Introduction to SageMaker

- How ML engineers get most out of AWS
- Getting started with SageMaker & SageMaker Studio
- Onboarding with SageMaker studio
- Adding a user

How ML engineers can get the most out of AWS

There are many services and capabilities in the AWS platform that an ML engineer can choose from. Professionals who are already familiar with using virtual machines can easily spin up EC2 instances and run ML experiments using deep learning frameworks inside these virtual private servers. Services such as AWS Glue, Amazon EMR, and AWS Athena can be utilized by ML engineers and data engineers for different data management and processing needs. Once the ML models need to be deployed into dedicated inference endpoints, a variety of options become available:

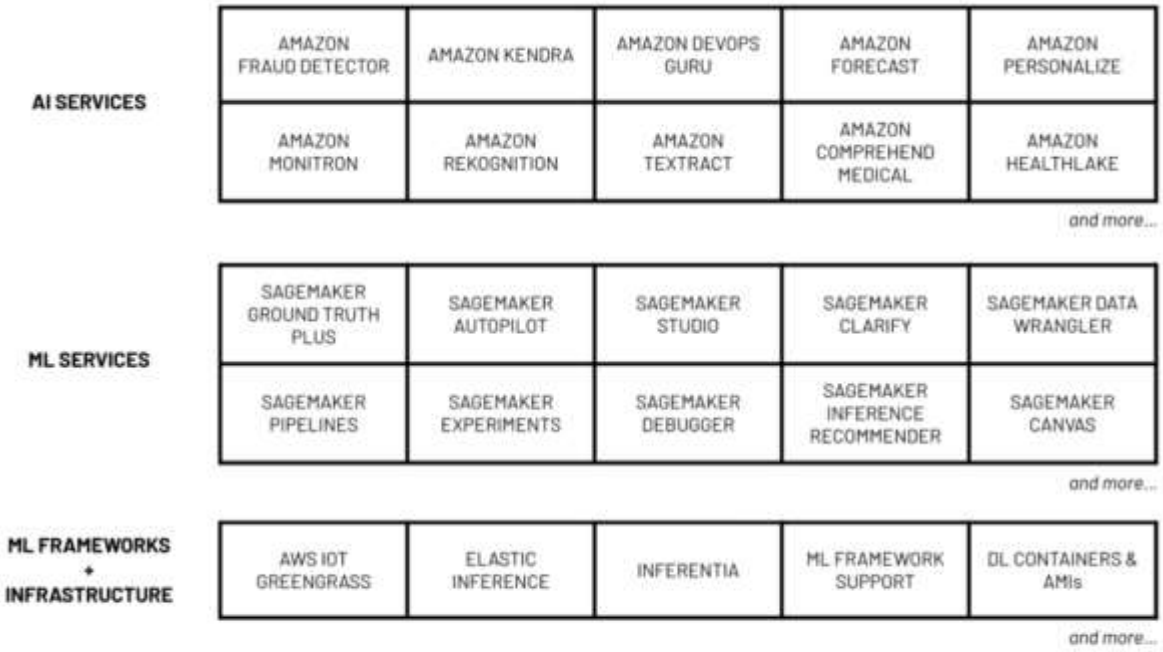


Figure 1.2 – AWS machine learning stack

As shown in the preceding diagram, data scientists, developers, and ML engineers can make use of multiple services and capabilities from the AWS machine learning stack. The services

grouped under AI services can easily be used by developers with minimal ML experience. To use the services listed here, all we need would be some experience working with data, along with the software development skills required to use SDKs and APIs. If we want to quickly build ML-powered applications with features such as language translation, text-to-speech, and product recommendation, then we can easily do that using the services under the AI Services bucket. In the middle, we have ML services and their capabilities, which help solve the more custom ML requirements of data scientists and ML engineers. To use the services and capabilities listed here, a solid understanding of the ML process is needed. The last layer, ML frameworks and infrastructure, offers the highest level of flexibility and customizability as this includes the ML infrastructure and framework support needed by more advanced use cases.

So, how can ML engineers make the most out of the AWS machine learning stack? The ability of ML engineers to design, build, and manage ML systems improves as they become more familiar with the services, capabilities, and tools available in the AWS platform. They may start with AI services to quickly build AI-powered applications on AWS. Over time, these ML engineers will make use of the different services, capabilities, and infrastructure from the lower two layers as they become more comfortable dealing with intermediate ML engineering requirements.

AI Services:

- **Amazon Fraud Detector:** Detects potentially fraudulent online activities.
- **Amazon Kendra:** A highly accurate and easy-to-use enterprise search service.
- **Amazon DevOps Guru:** Provides ML-powered insights to improve application availability.
- **Amazon Forecast:** Creates accurate time series forecasts.
- **Amazon Personalize:** Adds real-time personalization to applications.
- **Amazon Monitron:** Uses machine learning to detect anomalies in industrial equipment.
- **Amazon Rekognition:** Image and video analysis service.
- **Amazon Textract:** Extracts text and data from documents.
- **Amazon Comprehend Medical:** Analyzes and extracts health data from medical text.
- **Amazon HealthLake:** Stores, transforms, queries, and analyzes health data.

ML Services:

- **SageMaker Ground Truth Plus:** Provides data labeling services.
- **SageMaker Autopilot:** Automates machine learning model creation.
- **SageMaker Studio:** Integrated development environment (IDE) for ML.
- **SageMaker Clarify:** Detects bias in ML models.
- **SageMaker Data Wrangler:** Prepares data for machine learning.
- **SageMaker Pipelines:** Manages and automates ML workflows.
- **SageMaker Experiments:** Tracks, organizes, and compares ML experiments.
- **SageMaker Debugger:** Debugs and monitors ML models.
- **SageMaker Inference Recommender:** Helps choose the best ML instance type.
- **SageMaker Canvas:** Enables business analysts to build ML models without coding.

ML Frameworks + Infrastructure:

- **AWS IoT Greengrass:** Runs local compute, messaging, and data caching for connected devices.
- **Elastic Inference:** Provides lower-cost inference acceleration.
- **Inferentia:** AWS's custom machine learning inference chip.
- **ML Framework Support:** Supports popular ML frameworks.
- **DL Containers & AMIs:** Provides pre-configured environments for ML and deep learning.

Getting started with SageMaker and SageMaker Studio

When performing ML and ML engineering on AWS, professionals should consider using one or more of the capabilities and features of Amazon SageMaker. If this is your first time learning about SageMaker, it is a fully managed ML service that helps significantly speed up the process of preparing, training, evaluating, and deploying ML models.

If you are wondering what these capabilities are, check out some of the capabilities tagged under ML SERVICES in Figure 1.2 from the How ML engineers can get the most out of AWS section. We will tackle several capabilities of SageMaker as we go through the different chapters of this book. In the meantime, we will start with SageMaker Studio as we will need to set it up first before we work on the SageMaker Canvas and SageMaker Autopilot examples.

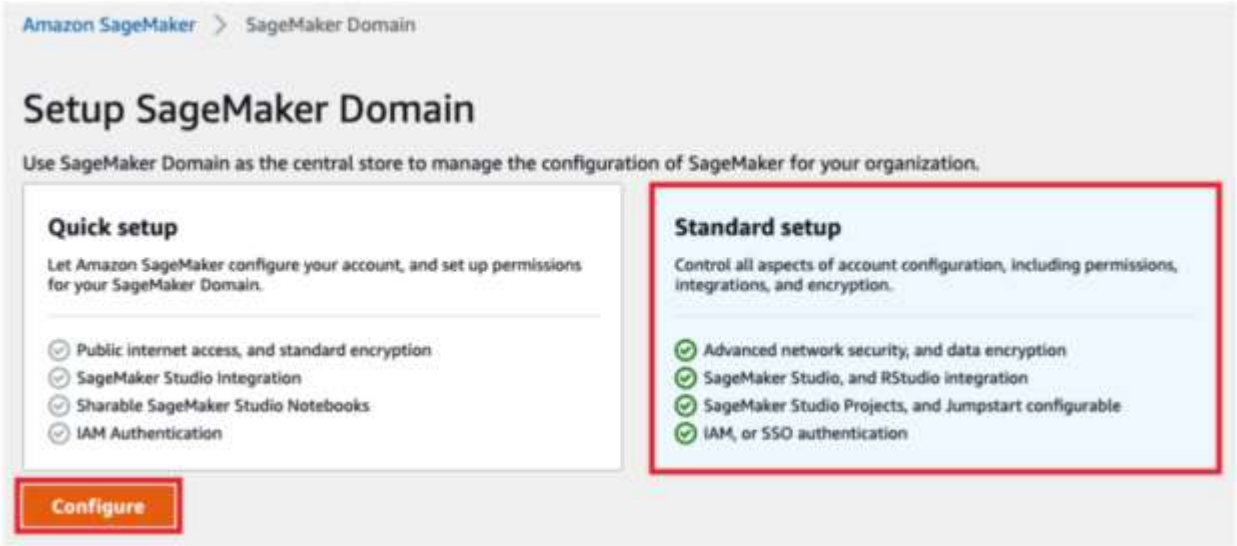
Onboarding with SageMaker Studio

SageMaker Studio provides a feature-rich IDE for ML practitioners. One of the great things about SageMaker Studio is its tight integration with the other capabilities of SageMaker, which allows us to manage different SageMaker resources by just using the interface.

For us to have a good idea of what it looks like and how it works, let's proceed with setting up and configuring SageMaker Studio:

1. In the search bar of the AWS console, type sagemaker studio. Select SageMaker Studio under Features.

2. Choose Standard setup, as shown in the following screenshot:



As we can see, Standard setup should give us more configuration options to tweak over Quick setup. Before clicking the Configure button, make sure that you are using the same region where the S3 bucket and training and test datasets are located.

3. Under Authentication, select AWS Identity and Access Management (IAM). For the default execution role under Permission, choose Create a new role. Choose Any S3 bucket. Then, click Create role.

4. Under Network and Storage Section, select the default VPC and choose a subnet (for example, us-west -2a), similar to what is shown in the following screen- shot:

Here, we have also configured the SageMaker Domain to use the default SageMaker internet access by selecting Public Internet Only. Under Encryption key, we leave this unchanged by choosing No Custom Encryption. Review the configuration and then click Next.

