Amazon EMR

Amazon EMR Serverless User Guide

Amazon EMR: Amazon EMR Serverless User Guide

Table of Contents

What is Amazon EMR Serverless?

Amazon EMR Serverless is a new deployment option for Amazon EMR. EMR Serverless provides a serverless runtime environment that simplifies running analytics applications using 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 required by the applications, acquires these resources to process your jobs, and relinquishes them when the jobs finish. For use cases where applications require a response within seconds, such as interactive data analysis, you can pre-initialize required resources during application creation.

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

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

Concepts

EMR Serverless terms and concepts.

Release version

An Amazon EMR release is a set of open-source applications from the big data ecosystem. Each release comprises different big data applications, components, and features that you select to have EMR Serverless deploy and configure to run your applications. When creating an application, you must specify its release version. You'll choose the Amazon EMR release version along with the open source framework version you want to use in your application.

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 you want to use. To determine your release version, see Release versions (p. 68).
- The specific runtime that you want your application to use, such as Apache Spark or Apache Hive.

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

Each EMR Serverless application is strictly isolated from other applications and runs on a secure Amazon Virtual Private Cloud (VPC). Additionally, you can use IAM policies to define which IAM 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 for the following scenarios:

- Using different open source frameworks
- Using different versions of open source frameworks for different use cases
- Performing A/B testing when upgrading from one version to another
- Maintaining separate logical environments for test and production scenarios
- Providing separate logical environments for different teams with independent cost controls and usage tracking
- Separating different line-of-business applications

EMR Serverless is a Regional service that simplifies running workloads across multiple Availability Zones within a Region. To learn more about using applications with EMR Serverless, see Interacting with your application (p. 12).

Job run

A job run is a request submitted to an EMR Serverless application that is asynchronously executed and tracked through completion. Examples of jobs include a HiveQL query submitted to an Apache Hive application or a PySpark data processing script submitted to an Apache Spark application. When submitting a job, you must specify an execution role, authored in IAM, that will be used by the job to access AWS resources, such as Amazon S3 objects. Multiple job run requests can be submitted to an application, and each job run can use a different execution role to access AWS resources. EMR Serverless starts executing jobs as soon as they are received and runs multiple job requests concurrently. To learn more about running jobs, see Running jobs (p. 16).

Workers

An EMR Serverless application internally uses workers to execute your workloads. The default size of these workers are based on your application type and Amazon EMR release version. You can override these sizes when scheduling a job run.

When a job is submitted, EMR Serverless computes the resources needed 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 depending on the workload and parallelism required at every stage of the job, removing the need for you to estimate the number of workers required to run your workloads.

Pre-initialized capacity

EMR Serverless provides a feature that keeps workers initialized and ready to respond in seconds, effectively creating a warm pool of workers for an application. This feature is called *pre-initialized capacity* and can be configured for each application by setting the `initial-capacity` parameter of an application. Pre-initialized capacity allows jobs to start immediately, making it ideal for implementing iterative applications and time-sensitive jobs. To learn more about pre-initialized workers, see Configuring and managing pre-initialized capacity (p. 13).

Setting up

Step 1: Sign up for AWS

When you sign up for AWS, your AWS account is automatically signed up for all services, including the generally available Amazon EMR deployment options. You are charged only for the services that you use. If you have an AWS account already, skip to the next step. If you don't have an AWS account, use the following procedure to create one.

To create 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.

Step 2: Create an IAM user

As a best practice, create an AWS Identity and Access Management (IAM) user with administrator permissions, and then use that IAM user for all work that does not require root credentials. Create a password for console access, and create access keys to use command line tools. For instructions, see Creating your first IAM admin user and group in the IAM User Guide.

You can use this same process to create more groups and users and to give your users access to your AWS account resources. To learn about using policies that restrict user permissions to specific AWS resources, see Access management. If you choose to create a separate user to work with EMR Serverless, ensure the user has sufficient permissions to invoke EMR Serverless actions by attaching an IAM policy to the IAM user. For more information, see Access policies (p. 47). However, if you choose to continue with an Admin user, no further action will be required.

Step 3: Install and configure the AWS CLI

To set up EMR Serverless, you must have the latest version of AWS CLI installed. 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.

To configure the AWS CLI and set up of secure access to AWS services, including EMR Serverless, see Quick configuration with `aws configure`.

Getting started

This tutorial helps you get started using EMR Serverless by deploying a sample Spark or Hive workload. You'll create your application, run the sample application with logs stored in your S3 bucket and view event logs in the Spark History Server. Note that, for simplicity, we have chosen default options in most parts of this tutorial.

Prerequisites

Before you launch an EMR Serverless application, make sure you complete the following tasks:

- EMR Serverless is currently in preview release. To access the preview of EMR Serverless, follow the sign-up steps at https://pages.awscloud.com/EMR-Serverless-Preview.html.
- You must update the AWS CLI with the latest service model for EMR Serverless. Once you've received confirmation of access, use the following command to download the latest API model file and update the AWS CLI.

```
aws s3 cp s3://elasticmapreduce/emr-serverless-preview/artifacts/latest/dev/cli/
service.json ./service.json
aws configure add-model --service-model file://service.json
```

- To use EMR Serverless, you must choose the AWS Region where preview is available. This applies to any AWS services and resources that EMR Serverless will need to access as part of running your workloads. Preview is currently available in US East (N. Virginia) us-east-1, and you may want to configure the AWS CLI to send all your AWS requests to this specific region by default. You can do so with the following command.

```
aws configure set region us-east-1
```

- Validate that the AWS CLI configuration and permissions to interact with EMR Serverless are correctly set up. You can do so by running the following command to see a list of your EMR Serverless applications in your current Region.

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

  If the command returns with an error, see Troubleshooting EMR Serverless identity and access (p. 49).

Step 1: Plan an EMR Serverless application

Prepare output log 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 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*.

As noted in the prerequisites, the S3 bucket must be created in the same Region where EMR Serverless is available (`us-east-1`). Replace any further reference to *`DOC-EXAMPLE-BUCKET`* with the name of the newly created bucket.

Set up a job runtime role

Job runs in EMR Serverless use a runtime role that provide granular permissions to specific AWS services and resources at runtime. In this tutorial, the data and scripts are hosted in a public S3 bucket. The output, including logs, will be stored in *DOC-EXAMPLE-BUCKET*.

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

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 `sampleJobRuntimeRole` using the trust policy created in the previous step.

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

   Take note of the ARN in the output, as you will use the ARN of the new role during job submission, henceforth referred to as the *<runtime_role_arn>*.

3. Create a file named `emr-sample-access-policy.json` that defines the IAM policy for your workload to get read access the script and data stored in public S3 buckets and read-write access to *DOC-EXAMPLE-BUCKET*. You must replace *DOC-EXAMPLE-BUCKET* in the policy below with the actual bucket name created in **Step 1**).

```
{
    "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:::DOC-EXAMPLE-BUCKET",
                "arn:aws:s3:::DOC-EXAMPLE-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": ["*"]
        }
    ]
}
```

4. Create an IAM policy named `sampleS3andGlueAccessPolicy` using the policy file created in the previous step. 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 sampleS3andGlueAccessPolicy \
    --policy-document file://emr-sample-access-policy.json
```

   Take note of the new policy's ARN in the output, as you will substitute it for *<policy_arn>* in the next step.

5. Attach the IAM policy `sampleS3andGlueAccessPolicy` to the job runtime role `sampleJobRuntimeRole`.

```
aws iam attach-role-policy \
    --role-name sampleJobRuntimeRole \
    --policy-arn <policy_arn>
```

Step 2: Create an EMR Serverless application

Now you're ready create a new application using EMR Serverless.

Spark

To create a Spark application, run the following command.

```
aws emr-serverless create-application \
    --release-label emr-6.5.0-preview \
    --type "SPARK" \
```

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

Hive

To create a Hive application, run the following command.

```
aws emr-serverless create-application \
    --release-label emr-6.5.0-preview \
    --type "HIVE" \
    --name my-application
```

Take note of the application ID returned in the output, as you will use the ID to start the application and during job submission, henceforth referred to as the *<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 while creating the application. You may also choose to set a limit for the total maximum capacity that an application can use by setting the `maximumCapacity` parameter. To learn more about these options, see Configuring and managing pre-initialized capacity (p. 13).

Step 3: Start your application

Now you can schedule a job using your application, you must start the application. To check the state of your application, run the following command, substituting *<application_id>* with the ID of your new application.

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

When application has reached the `CREATED` state, start your application using the following command.

