second.

Resource	Default quota
ListJobRuns	1 transaction per second. Burst of 25 transactions per second.
StartJobRun	1 transaction per second. Burst of 25 transactions per second.
GetDashboardForJobRun	1 transaction per second. Burst of 2 transactions per second.
CancelJobRun	1 transaction per second. Burst of 25 transactions per second.
GetJobRun	10 transactions per second. Burst of 50 transactions per second.
StartApplication	1 transaction per second. Burst of 25 transactions per second.
StopApplication	1 transaction per second. Burst of 25 transactions per second.

Other considerations

The following list contains other considerations with EMR Serverless.

- EMR Serverless is available in the following AWS Regions:
  - US East (Ohio)
  - US East (N. Virginia)
  - US West (N. California)
  - US West (Oregon)
  - Africa (Cape Town)
  - Asia Pacific (Hong Kong)
  - Asia Pacific (Jakarta)
  - Asia Pacific (Mumbai)
  - Asia Pacific (Osaka)
  - Asia Pacific (Seoul)
  - Asia Pacific (Singapore)
  - Asia Pacific (Sydney)
  - Asia Pacific (Tokyo)
  - Canada (Central)
  - Europe (Frankfurt)
  - Europe (Ireland)
  - Europe (London)
  - Europe (Milan)
  - Europe (Paris)
  - Europe (Spain)
  - Europe (Stockholm)
  - Middle East (Bahrain)
  - Middle East (UAE)
  - South America (São Paulo)
  - AWS GovCloud (US-East)
  - AWS GovCloud (US-West)

For a list of endpoints associated with these Regions, see Service endpoints.

- The default timeout for a job run is 12 hours. You can change this setting with the `executionTimeoutMinutes` property in the `startJobRun` API or the AWS SDK. You can set `executionTimeoutMinutes` to 0 if you want your job run to never time out. For example, if you have a streaming application, you can set `executionTimeoutMinutes` to 0 to allow the streaming job to run continuously.
- The `billedResourceUtilization` property in the `getJobRun` API shows the aggregate vCPU, memory, and storage that AWS has billed for the job run. Billed resources include a 1-minute minimum usage for workers, plus additional storage over 20 GB per worker. These resources don't include usage for idle pre-initialized workers.
- Without VPC connectivity, a job can access some AWS service endpoints in the same AWS Region. These services include Amazon S3, AWS Glue, AWS Lake Formation, Amazon CloudWatch Logs, AWS KMS, AWS Security Token Service, Amazon DynamoDB, and AWS Secrets Manager. You can enable VPC connectivity to access other AWS services through AWS PrivateLink, but you aren't required to do this. To access external services, you can create your application with a VPC.
- EMR Serverless doesn't support HDFS. The local disks on workers are temporal storage that EMR Serverless uses to shuffle and process data during job runs.
- EMR Serverless doesn't support the existing emr-dynamodb-connector.

Amazon EMR Serverless release versions

An Amazon EMR release is a set of open source applications from the big data ecosystem. Each release includes big data applications, components, and features that you select to have Amazon EMR Serverless deploy and configure when you run your job.

With Amazon EMR 6.6.0 and higher, you can deploy EMR Serverless. This deployment option isn't available with earlier Amazon EMR release versions. When you submit your job, you must specify one of the following supported releases.

Topics

EMR Serverless 7.6.0

The following table lists the application versions available with EMR Serverless 7.6.0.

Application	Version
Apache Spark	3.5.3
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.5.0

The following table lists the application versions available with EMR Serverless 7.5.0.

Application	Version
Apache Spark	3.5.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.4.0

The following table lists the application versions available with EMR Serverless 7.4.0.

Application	Version
Apache Spark	3.5.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.3.0

The following table lists the application versions available with EMR Serverless 7.3.0.

Application	Version
Apache Spark	3.5.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.3.0 release notes

- **Job concurrency and queuing with EMR Serverless** – Job concurrency and queuing is enabled by default when you create a new EMR Serverless application on Amazon EMR release 7.3.0 or higher. For more information, see the section called "Job concurrency and queuing", which details how to get started with concurrency and queuing and also contains a list of feature considerations.

EMR Serverless 7.2.0

The following table lists the application versions available with EMR Serverless 7.2.0.

Application	Version
Apache Spark	3.5.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.2.0 release notes

- **Lake Formation with EMR Serverless** – you can now use AWS Lake Formation to apply fine-grained access controls on Data Catalog tables that are backed by S3. This capability lets you configure table, row, column, and cell level access controls for read queries within your EMR Serverless Spark jobs. For more information, see the section called "Lake Formation for FGAC" and the section called "Considerations".

EMR Serverless 7.1.0

The following table lists the application versions available with EMR Serverless 7.1.0.

Application	Version
Apache Spark	3.5.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.0.0

The following table lists the application versions available with EMR Serverless 7.0.0.

Application	Version
Apache Spark	3.5.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.15.0

The following table lists the application versions available with EMR Serverless 6.15.0.

Application	Version
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.15.0 release notes

- **TLS support** – With Amazon EMR Serverless releases 6.15.0 and higher, you can enable mutual-TLS encrypted communication between workers in your Spark job runs. When enabled, EMR Serverless automatically generates a unique certificate for each worker that it provisions under a job runs that workers utilize during TLS handshake to authenticate each other and establish an encrypted channel to process data securely. For more information about mutual-TLS encryption, see Inter-worker encryption.

EMR Serverless 6.14.0

The following table lists the application versions available with EMR Serverless 6.14.0.