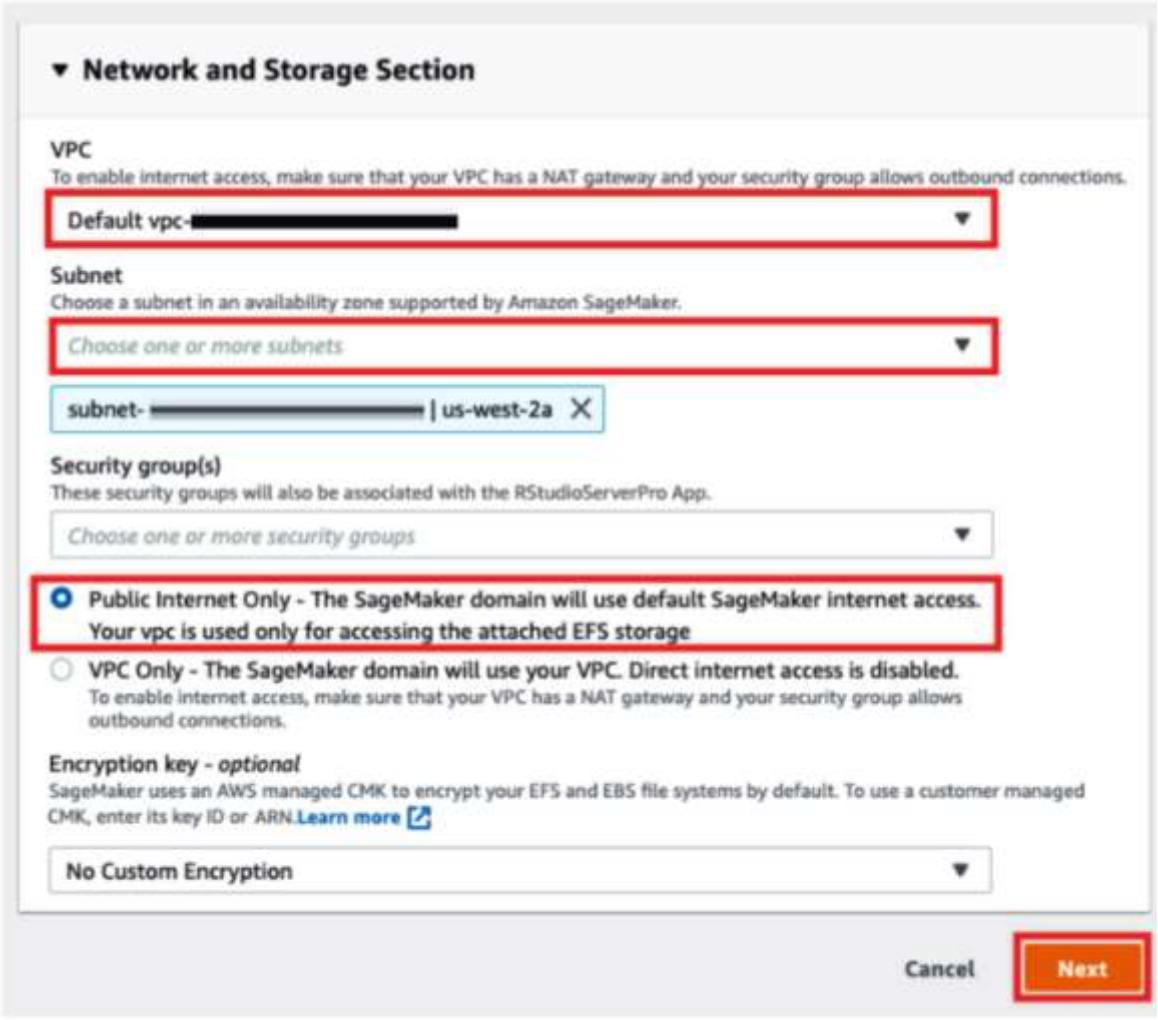


Figure 1.14 – Network and Storage Section

5. Under Studio settings, leave everything as-is and click Next.

6. Similarly, under General settings | RStudio Work- bench, click Submit.

Once you have completed these steps, you should see the Preparing SageMaker Domain loading message. This step should take around 3 to 5 minutes to complete. Once complete, you should see a notification stating The SageMaker Domain is ready.

Adding a user to an existing SageMaker Domain

Now that our SageMaker Domain is ready, let's create a user. Creating a user is straightforward. So, let's begin:

1. On the SageMaker Domain/Control Panel page, click Add user.

2. Specify the name of the user under Name. Under Default execution role, select the execution role that you created in the previous step. Click Next.

3. Under Studio settings | SageMaker Projects and JumpStart, click Next.

4. Under RStudio settings | Rstudio Workbench, click Submit.

AutoML with SageMaker Autopilot

SageMaker Autopilot allows ML practitioners to build high -quality ML models without having to write a single line of code. Of course, it is possible to programmatically configure, run, and manage SageMaker Autopilot experiments using the SageMaker Python SDK. Let's see what happens behind the scenes:

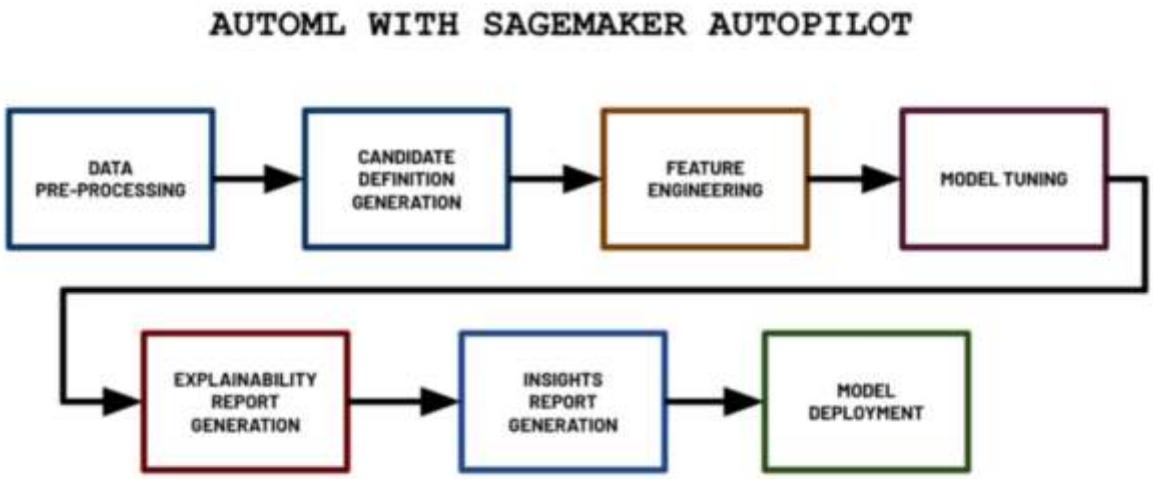


Figure 1.21 – AutoML with SageMaker Autopilot

In the preceding diagram, we can see the different steps that are performed by SageMaker Autopilot when we run the AutoML experiment. It starts with the data pre-processing step and proceeds with the generation of candidate models (pipeline and algorithm pair) step. Then, it continues to perform the feature engineering and model tuning steps, which would yield multiple trained models from different model families, hyperparameter values, and model performance metric values. The generated model with the best performance metric values is tagged as the "best model" by the Autopilot job. Next, two reports are generated: the explainability report and the insights report. Finally, the model is deployed to an inference endpoint.

Let's dive a bit deeper into what is happening in each step:

1. **Data pre-processing:** Data is cleaned automatically and missing values are automatically imputed.
2. **Candidate definition generation:** Multiple "candidate definitions" (composed of a data processing job and a training job) are generated, all of which will be used on the dataset.
3. **Feature engineering:** Here, data transformations are applied to perform automated feature engineering.
4. **Model tuning:** The Automatic Model Tuning (hyperparameter tuning) capability of SageMaker is used to generate multiple models using a variety of hyperparameter configuration values to find the "best model."
5. **Explainability report generation:** The model explainability report, which makes use of SHAP values to help explain the behavior of the generated model, is generated using tools provided by SageMaker Clarify (an- other capability of SageMaker focused on AI fairness and explainability). We'll dive a bit deeper into this topic later in Chapter 9, Security, Governance, and Compliance Strategies.
6. **Insights report generation:** The insights report, which includes data insights such as scalar metrics, which help us understand our dataset better, is generated.
7. **Model deployment:** The best model is deployed to a dedicated inference endpoint. Here, the value of the objective metric is used to determine which is the best model out of all the models trained during the model tuning step.

Deep Learning AMIs

Getting started with Deep Learning AMIs

Before we talk about DLAMIS, we must have a good idea of what AMIS are. We can think of an AMI as the "DNA" of an organism. Using this analogy, the organism would correspond and map to one or more EC2 instances:

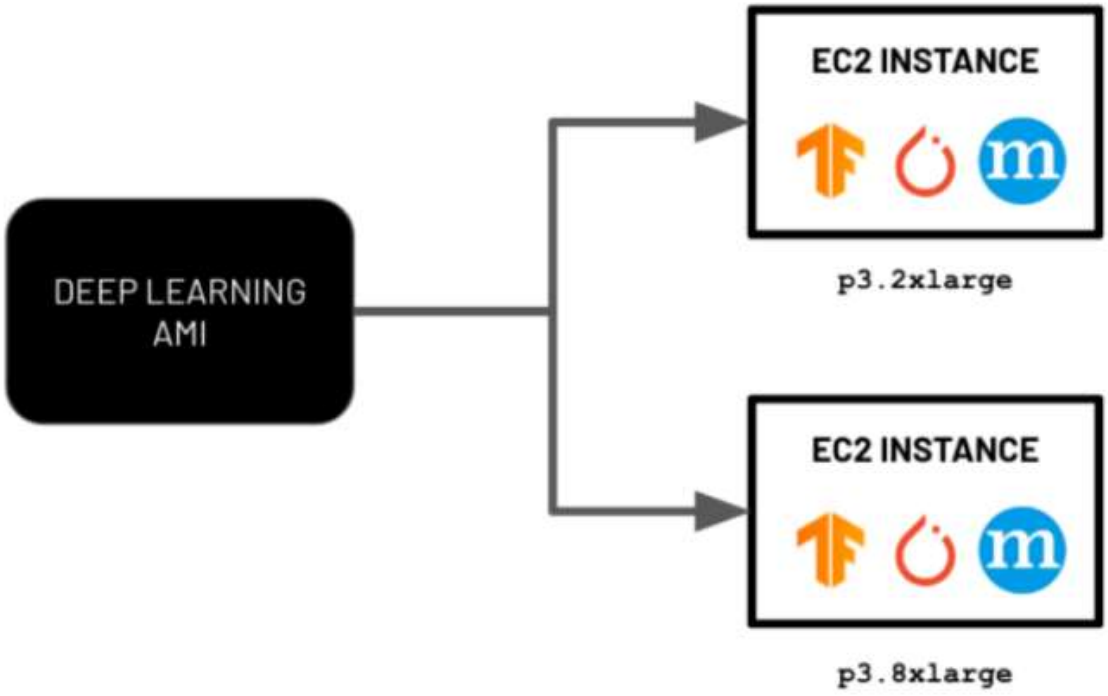


Figure 2.1 – Launching EC2 instances using Deep Learning AMIs

If we were to launch two EC2 instances using the same AMI (similar to what is shown in Figure 2.1), both instances would have the same set of installed packages, frameworks, tools, and operating systems upon instance launch. Of course, not everything needs to be the same as these instances may have different instance types, different security groups, and other configurable properties.

AMIS allow engineers to easily launch EC2 instances in consistent environments without having to spend hours installing different packages and tools. In addition to the installation steps, these EC2 instances need to be configured and optimized before they can be used for specific workloads. Pre-built AMIs such as DLAMIS have popular deep learning frameworks such as TensorFlow, PyTorch, and MXNet pre-installed already. This means that data scientists, developers, and ML engineers may proceed with performing ML experiments and deployments without having to worry about the installation and setup process.

If we had to prepare 20 ML environments with these deep learning frameworks installed, I'm pretty sure that it would not take us 20 or more hours to do so. If we were to use DLAMIs, probably 2 to 3 hours would be more than enough to get the job done. You don't believe me? In the next section, we will do just that! Of course, we will only be preparing a single ML environment instead of 20.

Launching an EC2 instance using a Deep Learning AMI

Launching an EC2 instance from a DLAMI is straightforward. Once we have an idea of which DLAMI to use, the rest of the steps would just be focused on configuring and launching the EC2 instance. The cool thing here is that we are not limited to launching a single instance from an existing image. During the configuration stage, before an instance is launched from an AMI, it is important to note that we can specify the desired value for the number of instances to be launched (for example, 20). This would mean that instead of launching a single in- stance, we would launch 20 instances all at the same time instead.