```
aws emr-serverless start-application \
    --application-id <application_id>
```

Before moving to the next step, ensure your application has reached the `STARTED` state using the `get-application` API.

Step 4: Schedule a job run to your EMR Serverless application

Now your EMR Serverless application is ready to run jobs.

Spark

In this tutorial, we use a PySpark script to compute the number of occurrences of unique words across multiple text files. Both the script and the dataset are stored in a public, read-only S3 bucket. The output file and the log data from the Spark runtime will be pushed to `/output` and `/logs` directory in the S3 bucket you created.

To run a Spark job

1. In the command below, substitute *<application_id>* with your application ID. Substitute *<runtime_role_arn>* with the runtime role ARN you created in **Step 1**. Replace all *DOC-*

EXAMPLE-BUCKET strings with the Amazon S3 bucket you created, adding `/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 <runtime_role_arn> \
    --job-driver '{
        "sparkSubmit": {
          "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
          "entryPointArguments": ["s3://DOC-EXAMPLE-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"
        }
    }' \
    --configuration-overrides '{
        "monitoringConfiguration": {
          "s3MonitoringConfiguration": {
            "logUri": "s3://DOC-EXAMPLE-BUCKET/emr-serverless-spark/logs"
           }
        }
    }'
```

2. Note the job run ID returned in the output, as you'll 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 which 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 named `hive-query.ql` that contains all the queries 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://DOC-EXAMPLE-BUCKET/emr-serverless-hive/query/hive-
query.ql
```

3. In the command below, substitute *<application_id>* with your own application ID. Substitute *<runtime_role_arn>* with the runtime role ARN you created in **Step 1**. Replace all *DOC-EXAMPLE-BUCKET* strings with the Amazon S3 bucket you created, adding `/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 <runtime_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"
            }
        }],
        "monitoringConfiguration": {
          "s3MonitoringConfiguration": {
            "logUri": "s3://DOC-EXAMPLE-BUCKET/emr-serverless-hive/logs"
           }
        }
    }'
```

4. Note the job run ID returned in the output, as you'll replace *<job_run_id>* with this ID in the following steps.

Step 5: 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 using 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://DOC-EXAMPLE-BUCKET/emr-serverless-spark/logs`, the logs for this specific job run can be found under `s3://DOC-EXAMPLE-BUCKET/emr-serverless-spark/logs/applications/<application_id>/jobs/<job_run_id>`.

For Spark applications, EMR Serverless will push event logs every 30 seconds to the `sparklogs` folder in your S3 log destination. The Spark runtime logs for the driver and executors will upload upon completion of the job to folders named appropriately by the worker type, such as `driver` or `executor`. The output of the PySpark job will upload upon sucessful completion of the job to `s3://DOC-EXAMPLE-BUCKET/output/`.

Hive

You can check for the state of your Hive job using 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://DOC-EXAMPLE-BUCKET/emr-serverless-hive/logs`, the logs for this specific job run can be found under `s3://DOC-EXAMPLE-BUCKET/emr-serverless-hive/logs/applications/<application_id>/jobs/<job_run_id>`.

For Hive applications, EMR Serverless will continuously upload the Hive driver and Tez tasks logs to the `HIVE_DRIVER` or `TEZ_TASK` folders, respectively, of your S3 log destination. Once the job run reaches the `SUCCEEDED` state, the output of your Hive query will be available in the Amazon S3 location you specified in the `monitoringConfiguration` field of `configurationOverrides`.

Step 6: Clean up

When you're done working with this tutorial, consider deleting the resources you created. This will help you avoid any unnecessary expenses. Note that in preview, there is no additional cost to using EMR Serverless. However, we still recommend following best practives by releasing resources that you don't intend to use again.

Delete your application

To delete an application, it must be in the `STOPPED` state. Use the following command to stop the application.

```
aws emr-serverless stop-application \
    --application-id <application_id>
```

Once the application is in the `STOPPED` state, use the following command to delete the application.

```
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 *DOC-EXAMPLE-BUCKET* with the actual name of the S3 bucket created in **Step 1**.

```
aws s3 rm s3://DOC-EXAMPLE-BUCKET --recursive
aws s3api delete-bucket --bucket DOC-EXAMPLE-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 sampleJobRuntimeRole \
    --policy-arn <policy_arn>
```

To delete the role, use the following command.

```
aws iam delete-role \
```

```
    --role-name sampleJobRuntimeRole
```

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

```
aws iam delete-policy \
    --policy-arn <policy_arn>
```

This concludes the tutorial. For examples of running Hive applications, see Running Hive jobs (p. 23).

Interacting with your application

In this section, we'll cover how you can interact with your Amazon EMR Serverless application using the AWS CLI and the defaults for Spark and Hive engines.

Topics

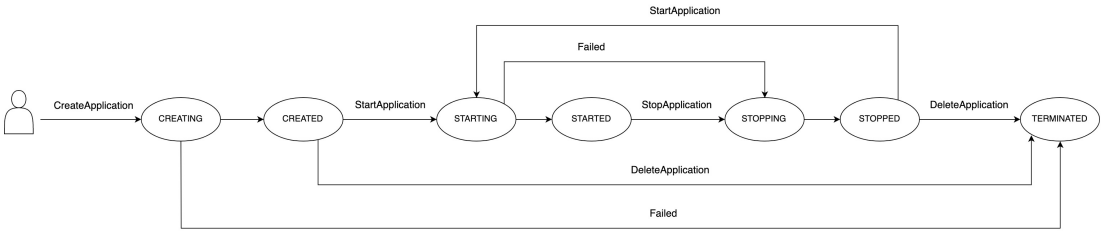
Application states

When you create an application with Amazon 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 is not yet ready to use.
Created	The application has been created but has not yet provisioned capacity. It can be modified to make changes to the initial capacity configuration.
Starting	The application is starting and is provisioning capacity.
Started	The application is ready to accept new jobs. Jobs will only be accepted in this state.
Stopping	All jobs have completed and the application is releasing its capacity. EMR Serverless may move an application to this state when there are failures in provisioning capacity.
Stopped	The application is stopped and no resources are running on the application. It can be modified to make changes to the initial capacity configuration.
Terminated	The application has been terminated and will not appear on your list. EMR Serverless may move an application to this state when there are failures in creation.

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



Working with your application on the AWS CLI

This section covers how to create, describe, and delete individual applications on the command line, as well as how to list all of your applications to view them at a glance. For more application operations, such starting, stopping, and updating your application, see the EMR Serverless API Reference.

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 and managing pre-initialized capacity

EMR Serverless provides an optional feature that keeps driver and workers pre-initialized and ready to respond in seconds, effectively creating a warm pool of workers for an application. This feature is called *pre-initialized capacity* and can be configured for each application by setting the `initialCapacity` parameter of an application to the number of workers you want to pre-initialize. Pre-initialized worker capacity allows jobs to start immediately, making it ideal for implementing iterative applications and time-sensitive jobs.

When a job is run, if any workers from `initialCapacity` are available (not already in use from jobs previously submitted), those resources are used to start running the job. If those resources are not

available because they are in use by other jobs, or if resources are insufficient to execute the job because the job requires more than what is available from `intialCapacity`, then additional workers are requested and acquired, up to the maximum limits on resources set for the application. When jobs finish running, the workers used by the job are released, and the number of resources available for the application returns to `initialCapacity`. An application maintains the `initialCapacity` of resources even after jobs finish running. Excess resources beyond `initialCapacity` are released immediately when they're no longer required to run jobs.

Pre-initialized capacity is available and ready to use when the application has started. It is decommissioned 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. For the entire duration that the application is in the `STARTED` state, EMR Serverless ensures that the pre-initialized capacity is available for use or is in use by jobs or interactive workloads. Capacity is replenished for released or failed containers to maintain the number of workers specified in the `InitialCapacity` parameter. For an application with no pre-initialized capacity, the state can immediately transition from `CREATED` to `STARTED`.

You can configure pre-initialized capacity to be released if unused for a certain period of time, with a default of 15 minutes. A stopped application starts automatically when a new job is submitted. You can set these automatic start and stop configurations at the time of application creation, or you can modify them when the application is in a `CREATED` or `STOPPED` state.

You can modify the `InitialCapacity` counts, and specify compute configurations such as vCPU, memory, and disk, for each worker. Make sure you specify all compute configurations when changing values, as partial modifications are not supported. Modifications are only allowed when the application is in the `CREATED` or `STOPPED` state.

Customizing pre-initialized capacity for specific big data frameworks

You can further customize pre-initialized capacity to suit workloads running on specific big data frameworks. For example, when running 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 are used to 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-5.34.0-preview. The application starts with 5 pre-initialized Hive drivers, each with 2 vCPU and 6 GB of memory, and 50 pre-initialized Tez task workers, each with 1 vCPU and 6 GB of memory. When Hive queries are 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-5.34.0-preview \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "resourceConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "TEZ_TASK": {
```

```
        "workerCount": 50,
        "resourceConfiguration": {
            "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 running Apache Spark 3.1 based on Amazon EMR release 6.5. 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 are 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 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. In order for jobs to use pre-initialized workers, the initial capacity memory configuration should be greater than the memory requested by the job and the overhead..

```
aws emr-serverless create-application \
  --type "SPARK" \
  --name <"my_application_name"> \
  --release-label "emr-6.5.0-preview" \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "resourceConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "EXECUTOR": {
        "workerCount": 50,
        "resourceConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
  }'
```

Running jobs

After you've provisioned your application, you can submit jobs to it. This section covers how to use the AWS CLI to run jobs on your application and what the defaults are for each type of application 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. It then passes through the following states until it succeeds (exits with code `0`) or fails (exits with a non-zero code).

Job runs can have the following states:

State	Description
Submitted	The initial job state when a job run is submitted to EMR Serverless. The job is waiting to be scheduled for the application, and EMR Serverless is working on prioritizing and scheduling this job run.
Pending	The job run is being evaluated by the scheduler to prioritize and schedule this job run for the application.
Scheduled	The job run has been scheduled for the application, and EMR Serverless is allocating resources to execute it.
Running	The job run has necessary initial resources and is running in the application. In Spark applications, this means that the Spark driver process is in the `running` state.
Failed	The job run failed to be submitted to the application or it completed unsuccessfully. See `StateDetails` for additional information about this job failure.
Completed	The job run completed successfully.
Cancelling	The job run has been requested for cancellation, either through the `CancelJobRun` API or due to

State	Description
	timeout. 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 it used have been released.

Submitting jobs on the AWS CLI

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

To submit a new job, use `start-job-run` and supply the ID of the application you want to run, along with job-specific properties. For Spark examples, see Running Spark jobs (p. 17). For Hive examples, see Running Hive jobs (p. 23). This command returns your `application-id`, ARN, and new `job-id`.

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, call `list-job-runs`. This command returns an abbreviated set of properties including 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'd like to see up to 50. The the following command demonstrates how to specify that you'd like to see your two last job runs.

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

To cancel a job, call `cancel-job-run`, supplying both the `application-id` and the `job-id` of the job you want to cancel.