Application	Version
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.13.0

The following table lists the application versions available with EMR Serverless 6.13.0.

Application	Version
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.12.0

The following table lists the application versions available with EMR Serverless 6.12.0.

Application	Version
Apache Spark	3.4.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.11.0

The following table lists the application versions available with EMR Serverless 6.11.0.

Application	Version
Apache Spark	3.3.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.11.0 release notes

- **Access S3 resources in other accounts** - With releases 6.11.0 and higher, you can configure multiple IAM roles to assume when you access Amazon S3 buckets in different AWS accounts from EMR Serverless.

EMR Serverless 6.10.0

The following table lists the application versions available with EMR Serverless 6.10.0.

Application	Version
Apache Spark	3.3.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.10.0 release notes

- For EMR Serverless applications with release 6.10.0 or higher, the default value for the `spark.dynamicAllocation.maxExecutors` property is `infinity`. Earlier releases default to `100`. For more information, see Spark job properties.

EMR Serverless 6.9.0

The following table lists the application versions available with EMR Serverless 6.9.0.

Application	Version
Apache Spark	3.3.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.9.0 release notes

- The Amazon Redshift integration for Apache Spark is included in Amazon EMR releases 6.9.0 and later. Previously an open-source tool, the native integration is a Spark connector that you can use to build Apache Spark applications that read from and write to data in Amazon Redshift and Amazon Redshift Serverless. For more information, see Using Amazon Redshift integration for Apache Spark on Amazon EMR Serverless.
- EMR Serverless release 6.9.0 adds support for AWS Graviton2 (arm64) architecture. You can use the `architecture` parameter for the `create-application` and `update-application` APIs to choose the arm64 architecture. For more information, see Amazon EMR Serverless architecture options.
- You can now export, import, query, and join Amazon DynamoDB tables directly from your EMR Serverless Spark and Hive applications. For more information, see Connecting to DynamoDB with Amazon EMR Serverless.

Known issues

- If you use the the Amazon Redshift integration for Apache Spark and have a time, timetz, timestamp, or timestamptz with microsecond precision in Parquet format, the connector rounds the time values to the nearest millisecond value. As a workaround, use the text unload format `unload_s3_format` parameter.

EMR Serverless 6.8.0

The following table lists the application versions available with EMR Serverless 6.8.0.

Application	Version
Apache Spark	3.3.0
Apache Hive	3.1.3
Apache Tez	0.9.2

EMR Serverless 6.7.0

The following table lists the application versions available with EMR Serverless 6.7.0.

Application	Version
Apache Spark	3.2.1
Apache Hive	3.1.3
Apache Tez	0.9.2

Engine-specific changes, enhancements, and resolved issues

The following table lists a new engine-specific feature.

Change	Description
Feature	Tez scheduler now supports preemption of Tez task instead of preemption of container

EMR Serverless 6.6.0

The following table lists the application versions available with EMR Serverless 6.6.0.

Application	Version
Apache Spark	3.2.0
Apache Hive	3.1.2
Apache Tez	0.9.2

EMR Serverless initial release notes

- EMR Serverless supports the Spark configuration classification `spark-defaults`. This classification changes values in Spark's `spark-defaults.conf` XML file. Configuration classifications allow you to customize applications. For more information, see Configure applications.
- EMR Serverless supports the Hive configuration classifications `hive-site`, `tez-site`, `emrfs-site`, and `core-site`. This classification can change the values in Hive's `hive-site.xml` file, Tez's `tez-site.xml` file, Amazon EMR's EMRFS settings, or Hadoop's `core-site.xml` file, respectively. Configuration classifications allow you to customize applications. For more information, see Configure applications.

Engine-specific changes, enhancements, and resolved issues

- The following table lists Hive and Tez backports.

Hive and Tez changes

Change	Description
Backport	TEZ-4430: Fixed issue with `tez.task.launch.cmd-opts` property
Backport	HIVE-25971: Fixed Tez task shutdown delays due to open cached thread pool

Document history

The following table describes the important changes to the documentation since the last release of EMR Serverless. For more information about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
New release	EMR Serverless 7.2.0	July 25, 2024
New release	EMR Serverless 7.1.0	April 17, 2024
Update to an existing policy.	Added the new `Sid` `CloudWatchPolicySt atement` and `EC2Policy Statement` to the AmazonEMRServerles sServiceRolePolicy policy.	January 25, 2024
New release	EMR Serverless 7.0.0	December 29, 2023
New release	EMR Serverless 6.15.0	November 17, 2023
New feature	Configure multiple IAM roles to assume when you access Amazon S3 buckets in different accounts from EMR Serverless (6.11 and higher)	October 18, 2023
New release	EMR Serverless 6.14.0	October 17, 2023
New feature	Default application configura tion for EMR Serverless	September 25, 2023
Update to default Hive properties	Updated the default values for `hive.driver.disk` , `hive.tez.disk.size` , `hive.tez.auto.redu cer.parallelism` , and	September 12, 2023

	`tez.grouping.min-size` Hive job properties.	
New release	EMR Serverless 6.13.0	September 11, 2023
New release	EMR Serverless 6.12.0	July 21, 2023
New release	EMR Serverless 6.11.0	June 8, 2023
Update to service-linked role policy	Updated the `AmazonEMRServerlessServiceRolePolicy` SLR role to publish account-level usage in `"AWS/Usage"` namespace.	April 20, 2023
EMR Serverless general availability (GA)	This is the first public release of EMR Serverless.	June 1, 2022