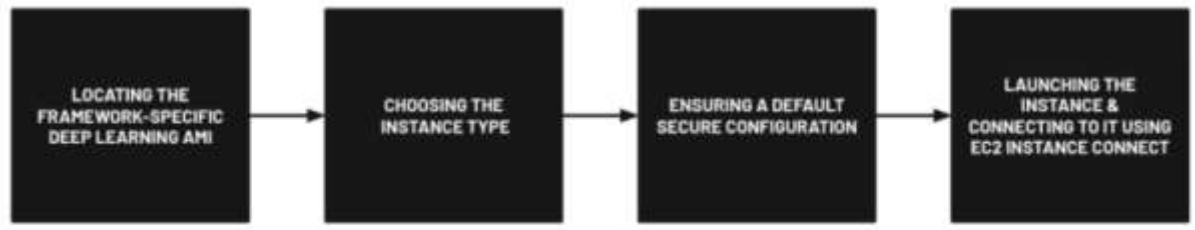


Figure 2.2 – Steps to launch an EC2 instance using a DLAMI

We will divide this section into four parts. As shown in the preceding diagram, we'll start by locating the framework-specific Deep Learning AMI in the AMI Catalog - a repository that contains a variety of AMIS that can be used when launching EC2 instances. We will then launch and configure an EC2 instance using the selected DLAMI and choose a GPU instance type, p3.2xlarge, as the instance type. We'll then configure the security settings, including the

network security settings, to be used by the instance. Finally, we will launch the instance and connect to it from the browser using EC2 Instance Connect.

Locating the framework-specific DLAMI

When looking for an AMI, the first place we should check is the AWS AMI Catalog. In the AMI Catalog, we should find a variety of DLAMIS. These DLAMIS can be categorized into either multi-framework DLAMIS or framework-specific DLAMIS. What's the difference? Multi-framework DLAMIS include multiple frameworks in a single AMI such as TensorFlow, PyTorch, or MXNet. This allows for easy experimentation and exploration of several frameworks for developers, ML engineers, and data scientists. On the other hand, framework-specific DLAMIS are more optimized for production environments and support only a single framework.

In the next set of steps, we will navigate to the AMI Catalog and use the framework-specific (TensorFlow) Deep Learning AMI to launch an instance:

1. Navigate to the AWS Management Console and then type ec2 in the search bar. Select EC2 from the list of results:

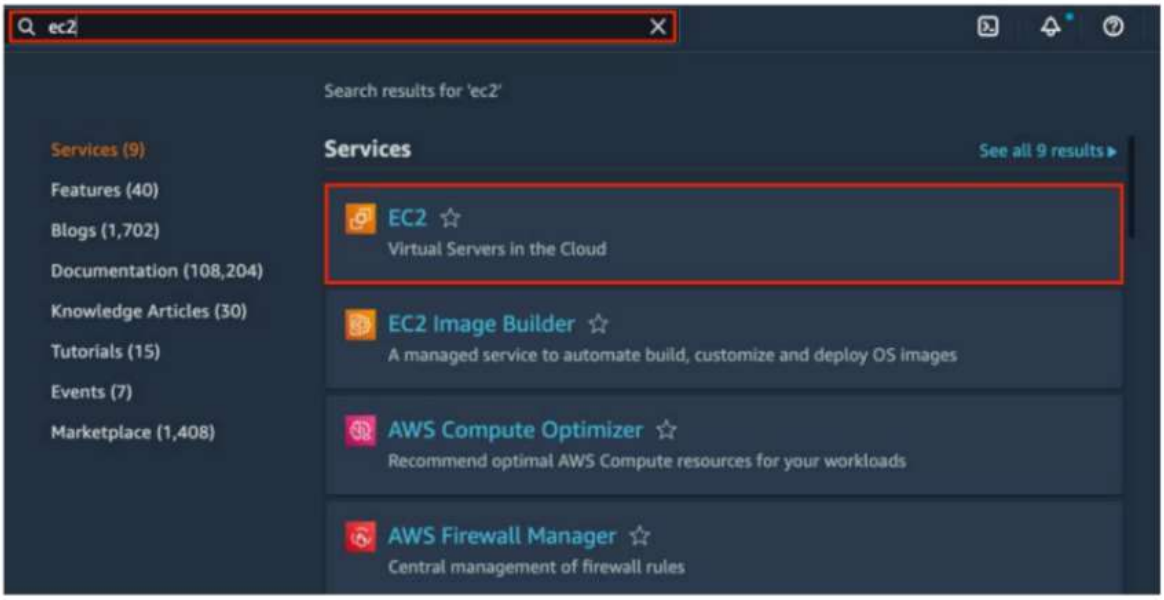


Figure 2.3 – Navigating to the EC2 console

We should see a list of matching results such as EC2, EC2 Image Builder, and AWS Compute Optimizer, similar to what is shown in Figure 2.2. From this list, we'll choose the first one, which should redirect us to the EC2 console.

2. In the sidebar, locate and click AMI Catalog under Images to navigate to the EC2 > AMI Catalog page.

3. Next, type deep learning ami in the search bar within the AMI Catalog page. Make sure that you press Enter to search for relevant AMIs related to the search query:

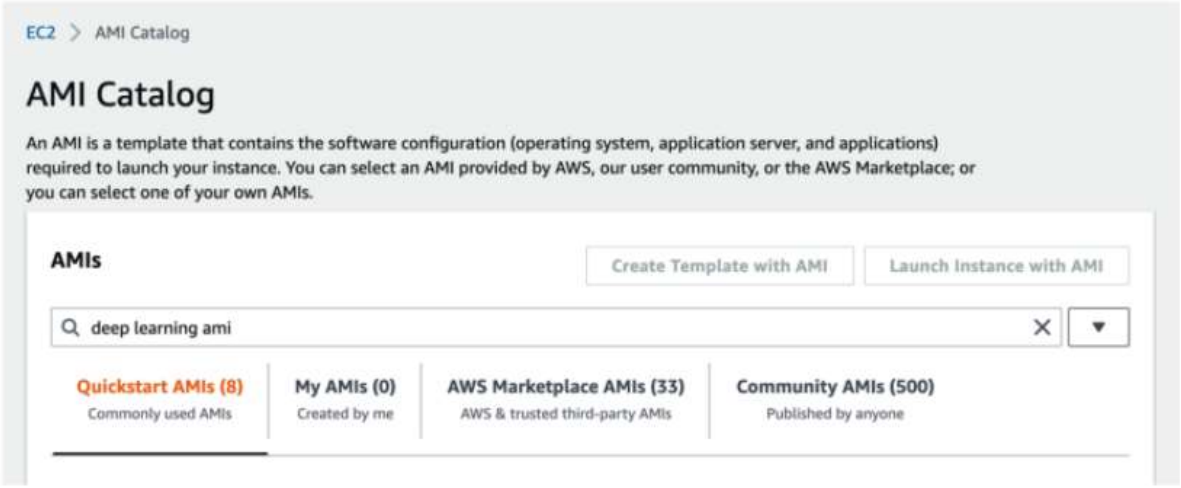


Figure 2.4 – Searching for the framework-specific Deep Learning AMI

As shown in the preceding screenshot, we should have a couple of matching results under Quickstart AMIs. There should be matching results under AWS Marketplace AMIs and Community AMIs as well. Quickstart AMIs include the commonly used AMIS for key workloads such as the Amazon Linux 2 AMI, the Ubuntu Server 20.04 LTS AMI, the Deep Learning AMI (Amazon Linux 2) AMI, and more. AWS Marketplace AMIS include several AMIs created by AWS, along with AMIs created by trusted third-party sources. These should include AMIS such as the OpenVPN Access Server AMI, the Kali Linux AMI, and the Splunk Enterprise AMI. All publicly available AMIs can be found under Community AMIS.

4. Scroll down the list of Quickstart AMIs and locate the framework-specific Deep Learning AMI, as shown in the following screenshot:

Figure 2.5 – Locating the TensorFlow DLAMI

Here, we are choosing the framework-specific (TensorFlow) Deep Learning AMI for Amazon Linux 2 since we'll be training an ML model using TensorFlow later in this chapter. Verify the selection by reading the name and description of the AMI. Then, click the Select button.

5. After you have clicked the Select button in the previous step, scroll up to the top of the page and click the Launch Instance with AMI button, as shown in the following screenshot:

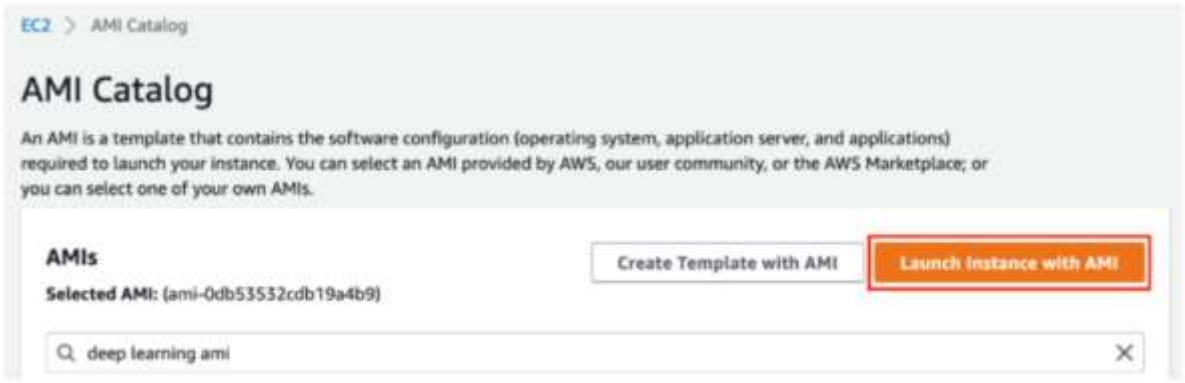


Figure 2.6 – Launch Instance with AMI

As we can see, the Launch Instance with AMI button is just beside the Create Template with AMI button.