```
aws emr-serverless cancel-job-run \
--job-run-id <job_id> \
--application-id <application_id>
```

For more information on running jobs using the AWS CLI, see the EMR Serverless API Reference.

Running Spark jobs

You can run Spark jobs on an application with the `type` parameter set to `'SPARK'`. Jobs must be compatible with the Spark version referenced in the Amazon EMR release version. For example, when running jobs on an application with Amazon EMR release 6.5, your job must be compatible with Apache Spark 3.1.2.

To run a Spark job, you must specify the following parameters when using the `start-job-run` API.

Execution role (`executionRoleArn`)

This is an IAM role ARN 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 is located
- Write to S3 buckets where you intend to write your final output
- Write logs to an S3 bucket or prefix specified by `S3MonitoringConfiguration`
- Access to KMS keys if you use KMS keys to encrypt data in your S3 bucket
- Access to AWS Glue Catalog if you use SparkSQL

Failure to provide these permissions to the IAM role can lead to job failures. If your Spark job is reading or writing data to or from other data sources, make sure that the appropriate permissions are specified in this IAM role. For more information, see Using job execution roles with EMR Serverless (p. 42) and Logging (p. 57).

Job driver (`jobDriver`)

A job's driver is used to provide input to the job. This parameter accepts only one value for the job type that you want to run. For a Spark job, that value is `sparkSubmit`. You can use this job type to run Scala, Java, PySpark, SparkR, and any other supported jobs through Spark submit. This job type has the following parameters:

- `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, the main entry class should be specified in the `SparkSubmitParameters` using the `--class` argument.
- `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. Each argument in the array should be separated by a comma.
- `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 the `--conf` or `--class` parameters.

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

Configuration overrides (`configurationOverrides`)

This parameter is used for overriding application and monitoring level configuration properties. This parameter accepts a JSON object having the following two fields.

- `applicationConfiguration` - This field allows you to override the default configurations for applications by supplying a configuration object. 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 you want to override in that file. You can specify multiple classifications for multiple applications in a single JSON object. The configuration classifications that are available vary by specific release version for Amazon EMR. For a list of configuration classifications that are available for each release version of Amazon EMR, see Release versions (p. 68).

  If you pass the same configuration in an application override and in Spark submit parameters, the Spark submit parameters take precedence. The complete configuration priority list follows, in order of highest priority to lowest priority:
  - Configuration supplied when creating `SparkSession`.
  - Configuration supplied as part of `sparkSubmitParameters` using `–conf`.
  - Configuration provided as part of application overrides.
  - Optimized configurations chosen by Amazon EMR for the release.

- Default open source configurations for the application.

- `monitoringConfiguration` - This field allows you 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 hosting your application, and in the same Region where your job is running.

Spark job defaults

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

Key	Explanation	Default value
`spark.emr-serverless.allocation.batch.size`	The number of containers to request in each cycle of executor allocation. There is a 1-second gap between each allocation cycle.	20
`spark.emr-serverless.allocation.executor.timeout`	The time to wait for a newly-created executor container to reach the running state before the request is cancelled.	300s
`spark.emr-serverless.memoryOverheadFactor`	Sets the Memory Overhead Factor to be added to the driver and executor container memory.	0.1
`spark.executor.memory`	The amount of memory used by each executor.	14G
`spark.executor.cores`	The number of cores used by each executor.	4
`spark.driver.memory`	The amount of memory used by the driver.	14G
`spark.driver.cores`	The number of cores used by the driver.	4
`spark.emr-serverless.driver.disk`	The Spark driver disk.	21G
`spark.emr-serverless.executor.disk`	The Spark executor disk.	21G
`spark.executor.instances`	The number of Spark executor containers to allocate.	3
`spark.executor.extraJavaOptions`	Extra Java options for the Spark executor.	`NULL`
`spark.driver.extraJavaOptions`	Extra Java options for the Spark driver.	`NULL`
`spark.driver.extraJavaOptions`	Extra Java options for the Spark driver.	`NULL`

Key	Explanation	Default value
`spark.dynamicAllocation.enabled`	Enables Spark Dynamic Resource Allocation.	`TRUE`
`spark.emr-serverless.driverEnv.[KEY]`	Adds additional environment variables to the Spark driver.	`NULL`
`spark.executorEnv.[KEY]`	Adds additional environment variables to the Spark executors.	`NULL`
`spark.files`	A comma-separated list of files to be placed in the working directory of each executor. The file paths of these files in executors can be accessed by running `SparkFiles.get(fileName)`.	`NULL`
`spark.jars`	Additional jars to add to the runtime classpath of the driver and executors.	`NULL`
`spark.archives`	A comma-separated list of archives to be extracted into the working directory of each executor. Supported file types include `.jar`, `.tar.gz`, `.tgz` and `.zip`. You can specify the directory name to unpack by adding `#` after the file name to unpack. For example, `file.zip#directory`. This configuration is experimental.	`NULL`
`spark.submit.pyFiles`	A comma-separated list of `.zip`, `.egg`, or `.py` files to place in the `PYTHONPATH` for Python apps.	`NULL`
`spark.sql.warehouse.dir`	The default location for managed databases and tables.	The value of `$PWD/spark-warehouse`
`spark.hadoop.hive.metastore.client.factory.class`	The Hive metastore implementation class.	`NULL`
`spark.authenticate`	Enables authentication of Spark's internal connections.	`TRUE`
`spark.network.crypto.enabled`	Enables AES-based RPC encryption, including the new authentication protocol added in 2.2.0.	`FALSE`
`spark.dynamicAllocation.enabled`	Enables dynamic resource allocation, which scales the number of executors registered with the application up or down, based on the workload.	`TRUE`

Key	Explanation	Default value
`spark.dynamicAllocation.executorIdleTimeout`	The duration, in time an executor can have before it will be removed. This only applies if dynamic allocation is enabled.	60s
`spark.dynamicAllocation.initialExecutors`	The initial number of executors to run if dynamic allocation is enabled.	3
`spark.dynamicAllocation.maxExecutors`	The upper bound for the number of executors if dynamic allocation is enabled.	1000
`spark.dynamicAllocation.minExecutors`	The lower bound for the number of executors if dynamic allocation is enabled.	0

The following table lists the default Spark submit parameters.

Key	Explanation	Default value
`executor-memory`	The amount of memory used by executor.	14G
`executor-cores`	The number of cores used by each executor.	4
`driver-memory`	The amount of memory used by the driver.	14G
`driver-cores`	The number of cores to used by the driver.	4
`num-executors`	The number of executors to launch.	3
`files`	A comma-separated list of files to be placed in the working directory of each executor. File paths of these files in executors can be accessed via `SparkFiles.get(`*`fileName`*`)`.	`NULL`
`py-files`	A comma-separated list of `.zip`, `.egg`, or `.py` files to place on the `PYTHONPATH` for Python apps.	`NULL`
`archives`	A comma-separated list of archives to be extracted into the working directory of each executor.	`NULL`
`jars`	A comma-separated list of jars to include on the driver and executor classpaths.	`NULL`

Key	Explanation	Default value
`verbose`	Enables printing additional debug output.	`NULL`
`class`	The application's main class (for Java and Scala apps).	`NULL`
`conf`	An arbitrary Spark configuration property.	`NULL`

Spark examples

The following is an example of running a Python script using the `StartJobRun` API. For an end-to-end tutorial using this example, see Getting started (p. 4).

```
aws emr-serverless start-job-run \
    --application-id <application_id> \
    --execution-role-arn <iam_role_arn> \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
            "entryPointArguments": ["s3://DOC-EXAMPLE-BUCKET-OUTPUT/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 '{
        "monitoringConfiguration": {
           "s3MonitoringConfiguration": {
             "logUri": "s3://DOC-EXAMPLE-BUCKET-LOGGING/logs/"
           }
        }
    }'
```

The following is an example of running a Spark JAR using the `StartJobRun` API.

```
aws emr-serverless start-job-run \
    --application-id <application_id> \
    --execution-role-arn <iam_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 '{
        "monitoringConfiguration": {
           "s3MonitoringConfiguration": {
             "logUri": "s3://DOC-EXAMPLE-BUCKET-LOGGING/logs/"
           }
        }
    }'
```

Running Hive jobs

You can run Hive jobs on an application with the `type` parameter set to `'HIVE'`. Jobs must be compatible with the Hive version referenced in the Amazon EMR release version. For example, when running jobs on an application with Amazon EMR release 6.5.0, your job must be compatible with Apache Hive 3.1.2.

To run a Hive job, you must specify the following parameters when using the `start-job-run` API.

Execution role (`executionRoleArn`)

This is an IAM role ARN 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 are located
- Read and write to S3 buckets where your Hive Scratch directory and Hive Metastore warehouse directory are located
- Write to S3 buckets where you intend to write your final output
- Write logs to an S3 bucket or prefix specified by `S3MonitoringConfiguration`
- Access to KMS keys if you use KMS keys to encrypt data in your S3 bucket
- Access to AWS Glue Catalog

Failure to provide these permissions to the IAM role can lead to job failures. If your Hive job is reading or writing data to or from other data sources, make sure that the appropriate permissions are specified in this IAM role. For more information, see Using job execution roles with EMR Serverless (p. 42).

Job driver (`jobDriver`)

A job's driver is used to provide input to the job. This parameter accepts only one value for the job type that you want to run. A Hive query is passed to the `job-driver` parameter by specifying Hive as the job `type`. This job type has 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 you want to override. You can override properties by passing them to this parameter as `--hiveconf` *`<property=value>`* and variables passing by them as `--hivevar` *`<key=value>`*.
- `initQueryFile` - This is the init Hive query file. It will be executed prior to your query and can be used to initialize tables.

Configuration overrides (`configurationOverrides`)

This parameter is used for overriding application and monitoring level configuration properties. This parameters accepts a JSON objects having the following two fields:

- `applicationConfiguration` - This field allows you to override the default configurations for applications by supplying a configuration object. 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 you want to override in that file. You can specify multiple classifications for multiple applications in a single JSON object. The configuration classifications that are available vary by specific release version for Amazon EMR. For a list of configuration classifications that are available for each release version of Amazon EMR, see Release versions (p. 68).

If you pass the same configuration in an application override and in Hive parameters, the Hive parameters take precedence. The complete configuration priority list follows, in order of highest priority to lowest priority.

- Configuration supplied as part of Hive parameters using `--hiveconf <property=value>`.
- Configuration provided as part of application overrides.
- Optimized configurations chosen by Amazon EMR for the release.
- Default open source configurations for the application.

- `monitoringConfiguration` - This field allows you to specify the Amazon S3 URL (`s3MonitoringConfiguration`) where you want the EMR Serverless job to store logs of your Hive job. Make sure you've created this bucket with the same AWS account hosting your application, and in the same Region where your job is running.

Hive job properties

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

Setting	Description
`hive.exec.scratchdir`	The Amazon S3 location where temporary files are created 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 submitting a Hive job.

Setting	Description	Value
`hive.driver.memory`	The amount of memory to use per Hive driver process. This memory is shared equally between HiveCLI and Tez Application Master with 20% of headroom.	6G
`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.	21G
`hive.tez.disk.size`	The disk size for each task container.	21G
`hive.prewarm.enabled`	Enables container prewarm for Tez.	`FALSE`
`hive.prewarm.numcontainers`	The number of containers to prewarm for Tez.	10
`hive.tez.container.size`	The amount of memory to use per Tez task process.	6144

Setting	Description	Value
`hive.tez.cpu.vcores`	The number of cores to use for each Tez task.	1
`hive.max-task-containers`	The maximum number of concurrent containers. The configured mapper memory is also multuplied by this value to determines available memory that is used by split computation and task preemption.	100
`hive.exec.reducers.max`	The maximum number of reducers.	256
`hive.auto.convert.join.noconditionaltask.size`	The size below which a join is directly converted to a mapjoin.	Optimal value is calculated based on Tez task memory
`tez.runtime.io.sort.mb`	The size of the soft buffer when 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 not writing directly to disk.	Optimal value is calculated based on Tez task memory
`tez.am.task.max.failed.attempts`	The maximum number of attempts that can fail for a particular task before the task is failed. This does not count manually terminated attempts.	3
`hive.exec.stagingdir`	The name of the directory for storing temporary files that will be created inside table locations and in the scratch directory location specified using the `hive.exec.scratchdir` property.	`.hive-staging`
`hive.compute.query.using.stats`	Enables Hive to answer a few queries using statistics stored in the metastore. For basic statistics, set hive.stats.autogather to true. For a more advanced collection, run analyze table queries.	`TRUE`
`hive.vectorized.execution.enabled`	Enables vectorized mode of query execution.	`TRUE`
`hive.cbo.enable`	Enables cost-based optimizations using the Calcite framework.	`TRUE`

Setting	Description	Value
`hive.tez.auto.reducer.parallelism`	Enables Tez's auto-reducer parallelism feature. Hive will still estimate data sizes and set parallelism estimates. Tez will sample source vertices' output sizes and adjust the estimates at runtime as necessary.	`FALSE`
`hive.stats.fetch.column.stats`	Disables fetching of column statistics from the metastore. Fetching column statistics can be expensive when the number of columns is high.	`FALSE`
`hive.vectorized.execution.reduce.enabled`	Enables vectorized mode of a query execution's reduce-side.	`TRUE`
`hive.exec.max.dynamic.partitions.pernode`	Maximum number of dynamic partitions allowed to be created in each mapper and reducer node.	100
`hive.exec.max.dynamic.partitions`	The maximum number of dynamic partitions allowed to be created in total.	1000
`hive.auto.convert.join.noconditionaltask`	Enables optimization in converting a common join into a mapjoin based on the input file size.	`TRUE`
`hive.exec.dynamic.partition.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.merge.tezfiles`	Enables the merging of small files at the end of a Tez DAG.	`FALSE`
`hive.strict.checks.cartesian.product`	Enables strict Cartesian join checks, which disallows a Cartesian product (a cross join).	`FALSE`
`hive.stats.autogather`	Enables basic statistics to be gathered automatically during the `INSERT OVERWRITE` command.	`TRUE`

Setting	Description	Value
`hive.exec.orc.split.strategy`	Expects one of [`BI`, `ETL`, `HYBRID`]. This is not a user level config. `BI` strategy is used when you want to spend less time in split generation as opposed to query execution (split generation does not read or cache file footers). `ETL` strategy is used when you want to spend more time in split generation (split generation reads and caches file footers). `HYBRID` chooses between the above strategies based on heuristics.	`HYBRID`
`hive.auto.convert.join`	Enables auto-conversion of common joins into mapjoins, based on the input file size.	`TRUE`
`hive.default.fileformat`	The default file format for `CREATE TABLE` statements. You can explicitly override this by specifying `STORED AS [FORMAT]` in your `CREATE TABLE` command.	`TEXTFILE`
`hive.exec.reducers.bytes.per.reducer`	The size per reducer. The default is 256 MB. If the input size is 1G, the job will use 4 reducers.	256000000
`hive.exec.dynamic.partition`	Enables dynamic partitions in DML/DDL.	`TRUE`
`hive.merge.size.per.task`	The size of merged files at the end of the job.	256000000
`hive.merge.mapfiles`	Enables small files to be merged at the end of a map-only job.	`TRUE`
`hive.fetch.task.conversion`	Expects one of [`NONE`, `MINIMAL`, `MORE`]. Some select queries can be converted to single `FETCH` task, minimizing latency.	`MORE`
`hive.stats.gather.num.threads`	The number of threads used by the `partialscan` and `noscan` analyze command for partitioned tables. This is applicable only for file formats that implement `StatsProvidingRecordReader` (like ORC).	10
`hive.optimize.ppd`	Enables predicate pushdown.	`TRUE`

Setting	Description	Value
`hive.input.format`	The default input format. Set to `HiveInputFormat` if you encounter problems with `CombineHiveInputFormat`.	`org.apache.hadoop.hive.ql.io.Combine`
`hive.optimize.ppd.storage`	Enables predicate pushdown to storage handlers.	`TRUE`
`hive.groupby.position.alias`	Enables using a column position alias in `GROUP BY` statements.	`FALSE`
`hive.orderby.position.alias`	Enables using a column position alias in `ORDER BY` statements.	`TRUE`
`hive.mapred.reduce.tasks.speculative.execution`	Enables speculative execution for reducers.	`TRUE`
`hive.support.quoted.identifiers`	Expects one of [`NONE`, `COLUMN`]. `NONE` implies only alphanumeric and underscore characters are valid in identifiers. `COLUMN` implies column names can contain any character.	`COLUMN`
`hive.tez.min.partition.factor`	Puts a lower limit to the number of reducers that Tez specifies when auto-reducer parallelism is enabled.	0.25
`hive.strict.checks.type.safety`	Enables strict type safety checks and disables comparing `bigint` with both `string` and `double`.	`TRUE`
`hive.log.explain.output`	Enables explain extended output for any query in your Hive log.	`true`
`hive.log.level`	The Hive logging level.	`INFO`
`tez.am.log.level`	The root logging level passed to the Tez app master.	`INFO`
`tez.task.log.level`	The root logging level passed to the Tez tasks.	`INFO`
`tez.grouping.max-size`	The upper bound on the size (in bytes) of a grouped split, to avoid generating excessively large splits.	1073741824
`tez.grouping.min-size`	The lower bound on the size (in bytes) of a grouped split, to avoid generating too many small splits.	52428800

Setting	Description	Value
`tez.shuffle-vertex-manager.min-src-fraction`	The fraction of source tasks which must complete before tasks for the current vertex are scheduled (in case of a `ScatterGather` connection).	0.25
`tez.shuffle-vertex-manager.max-src-fraction`	The fraction of source tasks that must have completed before all tasks on the current vertex can be scheduled (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.am.speculation.enabled`	Enables speculative execution of slower tasks. This can help reduce job latency when some tasks are running slower due bad or slow machines.	`FALSE`
`tez.am.dag.cleanup.on.completion`	Enables the cleanup of shuffle data upon DAG completion.	`TRUE`
`tez.client.asynchronous-stop`	Enables pushing of ATS events before terminating the Hive driver.	`FALSE`
`tez.am.sleep.time.before.exit.millis`	The amount of time after which ATS events should be pushed upon AM shutdown request.	0
`tez.yarn.ats.event.flush.timeout.millis`	The maximum amount of time for which AM should wait for events to be flushed before shutting down.	300000

Hive job examples

The following is an example of running a Hive query using the `StartJobRun` API.