Clicking the Launch Instance with AMI button should redirect you to the Launch an instance page, as shown in the following screenshot:

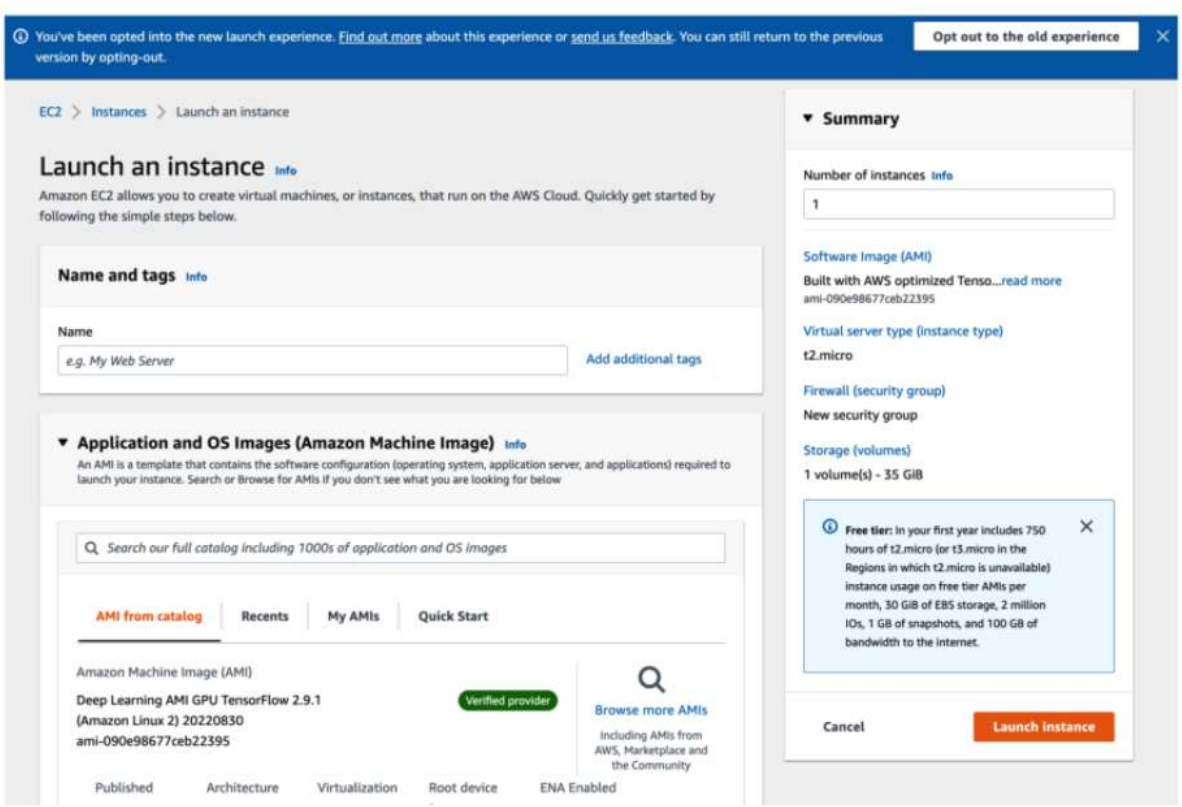


Figure 2.7 – The Launch an instance page

Since AWS regularly updates the experience of launching and managing resources in the console, you might see a few differences while you are performing the next set of steps. However, the desired final configuration will be the same, regardless of what the console looks like while you are working on this section.

6. Under Name and tags, specify MLE-CH02-DLAMI in the Name field.

After setting the Name field's value, the next step involves choosing the desired instance type for our EC2 instance. Before we proceed with selecting the desired instance type, we must have a quick discussion about what instances are available and which types of instances are suitable for large-scale ML work- loads.

Choosing the instance type

When performing deep learning experiments, data scientists and ML engineers generally prefer GPU instances over CPU instances. Graphics Processing Units (GPUs) help significantly speed up deep learning experiments since GPUs can be used to process multiple parallel computations

at the same time. Since GPU instances are usually more expensive than CPU instances, data scientists and ML engineers use a combination of both types when dealing with ML requirements. For example, ML practitioners may limit the usage of GPU instances just for training deep learning models only. This means that CPU instances would be used instead for inference endpoints where the trained models are deployed. This would be sufficient in most cases, and this would be considered a very practical move once costs are taken into consideration.

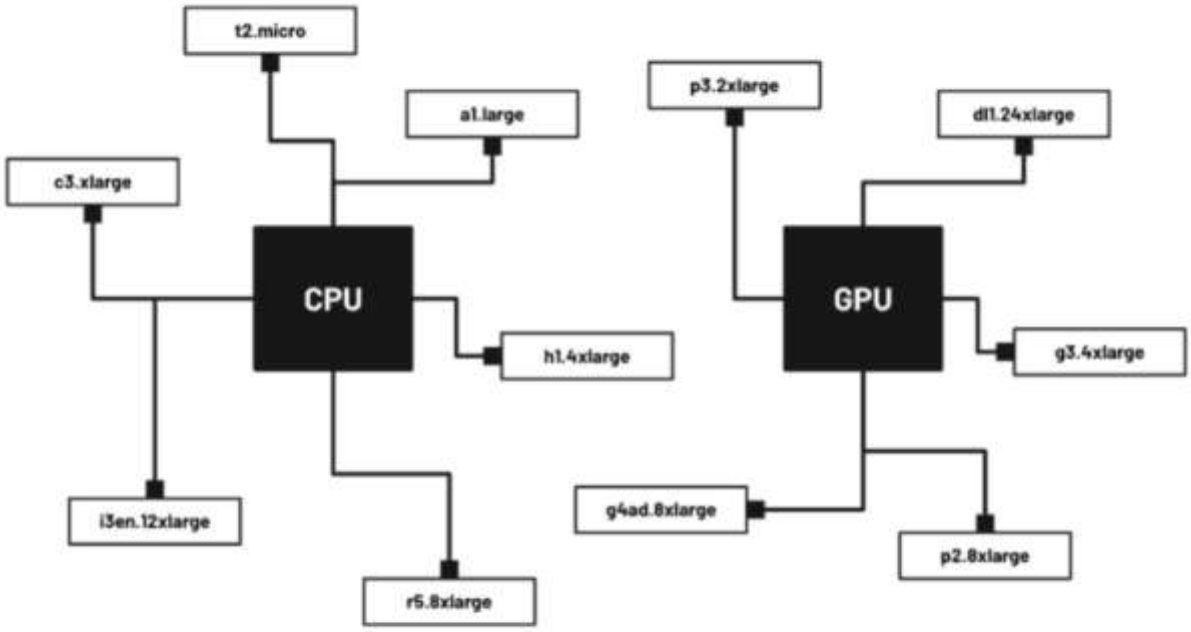


Figure 2.8 – CPU instances versus GPU instances

That said, we need to identify which instances fall under the group of GPU instances and which instances fall under the CPU instances umbrella. The preceding diagram shows some examples of GPU instances, including p3.2xlarge, dl1.24xlarge, g3.4xlarge, p2.8xlarge, and g4ad.8xlarge. There are other examples of GPU instance types not in this list, but you should be able to identify these just by checking the instance family. For example, we are sure that p3.8xlarge is a GPU instance type since it belongs to the same family as the p3.2xlarge instance type.

Now that we have a better idea of what CPU and GPU instances are, let's proceed with locating and choosing p3.2xlarge from the list of options for our instance type:

1. Continuing where we left off in the Locating the frame- work-specific DLAMI section, let's locate and click the Compare instance types link under the Instance type pane. This

should redirect you to the Compare instance types page, as shown in the following screenshot:

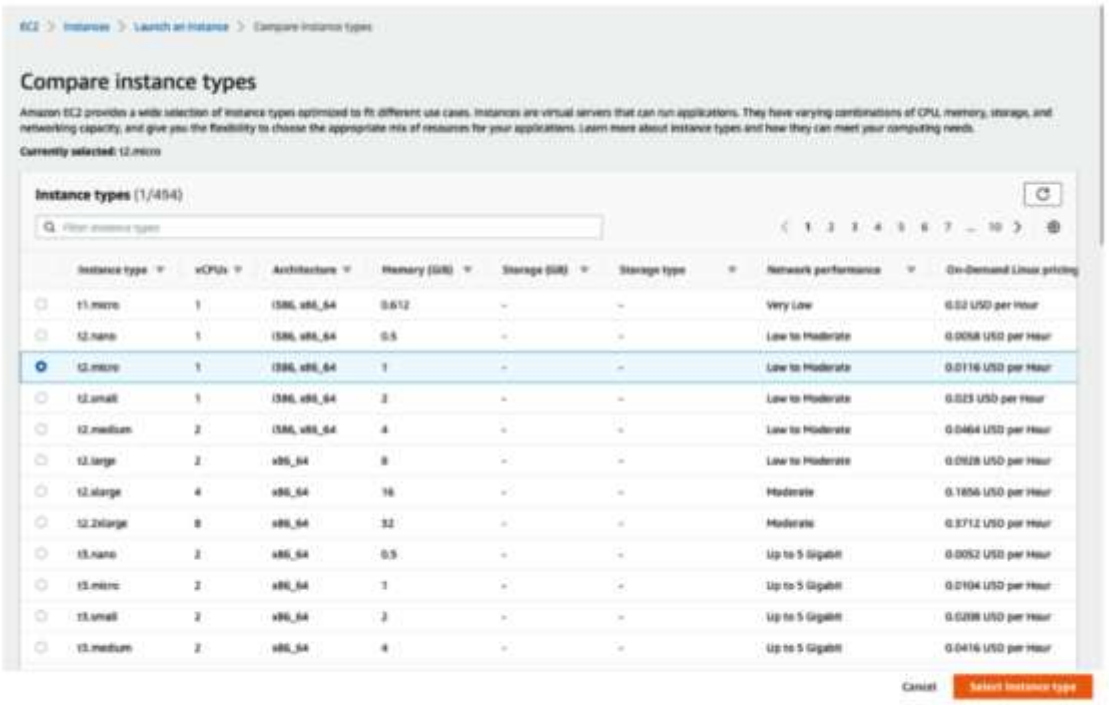


Figure 2.9 – Compare instance types

Here, we can see the different instance types, along with their corresponding specs and cost per hour.

2. Click the search field (with the Filter instance types placeholder text). This should open a drop-down list of options, as shown in the following screenshot:

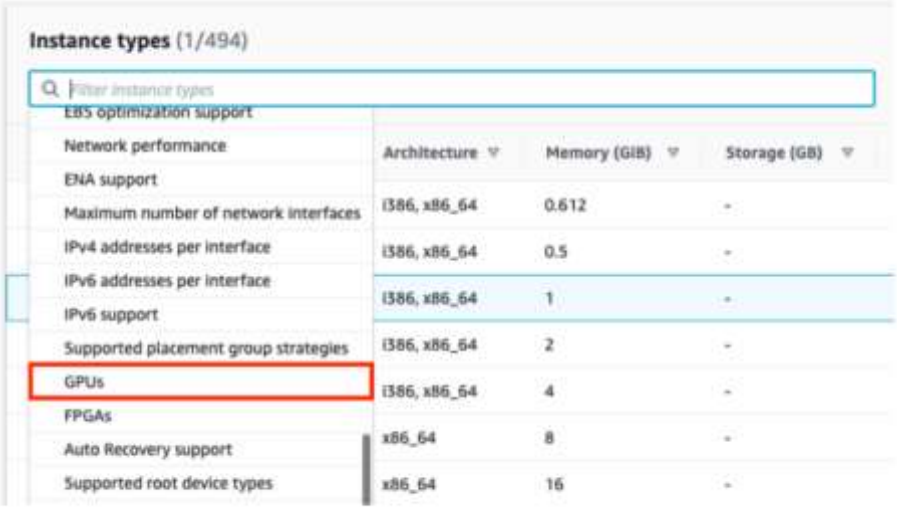


Figure 2.10 – Using the Filter instance types search field

Locate and select GPUs from the list of options. This should open the Add filter for GPUs window.

3. In the Add filter for GPUs window, open the drop- down menu and select > from the list of options available. Next, specify a value of 0 in the text field beside it. Click the Confirm button afterward.