```
aws emr-serverless start-job-run \
    --application-id <application_id> \
    --execution-role-arn <iam_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"
            }
        }],
        "monitoringConfiguration": {
            "s3MonitoringConfiguration": {
                "logUri": "s3://DOC-EXAMPLE-BUCKET/emr-serverless-hive/logs/"
            }
        }
    }'
```

While the job is running, the logs for the Hive driver and Tez tasks continuously upload to the Amazon S3 log location you configured in `monitoringConfiguration`.

Once the job run has a state of `SUCCEEDED`, the output of your Hive query will be available in the Amazon S3 location you specified in the `monitoringConfiguration` field of `configurationOverrides`. For example, if your log location is s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-hive/hive/logs, your Hive query's outpit will be available in s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-hive/hive/logs/applications/*<application-id>*/jobs/*<job-run-id>*/HIVE_DRIVER/stdout.gz.

The `hive-query.ql` file contains the query that Hive will run. The following is an example of a sample query.

```
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;
```

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

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 user 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. This feature is not yet available for Hive during EMR Serverless preview launch.
- 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.

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

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 EMR Serverless identity and access (p. 49).

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 employees 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 How EMR Serverless works with IAM (p. 37).

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 Identity-based policy examples for EMR Serverless (p. 45).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also 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 Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

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 When to create an IAM user (instead of a role) 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. You can temporarily assume an IAM role in the AWS Management Console by switching roles. 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 Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the *IAM User Guide*.
- **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 How IAM roles differ from resource-based policies 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.
  - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you

might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see Actions, resources, and condition keys for Amazon EMR Serverless in the *Service Authorization Reference*.

- **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 Creating 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 IAM 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 Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. 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**.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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 Creating IAM 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 Choosing 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 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 How SCPs work 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 (p. 38)	Yes
Resource-based policies (p. 38)	No
Policy actions (p. 39)	Yes
Policy resources (p. 39)	Yes
Policy condition keys (p. 40)	Yes
ACLs (p. 40)	No
ABAC (tags in policies) (p. 41)	No
Temporary credentials (p. 41)	Yes
Principal permissions (p. 41)	Yes
Service roles (p. 42)	No
Service-linked roles (p. 42)	No

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 Creating IAM 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*.

Identity-based policy examples for EMR Serverless

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless (p. 45).

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 How IAM roles differ from resource-based policies 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 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 (p. 45).

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 EMR Serverless.

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless (p. 45).

Policy condition keys for EMR Serverless

Supports service-specific policy condition keys	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 `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 EMR Serverless in the *Service Authorization Reference*.

To view examples of Amazon EMR Serverless identity-based policies, see Identity-based policy examples for EMR Serverless (p. 45).

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)	No

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.

For more information about ABAC, see What is ABAC? 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 Switching to a 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 principal permissions	Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see Actions, resources, and condition keys for Amazon EMR Serverless in the *Service Authorization Reference*.

Service roles for EMR Serverless

Supports service roles	No

Service-linked roles for EMR Serverless

Supports service-linked roles	No

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 job execution roles with EMR Serverless

Create an execution role

A job's execution role is an IAM role that grants the job permission to access AWS services and resources. You provide this role when you start a job, and EMR Serverless assumes the role when the job is invoked. Before you use the execution role, the administrator IAM role should specify that the execution role has access to the given application. The steps to invoke this API with a given execution role are listed in the following sections.

1. **Set up an execution role**

   The administrator IAM role creates an execution role to be used with a particular application. The following policy for the job execution role allows access to resource targets in Amazon S3. These permissions are necessary to monitor jobs and access logs. Replace *DOC-EXAMPLE-BUCKET-INPUT* with the name of the S3 bucket where you want EMR Serverless to retrieve input data. Replace *DOC-EXAMPLE-BUCKET-OUTPUT* with the name of the S3 bucket where you want EMR Serverless to store job output data. Replace *DOC-EXAMPLE-BUCKET-LOGGING* with the name of the S3 bucket where you want EMR Serverless to store job logs.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadFromOutputAndInputScriptBuckets",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-INPUT",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-INPUT/*",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-OUTPUT",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-OUTPUT/*"
            ]
        },
        {
            "Sid": "WriteToOutputDataBucket",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
```

```
                "s3:DeleteObject"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-OUTPUT/*"
            ]
        },
        {
            "Sid": "LoggingReadAndWriteStatement",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-LOGGING",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-LOGGING/*"
            ]
        }
    ]
}
```

With Hive, you'll need to add a policy for Glue as well. The following example Glue policy allows Create and Read access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "GlueCreateAndReadDataCatalog",
            "Effect": "Allow",
            "Action": [
                "glue:GetDatabase",
                "glue:GetDataBases",
                "glue:CreateTable",
                "glue:GetTable",
                "glue:GetTables",
                "glue:GetPartition",
                "glue:GetPartitions",
                "glue:CreatePartition",
                "glue:BatchCreatePartition",
                "glue:GetUserDefinedFunctions"
            ],
            "Resource": ["*"]
        }
    ]
}
```

2. **Create a trust policy to allow EMR Serverless to use the execution role**

   For an execution role to be used with a particular application, the IAM administrator role must update the trust policy of the execution role. This trust relationship allows EMR Serverless to assume the newly created job execution role and stream the credentials to applications running your code. The trust policy for the EMR Serverless is as follows.

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

```
        }
    }]
}
```

For more information about how to create IAM roles, see Creating IAM roles. To learn how to update your trust policy, see Modifying a role.

3. **Allow the job submitter to pass the execution role to EMR Serverless**

   After an application has been created but before jobs can start running on that application, the administrator IAM role must attach the following permissions policy to the IAM user, IAM group, or IAM role of the job submitter. This permissions policy allows the job submitter's IAM identity (an IAM user or IAM role) to submit a job on the application. This policy ensures that the `StartJobRun` action is allowed on application *<application_id>* using job execution role ARN *<iam_execution_role_arn>*.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "emr-serverless:StartJobRun",
      "Resource": "arn:aws:emr-serverless:<region>:<aws_account_id>:/
applications/<application_id>"
    },
    {
        "Effect": "Allow",
        "Action": "iam:PassRole",
        "Resource": "<iam_execution_role_arn>",
        "Condition": {
            "StringLike": {
                "iam:PassedToService": "emr-serverless.amazonaws.com"
            }
        }
    }
  ]
}
```

   After this is complete, the job submitter should be able to pass the execution role to EMR Serverless when submitting a job.

Limit execution roles

If the Amazon EMR application administrator creates a multi-tenanted EMR Serverless application, and the adminstrator IAM role onboards multiple execution roles that can be used to submit jobs by untrusted tenants, then you may want to restrict those tenants. Use either of the following options to restrict tenants from running code that gains the privileges assigned to the execution roles.

Restrict Execution Role ARN(s)

Multiple execution roles can be on-boarded to a single EMR Serverless application. If you want to control which IAM identities can use each execution role to invoke the `emr-serverless:StartJobRun` API on an EMR Serverless application, use the administrator IAM role to modify the IAM policy attached to the IAM identity you want to restrict. You can do this by restricting the resources for `iam:PassRole` action. This action accepts a list of execution role ARNs that the administrator IAM role permits for use with the application. The updated permissions policy would look like the following example.

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

```
    "Statement": [{
        "Effect": "Allow",
        "Action": "iam:PassRole",
        "Resource": [
            "<execution_role_arn_1>",
            "<execution_role_arn_2>",
            ...
        ],
        "Condition": {
            "StringLike": {
                "iam:PassedToService": "emr-serverless.amazonaws.com"
            }
        }
    }]
}
```

If you want to allow all execution roles that start with a particular prefix, such as `MyRole`, then replace the condition operator `StringEquals` with the `StringLike` operator, and replace the `execution_role_arn` value in the condition with a wildcard * character, such as `arn:aws:iam::<AWS_ACCOUNT_ID>:role/MyRole*`. All other string condition keys are also supported.

Create multiple applications

EMR Serverless provides complete network isolation for jobs running in different applications. We recommend that you run jobs that require different privileges in separate applications. Instead of creating a single multi-tenanted application with multiple execution roles with different levels of privileges, the EMR Serverless administrator can create multiple applications, isolating execution roles with similar privileges into a single application. Then, tenants can be given permissions to use only a specific application and the specific execution roles on-boarded to that application.

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.

Identity-based policy examples for EMR Serverless

By default, IAM users and roles don't have permission to create or modify Amazon EMR Serverless resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform actions on the resources that they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating IAM policies in the *IAM User Guide*.

Topics

Policy best practices

> **Note**
> EMR Serverless does not currently support managed policies, so the first practice listed below does not apply.

Identity-based policies are very powerful. They 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 using AWS managed policies** – To start using Amazon EMR Serverless quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the *IAM User Guide*.
- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the *IAM User Guide*.
- **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.
- **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition 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": "*"
        }
    ]
}
```

Access policies

For information about how to attach policies to IAM users (principals), see Managing IAM policies in the IAM User Guide.

IAM policy for full access

To grant all the required actions for EMR Serverless, create and attach a `AmazonEMRServerlessFullAccess` policy. The content of this policy statement is shown below. It reveals all the actions that Amazon EMR requires for other services.

To grant all the required actions for EMR Serverless, attach the following policy to the required IAM user, role, or group.

```
{
    "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": "*"
        }
    ]
}
```

IAM policy for read-only access

To grant read-only privileges to EMR Serverless, attach the following policy. The content of this policy statement is shown below. Wildcard characters for the `emr-serverless` element specify that only actions that begin with the specified strings are allowed. 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:ListJobRuns",
                "emr-serverless:GetJobRun"
            ],
            "Resource": "*"
        }
```

```
    ]
}
```

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 (p. 49).

The examples below demonstrate identity-based permissions policies that are used to control the actions that are allowed with EMR Serverless applications.

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"
        }
      }
    }
  ]
}
```

Troubleshooting 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 EMR Serverless

If the error message indicates 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` IAM user tries to use the console to view details about a fictional *`list-applications`* resource but does not have the `emr-serverless:`*`ListApplications`* permissions.

```
An error occurred (AccessDeniedException) when calling the ListApplications operation:
 User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: emr-
serverless:ListApplications on resource: arn:aws:emr-serverless:us-east-1:123456789012:/*
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *list-applications* operation using the `emr-serverless:ListApplications` action. For more information, see Access policies (p. 47).

I am not able to invoke `StartJobRun`

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` IAM user tries to invoke `StartJobRun` resource but does not have the `iam:PassRole` permissions.

```
arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: iam:PassRole on
 resource: arn:aws:iam::123456789012:role/JobExecutionRole
```

In this case, Mateo asks his administrator to update his policies to allow him to pass the `arn:aws:iam::123456789012:role/JobExecutionRole` role for the `iam:PassRole` action and attach it to `mateojackson` IAM user. For more information, see Using job execution roles with EMR Serverless (p. 42)

My job fails with error `could not assume execution role`

The following is an example error message that you'd see when the job execution role passed to the `StartJobRun` API is not set up properly.

```
Could not assume execution role arn:aws:iam::123456789012:user/JobExecutionRole because it
 does not exist or is not set up with the required trust relationship
```

If this happens, validate that the IAM Role specified by the `executionRoleArn` parameter passed to the `StartJobRun` API call exists in the account which is invoking the `StartJobRun` API. If the IAM Role exists, validate that the role's trust policy has been set up properly by following instructions in Using job execution roles with EMR Serverless (p. 42).

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

> **Important**
> Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys.

If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing access keys in the *IAM User Guide*.

I'm an administrator and want to allow others to access EMR Serverless

To allow others to access Amazon EMR Serverless, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in Amazon EMR Serverless.

To get started right away, see Creating your first IAM delegated user and group in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my 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 How EMR Serverless works with IAM (p. 37).
- 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 How IAM roles differ from resource-based policies in the *IAM User Guide*.

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.

Logging and monitoring

To detect incidents, receive alerts when incidents occur, and respond to them, use these options with EMR Serverless:

- Monitor Amazon EMR Serverless with AWS CloudTrail - AWS CloudTrail provides a record of actions taken by a user, role, or an AWS service in Amazon EMR Serverless. It captures calls from the EMR Serverless console and code calls to the EMR Serverless API operations as events. This allows you to 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. You can also use Athena to query CloudTrail log files for insight. For more information, see Querying AWS CloudTrail Logs and CloudTrail SerDe.

Compliance validation for Amazon EMR Serverless

Third-party auditors assess the security and compliance of Amazon EMR Serverless as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

To learn whether Amazon EMR or other AWS services are in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- Architecting for HIPAA Security and Compliance Whitepaper – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

  Note
  Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that 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 Serverless is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Amazon EMR Serverless through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). 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 (p. 52)
- Shared Responsibility Model
- Amazon Web Services: Overview of Security Processes (whitepaper)

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 (p. 48).

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	Yes	`CreateApplication`

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 do not propagate to job runs submitted to that application.		
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
```

Logging

To monitor your job progress on EMR Serverless and to troubleshoot failures, you must configure your jobs to send log information to Amazon S3. This section describes how to configure logging during job submission and how to launch the Spark History and Hive Application Timeline servers locally. You'll use Docker to view logs on the Spark and Tez interfaces after your job has completed.

Configure a job run to use Amazon S3 logs

Before your jobs can send log data to Amazon S3, you must include the following permissions in the permissions policy for the job execution role. Replace *DOC-EXAMPLE-BUCKET-LOGGING* with the name of your logging bucket.

Note
For EMR Serverless preview, ensure that you create *DOC-EXAMPLE-BUCKET-LOGGING* in the `us-east-1` region. If the bucket is created in another region, your job run will fail.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET-LOGGING/*"
            ]
        }
    ]
}
```

To set up an Amazon S3 bucket to store logs, use the `s3MonitoringConfiguration` configuration when starting a job run. This can be done by providing the following `--configuration-overides` configuration.

```
{
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://DOC-EXAMPLE-BUCKET-LOGGING/logs/"
        }
    }
}
```

After you've given your job execution role the permissions to send logs to Amazon S3 and have submitted a job with `s3MonitoringConfiguration`, you can find your logs in `s3://`*DOC-EXAMPLE-BUCKET-LOGGING*`/logs/applications/`*application_id*`/jobs/`*job_id*. You'll find your driver, executor, and event logs in the indicated subfolders.

For Spark, you can find logs in the following folders:

- Driver logs - /SPARK_DRIVER
- Executor logs - /SPARK_EXECUTOR
- Spark event logs - /sparklogs

For Hive, you can find logs in the following folders:

- Driver logs - /HIVE_DRIVER
- Executor logs - /TEZ_TASK
- Event logs - /ytslogs

Start the Spark history server and view the Spark UI locally using Docker

Prerequisites

1. You must have the `docker` command installed locally. For information about how to install Docker, see the Docker Engine community.
2. Run a Spark job that produces Spark Event logs by following the steps in Configure a job run to use Amazon S3 logs (p. 57).

After completing the prerequisites, follow the steps below.

1. Save both `pom.xml` and `Dockerfile` locally.

   In the `pom.xml` file, update the AWS SDK version to match your Amazon EMR release version. Your `pom.xml` file might look like the following example.

```
<project
        xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.amazonaws</groupId>
    <artifactId>EMRServerlessSparkHistoryServer</artifactId>
    <packaging>jar</packaging>
    <version>2.0-SNAPSHOT</version>
    <name>EMRServerlessSparkHistoryServer</name>
    <url>http://maven.apache.org</url>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <jdk.version>1.8</jdk.version>
        <hadoop.version>3.2.1</hadoop.version>
        <awssdk.version>1.11.977</awssdk.version>
        <httpclient.version>4.5.13</httpclient.version>
        <jackson.version>2.10.5</jackson.version>
        <jackson.databind.version>2.10.5.1</jackson.databind.version>
    </properties>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.amazonaws</groupId>
                <artifactId>aws-java-sdk-bom</artifactId>
                <version>${awssdk.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
```

```
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-java-sdk-s3</artifactId>
        </dependency>
    </dependencies>
</project>
```

In the `Dockerfile`, update the second FROM line with the Amazon EMR release and Region that you are using. To choose your base image URI, see How to select a base image URI. The following `Dockerfile` is a sample that you should modify to meet your requirements.

```
FROM maven:3.6-amazoncorretto-8
FROM 755674844232.dkr.ecr.us-east-1.amazonaws.com/spark/emr-6.3.0
USER root
WORKDIR /tmp/
ADD pom.xml /tmp
COPY --from=0 /usr/share/maven /usr/share/maven
RUN /usr/share/maven/bin/mvn dependency:tree
RUN /usr/share/maven/bin/mvn dependency:copy-dependencies -DoutputDirectory=/usr/lib/
spark/jars/
RUN mkdir /mnt/s3 \
    && chown spark:spark /mnt/s3
USER spark:spark
ENV SPARK_NO_DAEMONIZE=true
ENTRYPOINT [ "/usr/lib/spark/sbin/start-history-server.sh" ]
```

2. Build the Docker image using the files in the local directory, using the name `emr/sparkui`.

```
docker build -t emr/sparkui .
```

3. Define your Amazon S3 log location as an environment variable.

```
LOG_DIR="s3://DOC-EXAMPLE-BUCKET/logs/"
```

4. Define AWS access credentials as environment variables and define the Region you're running your job in.

```
export AWS_ACCESS_KEY_ID=AKIAAAABBBCCCDDD
export AWS_SECRET_ACCESS_KEY=abcd1234
export AWS_REGION=us-west-2
```

5. Create and start the Docker container. In the following commands, use the values obtained in the previous step.

```
docker run --rm \
 --user spark \
 -p 18080:18080 \
 -e SPARK_HISTORY_OPTS="-Dspark.history.fs.logDirectory=$LOG_DIR -
Dspark.hadoop.fs.s3.customAWSCredentialsProvider=com.amazonaws.auth.DefaultAWSCredentialsProviderCh
 \
 -e AWS_REGION -e AWS_ACCESS_KEY_ID -e AWS_SECRET_ACCESS_KEY \
 emr/sparkui
```

6. Open `http://localhost:18080` in your browser to view the Spark UI locally.

Start the application timeline server and view the Tez UI locally using Docker

Prerequisites

1. You must have the `docker` command installed locally. For information about how to install Docker, see the Docker Engine community.
2. Run a Hive job that produces Hive Event logs by following the steps in Configure a job run to use Amazon S3 logs (p. 57).

After completing the prerequisites, follow the steps below.

1. Save `pom.xml`, `Dockerfile` , `entrypoint.sh`, `event-log-sync.sh`, `hadoop-layout.sh`, `tez-ui.patch`, and `yarn-site.xml` locally.