4. Next, let's click the Preferences button, as highlight- ed in the following screenshot:

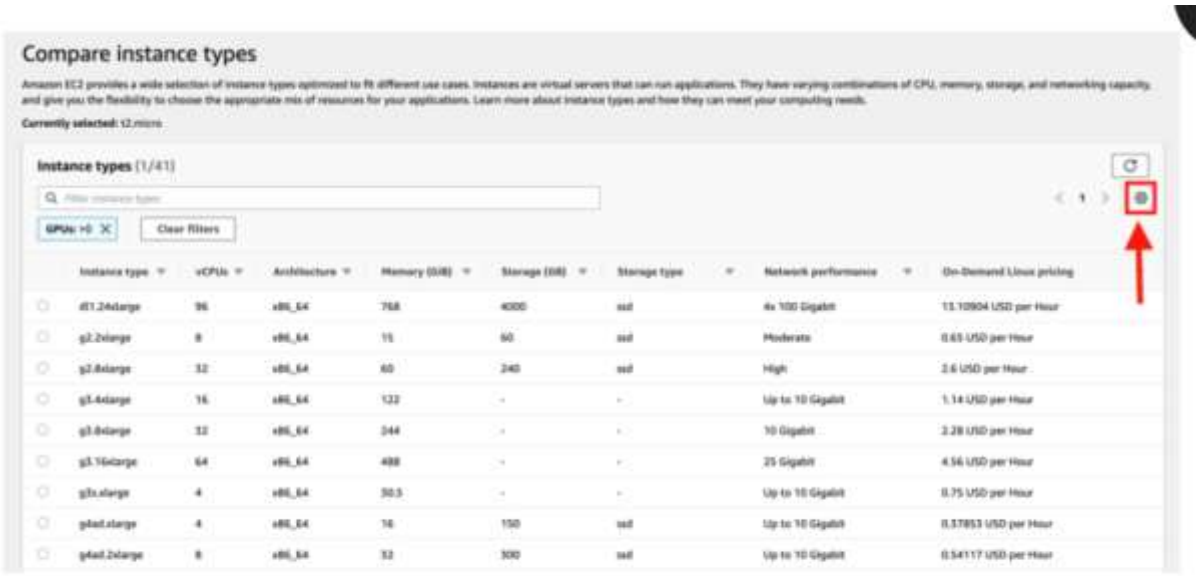


Figure 2.11 – Opening the Preferences window

This should open the Preferences window. Under Attribute columns, ensure that the GPUs radio button is toggled on, as shown in the following screenshot:

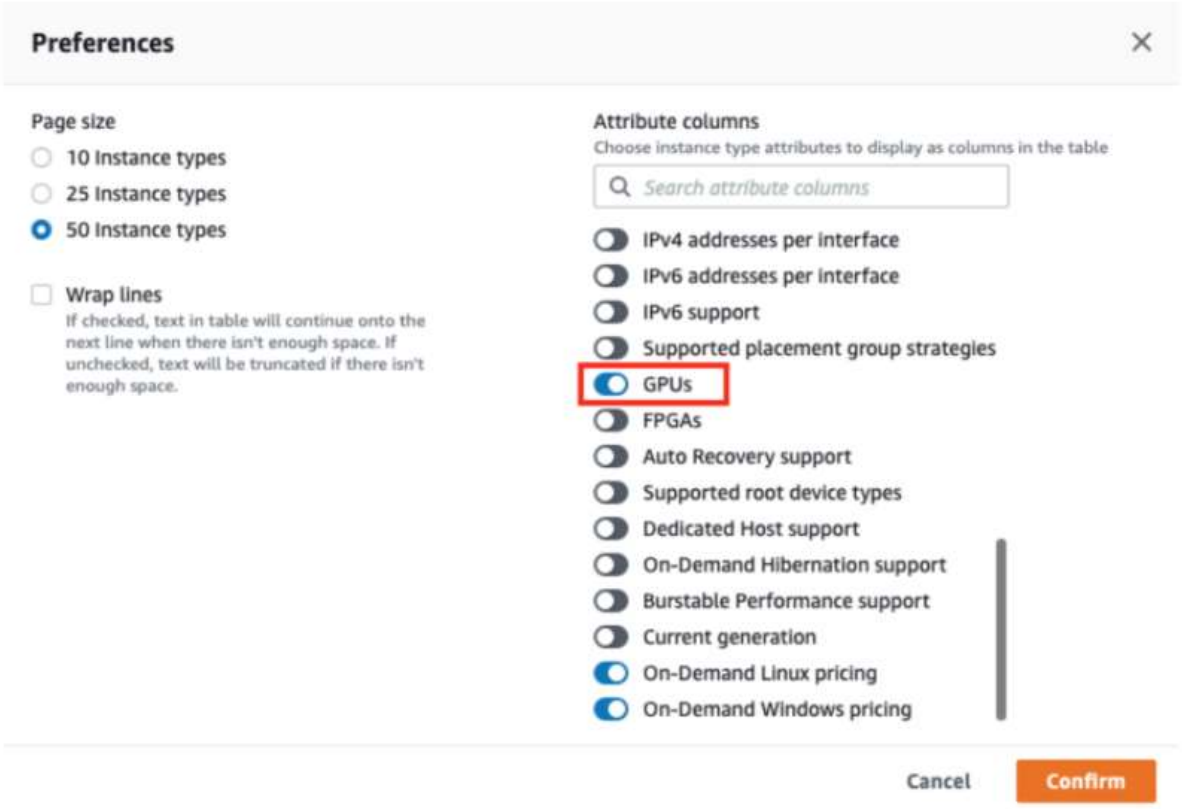


Figure 2.12 – Displaying the GPUs attribute column

Click the Confirm button afterward. This should update the table list display and show us the number of GPUs of each of the instance types in the list, as shown here:

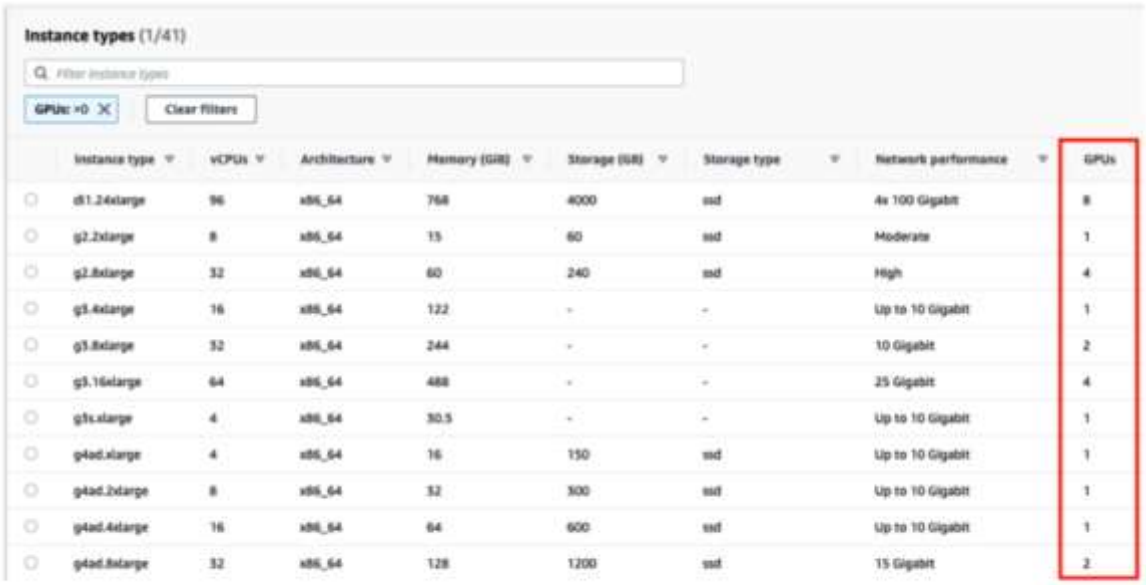


Figure 2.13 – GPUs of each instance type

Here, we should see a pattern that the number of GPUs general- ly increases as the instance type becomes "larger" within the same instance family.

5. Locate and select the row corresponding to the p3.2x- large instance type. Take note of the number of GPUs available, along with the cost per hour (on-demand Linux pricing) for the p3.2xlarge instance type.

6. Click the Select instance type button (located at the lower right portion of the screen) afterward.

This should close the Compare instance types window and return you to the Launch an instance page.

Ensuring a default secure configuration

When launching an EC2 instance, we need to manage the securi- ty configuration, which will affect how the instance will be ac- cessed. This involves configuring the following:

- **Key pair:** Files containing credentials used to securely access the instance (for example, using SSH)
- **Virtual Private Cloud (VPC):** A logically isolated vir- tual network that dictates how resources are accessed and how resources communicate with each other
- **Security group:** A virtual firewall that controls traffic going in and out of the EC2 instance using rules that filter the traffic based on the configured protocol and ports

That said, let's proceed with completing the remaining configura- tion parameters before we launch the EC2 instance:

1. Continuing where we left off in the Choosing the instance type section, let's proceed with creating a new key pair. Under Key pair (login), locate and click Create new key pair.
2. In the Create key pair window, specify a unique key pair name (for example, dlami-key) for Key pair name. Ensure that the following configuration holds as well:
    - Key pair type: RSA
    - Private key file format: .pem

3. Click the Create key pair button afterward. This should automatically download the .pem file to your local machine. Note that we won't need this .pem file for the hands-on solutions in this chapter since we'll be accessing the instance later using EC2 Instance Connect (through the browser).

4. Under Network settings, locate and click the Edit button (located at the top right of the pane). Make sure that the following configuration settings are ap- plied:

- VPC - required: [Select default VPC] vpc-xxxxxxxx (default)
- Auto-assign public IP: Enable
- Firewall (security groups): Create security group

5. Under Inbound security group rules of Network settings, specify a set of security group rules, similar to what is configured in the following screenshot:

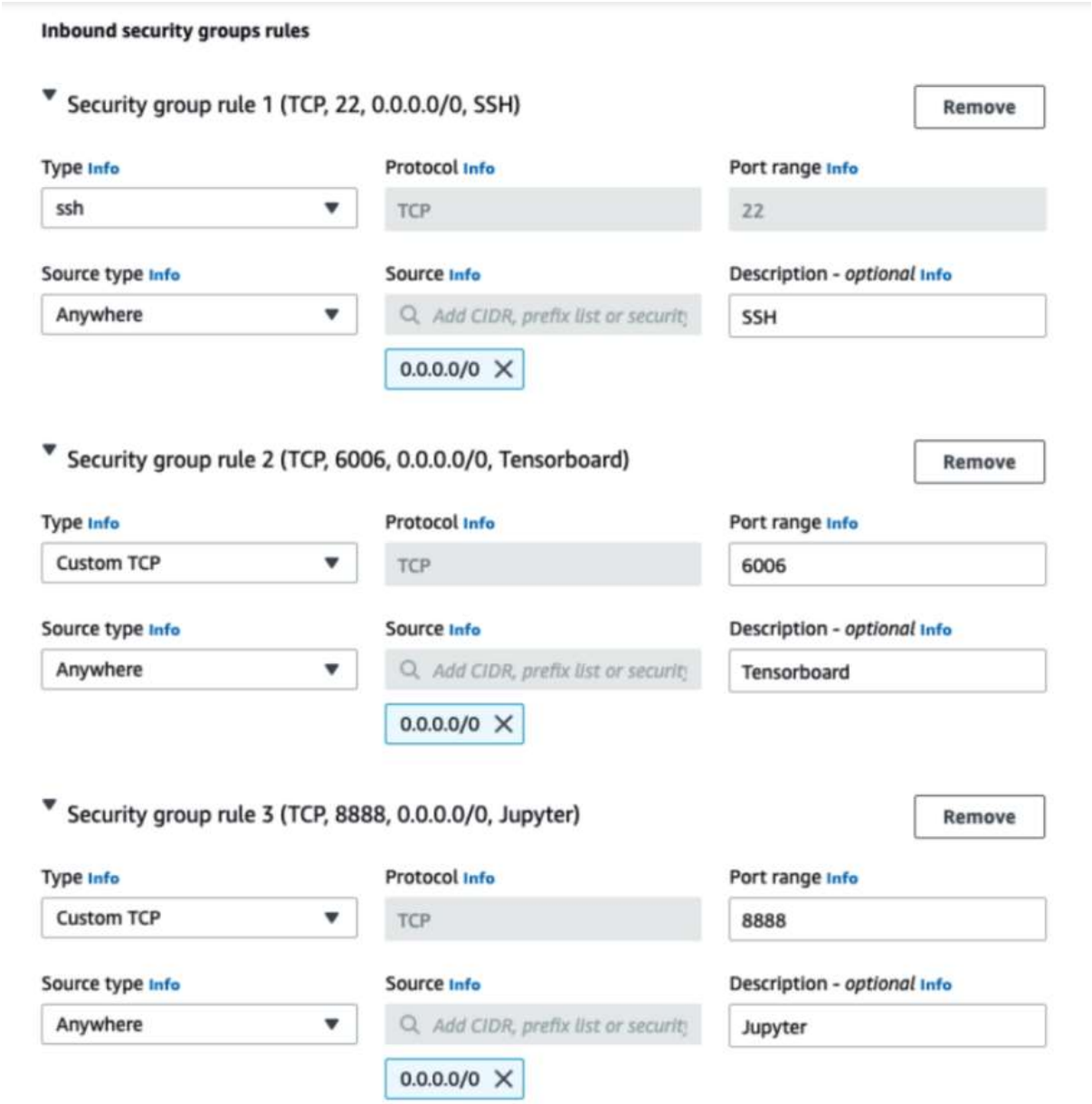


Figure 2.14 – Inbound security groups rules

As you can see, we will be configuring the new security group with the following rules:

- Type: SSH; Protocol: TCP; Port Range: 22; Source type: Anywhere | Source: 0.0.0.0/0; Description: SSH - allows any "computer" such as your local ma- chine to connect to the EC2 instance via the Secure Shell (SSH) protocol over port 22
- Type: Custom TCP; Protocol: TCP; Port Range: 6006; Source type: Anywhere | Source: 0.0.0.0/0; De- scription: Tensorboard - allows any "computer" such as your local machine to access port 6006 of the EC2 in- stance (which may be running an application such as TensorBoard)

- Type: Custom TCP; Protocol: TCP; Port Range: 8888; Source type: Anywhere | Source: 0.0.0.0/0; De- scription: Jupyter - allows any "computer" such as your local machine to access port 8888 of the EC2 in- stance (which may be running an application such as the Jupyter Notebook app)

You may proceed with the next step once you have configured the new security group with Security group name quired and Description – required and the relevant set of Inbound security group rules.

Launching the instance and connecting to it using EC2 Instance Connect

There are different ways to connect to an EC2 instance. Earlier, we configured the instance so that it can be accessed via SSH using a key file (for example, from the Terminal of your local ma- chine). Another possible option is to use EC2 Instance Connect to access the instance through the browser. We can also access the instance via SSH using Session Manager. In this section, we'll use EC2 Instance Connect to access our instance. Continuing where we left off in the Ensuring a default secure configuration section, let's proceed with launching the EC2 in- stance and access it from the browser:

1. Once you have configured the storage settings, locate and click the Launch instance button under the Summary pane (located at the right portion of the screen). Make sure that you terminate this instance within the hour it has been launched as the per-hour rate of these types of instances is a bit higher relative to other instance types. You may check the Cleaning up section of this chapter for more details.

2. You should see a success notification, along with the in- stance ID of the resource being launched, similar to what is shown in the following screenshot:

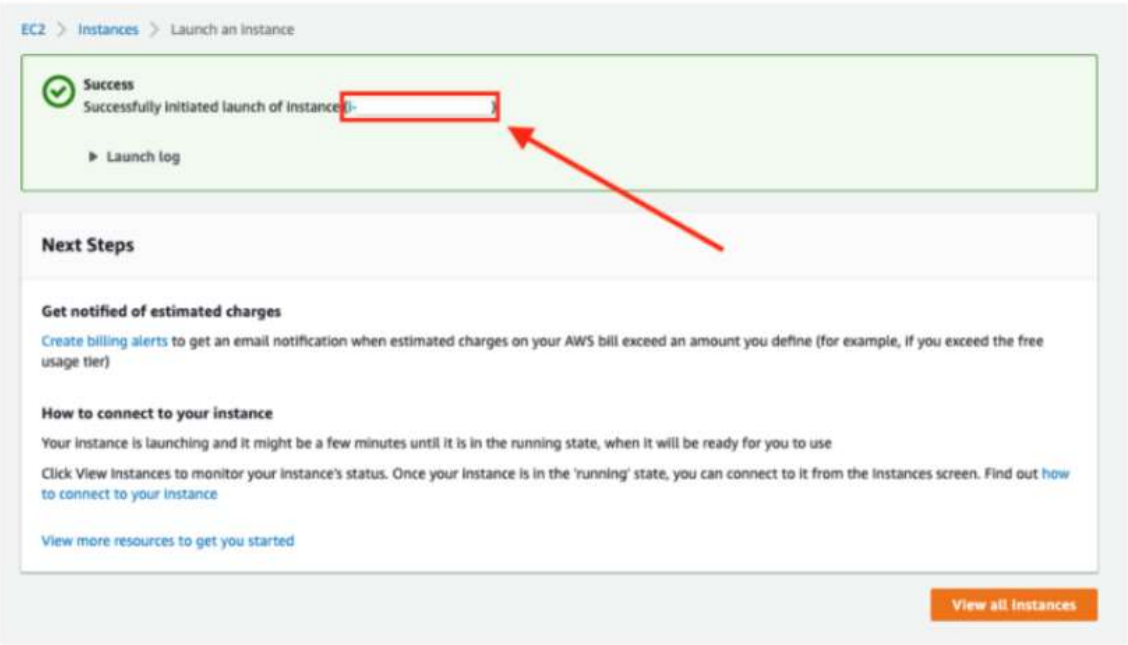


Figure 2.16 – Launch success notification

Click the link containing the instance ID (i-xxxxxxxxxxxxxxx- xx), as highlighted in the preceding screenshot, to navigate to the Instances page. You may click the refresh button (beside the Connect button) a few times while waiting for the EC2 in- stance (MLE-CH02-DLAMI) to appear in the list of instances.

3. Select the instance by toggling the checkbox highlight- ed in the following screenshot. Click the Connect button afterward:

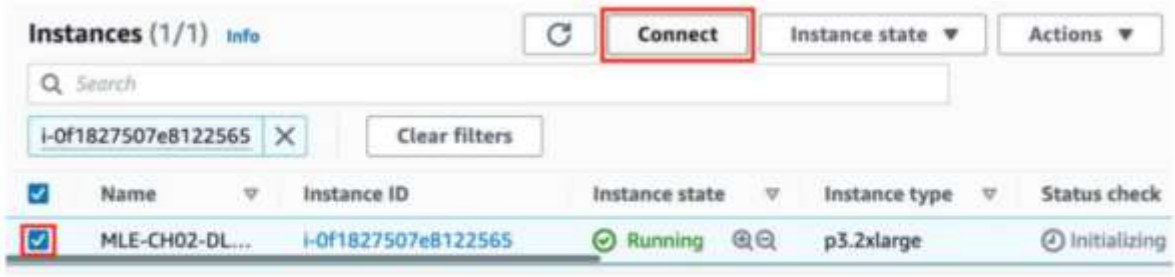


Figure 2.17 – Connecting to the instance directly

Here, we can see that there's an option to connect directly to the instance using the browser.

4. In the EC2 Instance Connect tab, locate and copy the Public IP address value (AA.BB.CC.DD) to a text editor on your local machine. Note that you will get a different public IP address value. We will use this IP address value later in this chapter

when accessing TensorBoard (the visualization toolkit of Tensor- Flow) and the Jupyter notebook application. Leave the User name value as-is (root) and then click Connect:

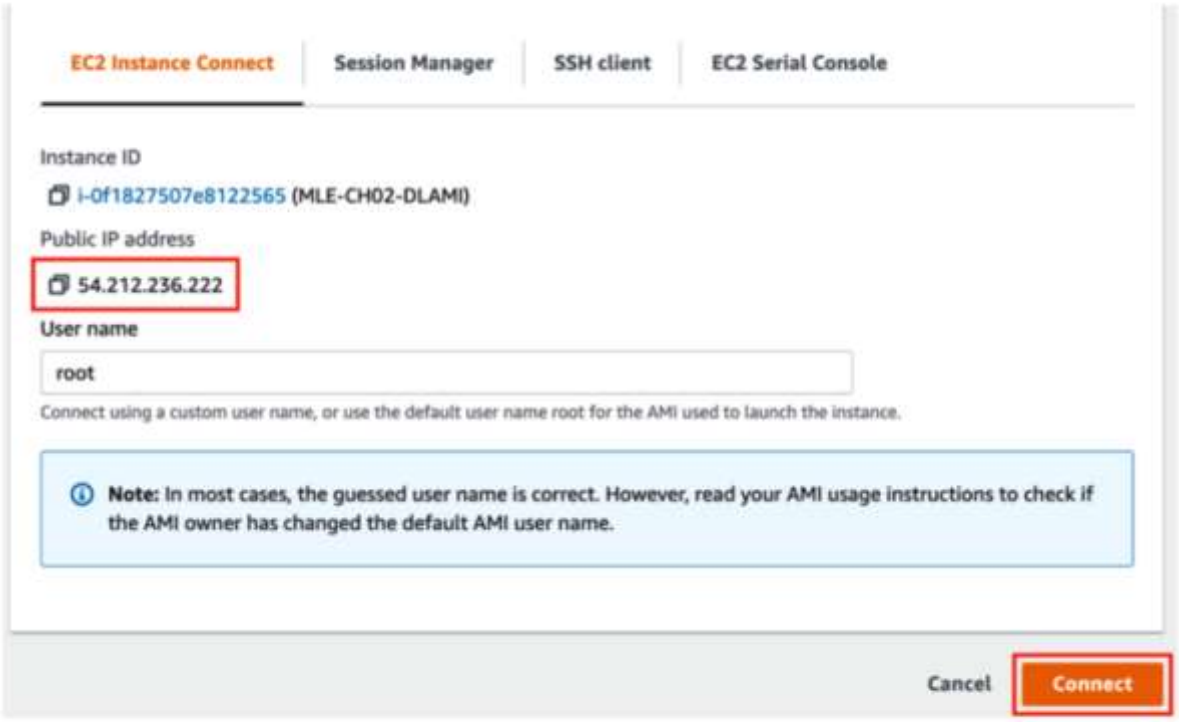


Figure 2.18 – EC2 Instance Connect

This should open a new tab that will allow us to run terminal commands directly from the browser. If you are getting a There was a problem connecting to your instance error message, wait for about 2 to 3 minutes before refreshing the page or click- ing the Retry button:

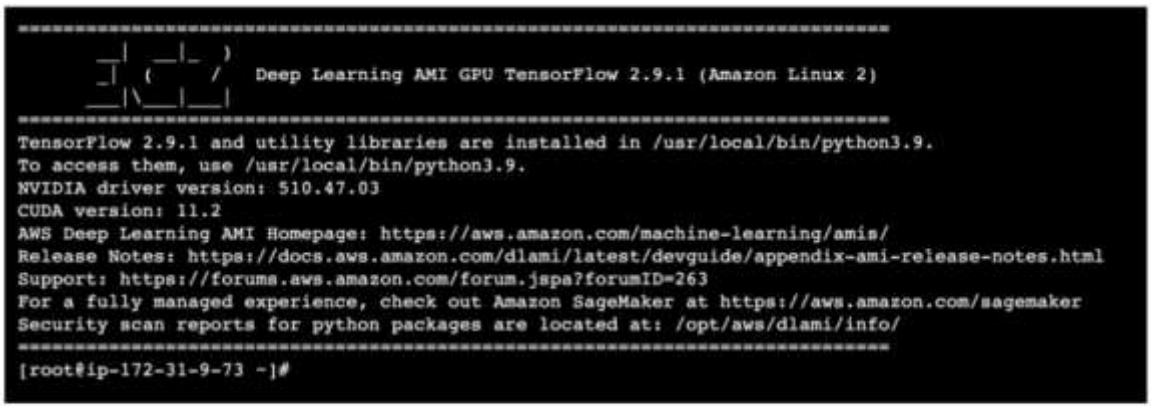


Figure 2.19 – EC2 Instance Connect terminal

What is Amazon EMR Serverless?

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

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

Concepts of Amazon EMR Serverless

Release version

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

Application

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

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

After you create an application, you can submit data-processing jobs or interactive requests to yourapplication. Each EMR Serverless application runs on a secure Amazon Virtual Private Cloud (VPC) strictly apartfrom other applications. Additionally, you can use AWS Identity and Access Management (IAM) policies to define which users and roles can access the application. You can also specify limits to control and track usage costs incurred by the application.

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

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

EMR Serverless is a Regional service that simplifies how workloads run across multiple AvailabilityZones in a Region.

Job run

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

Workers

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

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

Pre-initialized capacity

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

EMR Studio

EMR Studio is the user console that you can use to manage your EMR Serverless applications. If an EMR Studio doesn't exist in your account when you create your first EMR Serverless application, we automatically create one for you. You can access EMR Studio either from the Amazon EMR console, or you can turn on federated access from your identity provider (IdP) through IAM or IAM Identity Center. When you do this, users can access Studio and manage EMR Serverless applications withoutdirect access to the Amazon EMR console.

Setting up

Topics

- Sign up for an AWS account
- Create a user with administrative access
- Grant permissions
- Install and configure the AWS CLI
- Open the console

Sign up for an AWS account

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

To sign up for an AWS account

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

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

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

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

Create a user with administrative access

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

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** andentering your AWS account email address. On the next page, enter your password.
2. Turn on multi-factor authentication (MFA) for your root user.

Create a user with administrative access

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

Sign in as the user with administrative access

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

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.
2. Assign users to a group, and then assign single sign-on access to the group.

Grant permissions

In production environments, we recommend that you use finer-grained policies.

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EMRStudioCreate",
            "Effect": "Allow",
            "Action": [
                "elasticmapreduce:CreateStudioPresignedUrl",
                "elasticmapreduce:DescribeStudio",
                "elasticmapreduce:CreateStudio",
                "elasticmapreduce:ListStudios"
            ],
            "Resource": "*"
        },
        {
            "Sid": "EMRServerlessFullAccess",
            "Effect": "Allow",
            "Action": [
                "emr-serverless:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowEC2ENICreationWithEMRTags",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface"
            ],
            "Resource": [
```

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

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:
- Users managed in IAM through an identity provider:
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Creating a role for an IAMuser in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow theinstructions in Adding permissions to a user (console) in the *IAM User Guide*.

Grant programmatic access

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

Install and configure the AWS CLI

If you want to use EMR Serverless APIs, you must install the latest version of the AWS Command Line Interface (AWS CLI).

To set up the AWS CLI

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

```
aws emr-serverless help
```

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

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

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

Getting started with EMR Serverless from the console

Steps to complete

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

Step 1: Create an EMR Serverless application

Create a new application with EMR Serverless as follows.

1. Sign in to the AWS Management Console and open the Amazon EMR console at https://console.aws.amazon.com/emr.
2. In the left navigation pane, choose **EMR Serverless** to navigate to the EMR Serverless landingpage.
3. To create or manage EMR Serverless applications, you need the EMR Studio UI.
   - If you already have an EMR Studio in the AWS Region where you want to create an application, then select **Manage applications** to navigate to your EMR Studio, or select thestudio that you want to use.
   - If you don't have an EMR Studio in the AWS Region where you want to create an application,choose **Get started** and then Choose **Create and launch Studio**. EMR Serverless creates a EMR Studio for you so that you can create and manage applications.
4. In the **Create studio** UI that opens in a new tab, enter the name, type, and release version for your application. If you only want to run batch jobs, select **Use default settings for batch jobsonly**. For interactive workloads, select **Use default settings for interactive workloads**. You canalso run batch jobs on interactive-enabled applications with this option. If you need to, you can change these settings later.

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

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

Step 2: Submit a job run or interactive workload

Spark job run

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

To run a Spark job

1. Upload the sample script wordcount.pyinto your new bucket with the followingcommand.

```
aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/
scripts/wordcount.py s3://DOC-EXAMPLE-BUCKET/scripts/
```

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

   - In the **Name** field, enter the name that you want to call your job run.
   - In the **Runtime role** field, enter the name of the role that you created in Create a jobruntime role.
   - In the **Script location** field, enter s3://*DOC-EXAMPLE-BUCKET*/scripts/wordcount.pyas the S3 URI.
   - In the **Script arguments** field, enter ["s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-spark/output"].

```
--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
```

- In the **Spark properties** section, choose **Edit as text** and enter the followingconfigurations.

4. To start the job run, choose **Submit job** .

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

Hive job run

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

To run a Hive job

1. Create a file called hive-query.qlthat contains all the queries that you want to run inyour 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.qlto 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. Completing Step 1: Create an EMR Serverless application takes you to the **Application details** page in EMR Studio. There, choose the **Submit job** option.

4. On the **Submit job** page, complete the following.
   - In the **Name** field, enter the name that you want to call your job run.
   - In the **Runtime role** field, enter the name of the role that you created in Create a jobruntime role.
   - In the **Script location** field, enter s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-hive/query/hive-query.qlas the S3 URI.
   - In the **Hive properties** section, choose **Edit as text**, and enter the followingconfigurations.
5. To start the job run, choose **Submit job**.
6. In the **Job runs** tab, you should see your new job run with a **Running** status.

Step 3: View application UI and logs

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

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

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

Step 4: Clean up

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

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

Getting started from the AWS CLI

Step 1: Create an EMR Serverless application

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

Spark

To create a Spark application, run the following command.

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

Hive

To create a Hive application, run the following command.

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

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

Before you move on to Step 2: Submit a job run to your EMR Serverless application, make sure thatyour application has reached the CREATEDstate with the get-applicationAPI.

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

EMR Serverless creates workers to accommodate your requested jobs. By default, these are createdon demand, but you can also specify a pre-initialized capacity by setting the initialCapacity parameter when you create the application.

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

Now your EMR Serverless application is ready to run jobs.Spark

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

To run a Spark job

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

```
aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/
scripts/wordcount.py s3://DOC-EXAMPLE-BUCKET/scripts/
```

2. In the following command, substitute *application-id* with your application ID. Substitute *job-role-arn* with the runtime role ARN you created in Create a job runtimerole. Substitute *job-run-name*with the name you want to call your job run. Replace all *DOC-EXAMPLE-BUCKET* strings with the Amazon S3 bucket that you created, and add / output to the path. This creates a new folder in your bucket where EMR Serverless can copy the output files of your application.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --name job-run-name \
    --job-driver '{
        "sparkSubmit": {
          "entryPoint": "s3://DOC-EXAMPLE-BUCKET/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"
        }
    }'
```

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

Hive

In this tutorial, we create a table, insert a few records, and run a count aggregation query. Torun the Hive

job, first create a file that contains all Hive queries to run as part of single job,upload the file to S3, and specify this S3 path when you start the Hive job.

To run a Hive job

1. Create a file called hive-query.qlthat contains all the queries that you want to run inyour 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.qlto 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 following command, substitute *application-id* with your own application ID. Substitute *job-role-arn* with the runtime role ARN you created in Create a job runtimerole. Replace all *DOC-EXAMPLE-BUCKET* strings with the Amazon S3 bucket that you created, and add /outputand /logsto the path. This creates new folders in your bucket,where EMR Serverless can copy the output and log files of your application.

```
aws emr-serverless start-job-run \
    --application-id application-id \
    --execution-role-arn job-role-arn \
    --job-driver '{ "hive": {
  "query": "s3://DOC-EXAMPLE-BUCKET/emr-serverless-hive/query/hive-query.ql",
        "parameters": "--hiveconf hive.log.explain.output=false"
    }}' }}'
```

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

Step 3: Review your job run's output

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

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

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

With your log destination set to s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-spark/ logs, you can find the logs for this specific job run under s3://*DOC-EXAMPLE-BUCKET*/emr- serverless-spark/logs/applications/*application-id*/jobs/*job-run-id*.

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

Hive

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

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

With your log destination set to s3://*DOC-EXAMPLE-BUCKET*/emr-serverless-hive/ logs, you can find the logs for this specific job run under s3://*DOC-EXAMPLE-BUCKET*/emr- serverless-hive/logs/applications/*application-id*/jobs/*job-run-id*.

For Hive applications, EMR Serverless continuously uploads the Hive driver to the HIVE_DRIVER folder, and Tez tasks logs to the TEZ_TASK folder, of your S3 log destination. After the job run reaches the SUCCEEDEDstate, the output of your Hive query becomes available inthe Amazon S3 location that you specified in the monitoringConfiguration field of configurationOverrides.

Step 4: Clean up

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

Delete your application

To delete an application, use the following command.

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

Delete your S3 log bucket

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

```
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 andthe policy.

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

To delete the role, use the following command.

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

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

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

Interacting with an application

This section covers how you can interact with your Amazon EMR Serverless application with the AWS CLI and the defaults for Spark and Hive engines.

Topics

Application states

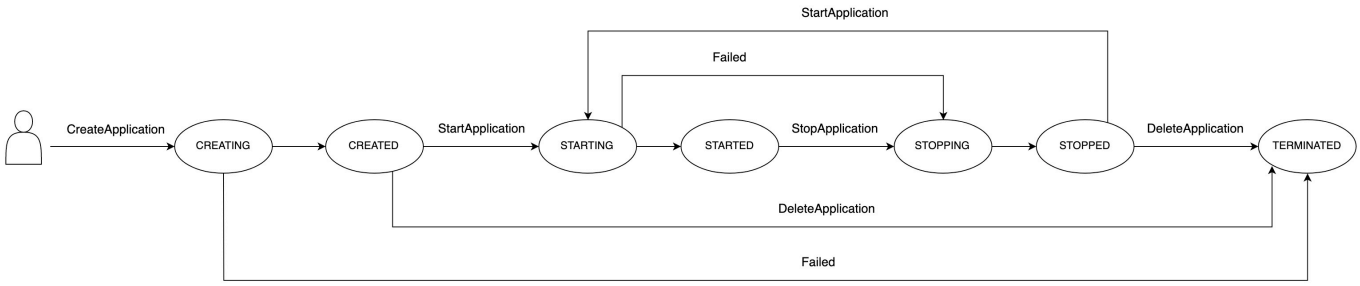
When you create an application with EMR Serverless, the application run enters the `CREATING` state. It then passes through the following states until it succeeds (exits with code `0`) or fails (exits with a non-zero code).

Applications can have the following states:

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

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

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



Interacting with your application from the EMR Studio console

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

Create an application

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

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

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

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

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

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

   For more information about pre-initialized capacity, application limits, and application behavior, see Configuring an application. For more information about network connectivity, see Configuring VPC access.
6. To create the application, choose **Create application** .

List applications

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

Manage applications

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

Start application

Choose this option to manually start an application.

Stop application

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

Configure application

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

Delete application

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

Interacting with your application on the AWS CLI

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

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

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

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

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

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

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

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

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

Configuring an application

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

Topics

Understanding application behavior

Default application behavior

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

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

Maximum capacity

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

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

Supported worker configurations

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

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

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

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

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

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

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

Pre-initialized capacity

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

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

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

You can configure the application to release pre-initialized capacity if it isn't used for a certain period of time, with a default of 15 minutes. A stopped application starts automatically when you submit a new job. You can set these automatic start and stop configurations when you create the application, or you can change them when the application is in a `CREATED` or `STOPPED` state.

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

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

Customizing pre-initialized capacity for Spark and Hive

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

Configuring an application running Apache Hive with pre-initialized capacity

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