   Your `pom.xml` file might look like the following example.

```
<project
        xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.amazonaws</groupId>
    <artifactId>TezUI</artifactId>
    <packaging>jar</packaging>
    <version>2.0-SNAPSHOT</version>
    <name>TezUI</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>joda-time</groupId>
            <artifactId>joda-time</artifactId>
            <version>2.9.3</version>
            <exclusions>
                <exclusion>
                    <groupId>*</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache.tez</groupId>
            <artifactId>tez-yarn-timeline-cache-plugin</artifactId>
            <version>0.10.0</version>
            <exclusions>
                <exclusion>
                    <groupId>*</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-runner</artifactId>
            <version>9.3.27.v20190418</version>
            <exclusions>
                <exclusion>
                    <groupId>*</groupId>
```

```
                    <artifactId>*</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
</project>
```

The following `Dockerfile` is a sample that you should modify to meet your requirements. Make sure that ports 8088 (Hadoop Resource Manager), 8188 (Hadoop Application Timeline Server), 9999 (Tez UI) are not occupied.

```
FROM amazonlinux:2
FROM amazoncorretto:8
FROM maven:3.6-amazoncorretto-8

RUN yum install -y procps awscli rsync

WORKDIR /tmp/
ENV ENTRYPOINT /usr/bin/entrypoint.sh
ENV TEZ_HOME /hadoop/usr/lib/tez
ENV YARN_HOME /hadoop/usr/lib/hadoop-yarn
ENV HADOOP_HOME /hadoop/usr/lib/hadoop
ENV HDFS_HOME /hadoop/usr/lib/hadoop-hdfs
ENV TEZ_HOME /hadoop/usr/lib/tez
ENV HADOOP_CONF /hadoop/etc/hadoop/conf

RUN curl -o ./apache-tez-0.9.2-bin.tar.gz https://archive.apache.org/dist/tez/0.9.2/
apache-tez-0.9.2-bin.tar.gz && \
    curl -o  ./hadoop-2.10.1.tar.gz https://archive.apache.org/dist/hadoop/common/
hadoop-2.10.1/hadoop-2.10.1.tar.gz && \
    tar -xzf hadoop-2.10.1.tar.gz && \
    tar -xzf apache-tez-0.9.2-bin.tar.gz

RUN mkdir -p $HADOOP_HOME/lib && \
    mkdir -p $TEZ_HOME && \
    mkdir -p $HADOOP_CONF && \
    mkdir -p $YARN_HOME && \
    mkdir -p $HDFS_HOME &&\
    mkdir -p /tmp/tez-ui


COPY hadoop-layout.sh $HADOOP_HOME/libexec/hadoop-layout.sh
COPY yarn-site.xml .
COPY pom.xml .

RUN mvn dependency:copy-dependencies -DoutputDirectory=/tmp/tez-ui/ && \
    cp /tmp/tez-ui/joda-time-2.9.3.jar $HADOOP_HOME/lib/ && \
    cp /tmp/tez-ui/jetty-runner-*.jar $TEZ_HOME && \
    cp /tmp/tez-ui/tez-yarn-timeline-cache-plugin*.jar $TEZ_HOME

COPY event-log-sync.sh .
COPY entrypoint.sh /usr/bin/entrypoint.sh
COPY tez-ui.patch /tmp/

RUN yum install -y patch

RUN chmod 744 $ENTRYPOINT

ENTRYPOINT [ "/usr/bin/entrypoint.sh" ]
EXPOSE 8088
EXPOSE 8188
EXPOSE 9999
```

```
CMD tail -f /dev/null
```

Your `yarn-site.xml` should look like the following.

```
<?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>
    <property>
        <name>yarn.timeline-service.hostname</name>
        <value>0.0.0.0</value>
    </property>
    <property>
        <name>yarn.timeline-service.bind-host</name>
        <value>0.0.0.0</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>0.0.0.0</value>
    </property>
    <property>
        <name>yarn.resourcemanager.bind-host</name>
        <value>0.0.0.0</value>
    </property>
    <property>
        <name>yarn.timeline-service.http-cross-origin.enabled</name>
        <value>true</value>
    </property>
    <property>
        <name>yarn.resourcemanager.http-cross-origin.enabled</name>
        <value>true</value>
    </property>
    <property>
        <name>yarn.timeline-service.enabled</name>
        <value>true</value>
    </property>
    <property>
        <name>yarn.timeline-service.version</name>
        <value>1.5</value>
    </property>
    <property>
        <name>yarn.timeline-service.store-class</name>
        <value>org.apache.hadoop.yarn.server.timeline.EntityGroupFSTimelineStore</
value>
    </property>
    <property>
        <name>APPLICATION_ID</name>
        <value>APPLICATION_ID</value>
    </property>
    <property>
        <name>JOB_RUN_ID</name>
        <value>JOB_RUN_ID</value>
    </property>
```

```
    <property>
        <name>yarn.timeline-service.entity-group-fs-store.active-dir</name>
        <value>file:////tmp/timeline-data/${JOB_RUN_ID}/active</value>
    </property>
    <property>
        <name>yarn.timeline-service.entity-group-fs-store.done-dir</name>
        <value>file:////tmp/timeline-data/${JOB_RUN_ID}/done</value>
    </property>
    <property>
        <name>yarn.timeline-service.entity-group-fs-store.group-id-plugin-classes</
name>
        <value>org.apache.tez.dag.history.logging.ats.TimelineCachePluginImpl</value>
    </property>
    <property>
        <name>yarn.timeline-service.entity-group-fs-store.summary-store</name>
        <value>org.apache.hadoop.yarn.server.timeline.RollingLevelDBTimelineStore</
value>
    </property>
    <property>
        <name>yarn.timeline-service.ttl-enable</name>
        <value>false</value>
    </property>
    <property>
        <name>yarn.timeline-service.entity-group-fs-store.scan-interval-seconds</name>
        <value>10</value>
    </property>
</configuration>
```

Your `hadoop-layout.sh` file should look like the following.

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
HADOOP_COMMON_DIR="./"
HADOOP_COMMON_LIB_JARS_DIR="lib"
HADOOP_COMMON_LIB_NATIVE_DIR="lib/native"
HDFS_DIR="./"
HDFS_LIB_JARS_DIR="lib"
YARN_DIR="./"
YARN_LIB_JARS_DIR="lib"
MAPRED_DIR="./"
MAPRED_LIB_JARS_DIR="lib"

HADOOP_LIBEXEC_DIR=${HADOOP_LIBEXEC_DIR:-"/usr/lib/hadoop/libexec"}
HADOOP_CONF_DIR=${HADOOP_CONF_DIR:-"/etc/hadoop/conf"}
HADOOP_COMMON_HOME=${HADOOP_COMMON_HOME:-"/usr/lib/hadoop"}
HADOOP_HDFS_HOME=${HADOOP_HDFS_HOME:-"/usr/lib/hadoop-hdfs"}
HADOOP_MAPRED_HOME=${HADOOP_MAPRED_HOME:-"/usr/lib/hadoop-mapreduce"}
HADOOP_YARN_HOME=${HADOOP_YARN_HOME:-"/usr/lib/hadoop-yarn"}
```

Your `event-log-sync.sh` should look like the following. Timeline data is regularly downloaded to a container's local disk using this script.

```
job_path=/tmp/timeline-data/$JOB_RUN_ID
mkdir -p $job_path/active
mkdir -p $job_path/done

while [[ true ]]; do
 aws s3 sync $S3_LOG_URI/applications/$APPLICATION_ID/jobs/$JOB_RUN_ID/ytslogs/active
 $job_path/active  --exclude '*$folder$*'
  # Hack to move done events to active folder as ATS doesnt read done path
  aws s3 sync $S3_LOG_URI/applications/$APPLICATION_ID/jobs/$JOB_RUN_ID/ytslogs/done
 $job_path/done_events/  --exclude '*$folder$*'
  if [ -d "$job_path/done_events/*/*/*/*" ]; then
     rsync -r $job_path/done_events/*/*/*/* $job_path/active/
  fi
  echo " `date +%s` sleeping for 30 seconds"
  sleep 30s
done
```

Your `tez-ui.patch` should look like the following.

```
--- assets/tez-ui.js
+++ assets/tez-ui.js
@@ -7011,6 +7011,11 @@

       if (logURL) {
         if (logURL.indexOf("://") === -1) {
+   if (this.get("env.app.AWS_CONSOLE_BASE_PATH_URL")) {
+  var syslog = "syslog_";
+  var _attemptID = syslog.concat(this.get("entityID")).concat(".gz");
+  return [this.get("env.app.AWS_CONSOLE_BASE_PATH_URL"),
 logURL].join("/").concat(_attemptID);
+   }
           var attemptID = this.get("entityID"),
               yarnProtocol = this.get('env.app.yarnProtocol');
           return yarnProtocol + '://' + logURL + '/syslog_' + attemptID;
@@ -10080,7 +10085,10 @@
         yarnProtocol = this.get('env.app.yarnProtocol');

     if (logURL && logURL.indexOf("://") === -1) {
-      return yarnProtocol + '://' + logURL;
+     if (this.get("env.app.AWS_CONSOLE_BASE_PATH_URL")) {
+ return [this.get("env.app.AWS_CONSOLE_BASE_PATH_URL"), logURL].join("/");
+     }
+     return yarnProtocol + '://' + logURL;
     }
   }

@@ -10257,7 +10265,9 @@
     for (var key in otherinfo) {
       if (key.indexOf('inProgressLogsURL_') === 0) {
         var logs = _ember['default'].get(source, 'otherinfo.' + key);
-        if (logs.indexOf('http') !== 0) {
+        if (this.get("env.app.AWS_CONSOLE_BASE_PATH_URL")) {
+                logs = [this.get("env.app.AWS_CONSOLE_BASE_PATH_URL"),
 logs].join("/");
+        } else if (logs.indexOf('http') !== 0) {
           logs = 'http://' + logs;
         }
         var attemptID = key.substring(18);
```

Your `entrypoint.sh` should look like the following.

```
#!/bin/bash


if [ -z "$JOB_RUN_ID" ]; then
  echo "JOB_RUN_ID is not set"
  exit
fi

if [ -z "$APPLICATION_ID" ]; then
  echo "APPLICATION_ID is not set"
  exit
fi

if [ -z "$S3_LOG_URI" ]; then
  echo "S3_LOG_URI is not set"
  exit
fi

function cpp(){
 [ -d $2 ] || mkdir $2
 cp -r $1 $2
}

cpp "hadoop-2.10.1/share/hadoop/hdfs/*.jar" /hadoop/usr/lib/hadoop-hdfs/
cpp "hadoop-2.10.1/share/hadoop/hdfs/*.jar" /hadoop/usr/lib/hadoop-hdfs/
cpp "hadoop-2.10.1/share/hadoop/hdfs/lib/*.jar" /hadoop/usr/lib/hadoop-hdfs/
cpp "hadoop-2.10.1/share/hadoop/common/lib/*.jar" /hadoop/usr/lib/hadoop/
cpp "hadoop-2.10.1/share/hadoop/common/*.jar" /hadoop/usr/lib/hadoop/
cpp "hadoop-2.10.1/share/hadoop/yarn/lib/*.jar" /hadoop/usr/lib/hadoop-yarn
cpp "hadoop-2.10.1/share/hadoop/yarn/*.jar" /hadoop/usr/lib/hadoop-yarn
cpp hadoop-2.10.1/share/hadoop/yarn/timelineservice/ /hadoop/usr/lib/hadoop-yarn/

cpp "apache-tez-0.9.2-bin/*" /hadoop/usr/lib/tez/
rm -rf $TEZ_HOME/lib/slf4j-log4j12-*

cpp hadoop-2.10.1/bin/yarn /hadoop/usr/lib/hadoop-yarn/bin

cp hadoop-2.10.1/etc/hadoop/* /hadoop/etc/hadoop/conf/
cp yarn-site.xml /hadoop/etc/hadoop/conf/
cpp hadoop-2.10.1/sbin/yarn-daemon.sh /hadoop/usr/lib/hadoop-yarn/sbin/
cpp hadoop-2.10.1/libexec /hadoop/usr/lib/hadoop/


bash event-log-sync.sh > event-log-sync.log &

export HADOOP_BASE_PATH=/hadoop
export HADOOP_COMMON_HOME=$HADOOP_BASE_PATH/usr/lib/hadoop
export HADOOP_LIBEXEC_DIR=$HADOOP_BASE_PATH/usr/lib/hadoop/libexec
export HADOOP_YARN_HOME=$HADOOP_BASE_PATH/usr/lib/hadoop-yarn
export HADOOP_HDFS_HOME=$HADOOP_BASE_PATH/usr/lib/hadoop-hdfs
export HADOOP_MAPRED_HOME=$HADOOP_BASE_PATH/usr/lib/hadoop
export HADOOP_CONF_DIR=$HADOOP_BASE_PATH/etc/hadoop/conf
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$HADOOP_BASE_PATH/usr/lib/tez/*:
$HADOOP_BASE_PATH/usr/lib/tez/lib/*:$HADOOP_BASE_PATH/usr/share/aws/aws-java-sdk/*
export TEZ_HOME=$HADOOP_BASE_PATH/usr/lib/tez
export PATH=$HADOOP_CLASSPATH:$PATH:$HADOOP_BASE_PATH/usr/lib/hadoop/bin
export USER=`id -u -n`

t=JOB_RUN_ID
sed -i "s#<value>$t</value>#<value>$JOB_RUN_ID</value>#" /hadoop/etc/hadoop/conf/yarn-
site.xml
t=APPLICATION_ID
sed -i "s#<value>$t</value>#<value>$APPLICATION_ID</value>#" /hadoop/etc/hadoop/conf/
yarn-site.xml
```

```
bash $HADOOP_BASE_PATH/usr/lib/hadoop-yarn/sbin/yarn-daemon.sh start resourcemanager
sleep 5s
bash $HADOOP_BASE_PATH/usr/lib/hadoop-yarn/sbin/yarn-daemon.sh start timelineserver

rm -rf hadoop-2.10.1* apache-tez-0.9.2-bin*

mkdir /tmp/tez-ui-files/ && cd /tmp/tez-ui-files/
jar -xvf $TEZ_HOME/tez-ui*.war
patch -p0 < /tmp/tez-ui.patch
sed -i "57 i AWS_CONSOLE_BASE_PATH_URL: \"$AWS_CONSOLE_BASE_PATH_URL\"," config/
configs.env

mkdir -p /hadoop/usr/lib/tez/logs/
java -jar $TEZ_HOME/jetty-runner-*.jar --port 9999 --path /tez-ui/ /tmp/tez-ui-files/ >
 $TEZ_HOME/logs/tez-ui.log &

echo "********************************************************************"
echo "Launching Tez UI. Access it using: http://localhost:9999/tez-ui/"
echo "********************************************************************"

tail -f /dev/null
```

2. Build the Docker image using the files in the local directory, using the name `emr/tezui`.

```
docker build -t emr/tezui .
```

3. Define your Amazon S3 log locations as environment variables.

```
export SERVERLESS_EMR_S3_LOG_BUCKET=DOC-EXAMPLE-BUCKET
export CONTAINER_LOG_BASE_PATH=logs
export S3_LOG_URI=s3://$SERVERLESS_EMR_S3_LOG_BUCKET/$CONTAINER_LOG_BASE_PATH
```

4. Define AWS access credentials as environment variables, and define the Region you're running your job in.

```
export AWS_ACCESS_KEY_ID=AKIAAAABBBCCCDDD
export AWS_SECRET_ACCESS_KEY=abcd1234
export AWS_REGION=us-east-1
```

5. Define the application ID and job run ID that you want to monitor. EMR Serverless currently support one job per Docker container.

```
export APPLICATION_ID=<application_id>
export JOB_RUN_ID=<job_run_id>
export AWS_CONSOLE_BASE_PATH_URL="https://s3.console.aws.amazon.com/s3/buckets/
${SERVERLESS_EMR_S3_LOG_BUCKET}?prefix=${CONTAINER_LOG_BASE_PATH}/applications/
$<application_id>/jobs/$<job_run_id>"
```

6. Create and start the Docker container. In the following command, use the values obtained in the previous step.

```
docker run -p 8088:8088 -p 8188:8188 -p 9999:9999 \
  -e AWS_ACCESS_KEY_ID -e AWS_SECRET_ACCESS_KEY -e AWS_REGION \
  -e S3_LOG_URI -e JOB_RUN_ID -e APPLICATION_ID -e AWS_CONSOLE_BASE_PATH_URL -it \
emr/tezui
```

7. Open `http://localhost:9999/tez-ui/` in your browser to view the Tez UI locally.
8. To view the next job run, you can stop the container and repeat **step 5** and **step 6**, providing the next job's ID. If you do so, remember to redefine your Amazon S3 log location if your S3 path contains a job ID.

Limitations

These are the limitations while EMR Serverless is in preview:

- You can have up to 10 running applications per account.
- You can have up to 100 active workers per application.
- You can request up to 30 GB of memory for workers.
- A job can run up to three hours.
- A job can access limited AWS service endpoints, including Amazon S3, AWS Glue, Amazon DynamoDB, Amazon CloudWatch, KMS and Secrets Manager within the same region.
- Access to internet services or AWS services hosted in your own VPC are currently not supported.

Release versions

An Amazon EMR release is a set of open-source applications from the big data ecosystem. Each release comprises different big data applications, components, and features that you select to have EMR Serverless deploy and configure when you run your job.

Beginning with Amazon EMR version 6.4.0, you can deploy EMR Serverless. This deployment option is not available with earlier Amazon EMR release versions. You must specify a supported release version when you submit your job.

EMR Serverless uses the following form of release label: emr-x.x.x-latest or emr-x.x.x-yyyymmdd with a specific release date. For example, emr-6.5.0-latest or emr-6.2.0-20210920. Using -latest ensures that your Amazon EMR version always includes the latest security updates.

Topics

Apache Hive

The following Amazon EMR releases are available for Amazon EMR Serverless Hive applications:

- emr-6.5.0-preview

Release notes for Amazon EMR 6.5.0

- Hive supports `hive-site`, `tez-site`, `emrfs-site`, and `core-site` classifications for EMR Serverless.

Classifications	Descriptions
`hive-site`	Change values in Hive's `hive-site.xml` file.
`tez-site`	Change values in Tez's `tez-site.xml` file.
`emrfs-site`	Change EMRFS settings.
`core-site`	Change values in Hadoop's `core-site.xml` file.

Configuration classifications allow you to customize applications. These often correspond to a configuration XML file for the application, such as `core-site.xml`. For more information, see Configure applications.

emr-6.5.0-preview (Hive 3.1.2)

Regions: `emr-6.5.0-preview` is available in all Regions supported by Amazon EMR Serverless.

Apache Spark

The following Amazon EMR releases are available for Amazon EMR Serverless Spark applications:

- emr-6.5.0-preview

Release notes for Amazon EMR 6.5.0

- Spark supports the `spark-defaults` classification for EMR Serverless.

Classifications	Descriptions
`spark-defaults`	Change values in Spark's `spark-defaults.conf` file.

Configuration classifications allow you to customize applications. These often correspond to a configuration XML file for the application, such as `spark-defaults.xml`. For more information, see Configure applications.

emr-6.5.0-preview (Spark 3.1.2)

Regions: `emr-6.5.0-preview` is available in all Regions supported by Amazon EMR Serverless.