```
aws emr-serverless create-application \
  --type "HIVE" \
  --name my-application-name \
  --release-label emr-6.6.0 \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "TEZ_TASK": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
  }'
```

Configuring an application running Apache Spark with pre-initialized capacity

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

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

```
aws emr-serverless create-application \
  --type "SPARK" \
  --name my-application-name \
  --release-label emr-6.6.0 \
  --initial-capacity '{
    "DRIVER": {
        "workerCount": 5,
        "workerConfiguration": {
            "cpu": "2vCPU",
            "memory": "4GB"
        }
    },
    "EXECUTOR": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB"
        }
    }
  }' \
  --maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
```

```
}'
```

Default application configuration for EMR Serverless

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

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

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

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

Declaring configurations at the application level

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

`monitoringConfiguration`

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

`runtimeConfiguration`

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

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

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

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

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

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

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

Example declaration

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

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

```
                "spark.executor.cores": "2",
                "spark.driver.memory": "8G",
                "spark.executor.memory": "8G",
                "spark.executor.instances": "2",

 "spark.hadoop.javax.jdo.option.ConnectionDriverName":"org.mariadb.jdbc.Driver",
                "spark.hadoop.javax.jdo.option.ConnectionURL":"jdbc:mysql://db-host:db-
port/db-name",
                "spark.hadoop.javax.jdo.option.ConnectionUserName":"connection-user-
name",
                "spark.hadoop.javax.jdo.option.ConnectionPassword":
 "EMR.secret@SecretID"
            }
        },
        {
            "classification": "spark-driver-log4j2",
            "properties": {
                "rootLogger.level":"error",
                "logger.IdentifierForClass.name": "classpathForSettingLogger",
                "logger.IdentifierForClass.level": "info"
            }
        }
    ]' \
    --monitoring-configuration '{
        "s3MonitoringConfiguration": {
            "logUri": "s3://DOC-EXAMPLE-BUCKET-LOGGING/logs/app-level"
        },
        "managedPersistenceMonitoringConfiguration": {
            "enabled": false
        }
    }'
```

Overriding configurations during a job run

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

The granularity level when the merge occurs is as follows:

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

- **`MonitoringConfiguration`** - Configuration type, for example `s3MonitoringConfiguration`.

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

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

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

- **Override an existing configuration** - Provide the same configuration parameter in the `StartJobRun` request with your override values.
- **Add an additional configuration** - Add the new configuration parameter in the `StartJobRun` request with the values that you want to specify.
- **Remove an existing configuration** - To remove an application *runtime configuration*, provide the key for the configuration that you want to remove, and pass an empty declaration `{}` for the configuration. We don't recommend removing any classifications that contain parameters that are required for a job run. For example, if you try to remove the required properties for a Hive job, the job will fail.

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

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

Example override

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

```
aws emr-serverless start-job-run \
    --application-id your-application-id \
    --execution-role-arn your-job-role-arn \
    --job-driver '{
        "sparkSubmit": {
            "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
            "entryPointArguments": ["s3://DOC-EXAMPLE-BUCKET-OUTPUT/wordcount_output"]
        }
    }' \
    --configuration-overrides '{
        "applicationConfiguration": [
            {
                // Override existing configuration for spark-defaults in the
 application
                "classification": "spark-defaults",
                "properties": {
                    "spark.driver.cores": "2",
                    "spark.executor.cores": "1",
                    "spark.driver.memory": "4G",
                    "spark.executor.memory": "4G"
                }
            },
            {
                // Add configuration for spark-executor-log4j2
                "classification": "spark-executor-log4j2",
                "properties": {
                    "rootLogger.level": "error",
                    "logger.IdentifierForClass.name": "classpathForSettingLogger",
                    "logger.IdentifierForClass.level": "info"
                }
            },
            {
                // Remove existing configuration for spark-driver-log4j2 from the
 application
                "classification": "spark-driver-log4j2",
                "properties": {}
            }
        ],
```

```
        "monitoringConfiguration": {
            "managedPersistenceMonitoringConfiguration": {
                // Override existing configuration for managed persistence
                "enabled": true
            },
            "s3MonitoringConfiguration": {
                // Remove configuration of S3 monitoring
            },
            "cloudWatchLoggingConfiguration": {
                // Add configuration for CloudWatch logging
                "enabled": true
            }
        }
    }'
```

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

- The classification `spark-defaults` will be updated with the properties specified at the job level. Only the properties included in `StartJobRun` would be considered for this classification.
- The classification `spark-executor-log4j2` will be added in the existing list of classifications.
- The classification `spark-driver-log4j2` will be removed.
- The configurations for `managedPersistenceMonitoringConfiguration` will be updated with configurations at job level.
- The configurations for `s3MonitoringConfiguration` will be removed.
- The configurations for `cloudWatchLoggingConfiguration` will be added to existing monitoring configurations.

Customizing an EMR Serverless image

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

- Installs and configures packages that are optimized to your workloads. These packages might not be widely available in the public distribution of Amazon EMR runtime environments.

- Integrates EMR Serverless with current established build, test, and deployment processes within your organization, including local development and testing.
- Applies established security processes, such as image scanning, that meet compliance and governance requirements within your organization.
- Lets you use your own versions of JDK and Python for your applications.

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

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

Prerequisites

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

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ECRRepositoryListGetPolicy",
            "Effect": "Allow",
            "Action": [
```

```
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "ecr:DescribeImages"
            ],
            "Resource": "ecr-repository-arn"
        }
    ]
}
```

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

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

First, create a Dockerfile that begins with a FROM instruction that uses your preferred base image. After the FROM instruction, you can include any modification that you want to make to the image. The base image automatically sets the USER to hadoop. This setting might not have permissions for all the modifications you include. As a workaround, set the USER to `root`, modify your image, and then set the USER back to `hadoop:hadoop`. To see samples for common use cases, see Using custom images with EMR Serverless.

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

USER root
# MODIFICATIONS GO HERE

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

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

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

Step 2: Validate image locally

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

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

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

You should see an output similar to the following.

```
Amazon EMR Serverless - Image CLI
Version: 0.0.1
... Checking if docker cli is installed
... Checking Image Manifest
[INFO] Image ID: 9e2f4359cf5beb466a8a2ed047ab61c9d37786c555655fc122272758f761b41a
[INFO] Created On: 2022-12-02T07:46:42.586249984Z
[INFO] Default User Set to hadoop:hadoop : PASS
[INFO] Working Directory Set to  : PASS
[INFO] Entrypoint Set to /usr/bin/entrypoint.sh : PASS
[INFO] HADOOP_HOME is set with value: /usr/lib/hadoop : PASS
[INFO] HADOOP_LIBEXEC_DIR is set with value: /usr/lib/hadoop/libexec : PASS
[INFO] HADOOP_USER_HOME is set with value: /home/hadoop : PASS
[INFO] HADOOP_YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] HIVE_HOME is set with value: /usr/lib/hive : PASS
[INFO] JAVA_HOME is set with value: /etc/alternatives/jre : PASS
[INFO] TEZ_HOME is set with value: /usr/lib/tez : PASS
[INFO] YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] File Structure Test for hadoop-files in /usr/lib/hadoop: PASS
[INFO] File Structure Test for hadoop-jars in /usr/lib/hadoop/lib: PASS
[INFO] File Structure Test for hadoop-yarn-jars in /usr/lib/hadoop-yarn: PASS
[INFO] File Structure Test for hive-bin-files in /usr/bin: PASS
[INFO] File Structure Test for hive-jars in /usr/lib/hive/lib: PASS
[INFO] File Structure Test for java-bin in /etc/alternatives/jre/bin: PASS
[INFO] File Structure Test for tez-jars in /usr/lib/tez: PASS
----------------------------------------------------------------
Overall Custom Image Validation Succeeded.
----------------------------------------------------------------
```

Step 3: Upload the image to your Amazon ECR repository

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

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

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

Step 4: Create or update an application with custom images

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

Console

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

CLI

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

```
aws emr-serverless create-application \
--release-label emr-6.9.0 \
--type SPARK \
```

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

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

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

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

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

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

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

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

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

Considerations and limitations

When you work with custom images, consider the following:

- Use the correct base image that matches the type (Spark or Hive) and release label (for example, `emr-6.9.0`) for your application.
- EMR Serverless ignores `[CMD]` or `[ENTRYPOINT]` instructions in the Docker file. Use common instructions in the Docker file, such as `[COPY]`, `[RUN]`, and `[WORKDIR]`.
- You shouldn't modify environment variables `JAVA_HOME`, `SPARK_HOME`, `HIVE_HOME`, `TEZ_HOME` when you create a custom image.
- Custom images can't exceed 5 GB in size.

- If you modify binaries or jars in the Amazon EMR base images, it might cause application or job launch failures.
- The Amazon ECR repository should be in the same AWS Region that you use to launch EMR Serverless applications.

Configuring VPC access

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

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

Create application

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

VPCs

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

Subnets

Choose the subnets within the VPC that contains your data store. The **Create application** page lists all subnets for the data stores in your VPC.

The subnets selected must be private subnets. This means that the associated route tables for the subnets should not have internet gateways.

For outbound connectivity to the internet, the subnets must have outbound routes using a NAT Gateway. To configure a NAT Gateway, see Work with NAT gateways.

For Amazon S3 connectivity, the subnets must have a NAT Gateway or a VPC endpoint configured. To configure an S3 VPC endpoint, see Create a gateway endpoint.

For connectivity to other AWS services outside the VPC, such as Amazon DynamoDB, you must configure either VPC endpoints or a NAT gateway. To configure VPC endpoints for AWS services, see Work with VPC endpoints.

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

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

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

Security groups

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

> **Note**
>
> We recommend that you create a separate security group for EMR Serverless applications. This makes isolating and managing network rules more efficient. For example, to communicate with Amazon Redshift clusters, you can define the traffic rules between the Redshift and EMR Serverless security groups, as demonstrated in the example below.

Example Example — Communication with Amazon Redshift clusters

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

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

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

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

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

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

Configure application

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

View job run details

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

Best practices for subnet planning

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

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

You need subnets with mask size lower than or equal to 22 to provision 1,000 network interfaces. Any mask greater than 22 will not meet the requirement. For example, a subnet mask of /23 provides 512 IP addresses, while a mask of /22 provides 1024 and a mask of /21 provides 2048 IP addresses. Below is an example of 4 subnets with /22 mask in a VPC of /16 netmask that can be allocated to different Availability Zones. There is a difference of five between available and usable IP addresses because first four IP addresses and last IP address in each subnet is reserved by AWS.

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

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

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

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

Amazon EMR Serverless architecture options

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

Topics

- Using x86_64 architecture
- Using arm64 architecture (Graviton)
- Launching new applications with Graviton support
- Configuring existing applications to use Graviton
- Considerations when using Graviton

Using x86_64 architecture

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

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

Using arm64 architecture (Graviton)

Graviton processors are custom designed by AWS with 64-bit ARM Neoverse cores. The **arm64** architecture (also known as AArch64 or 64-bit ARM) uses Graviton processors to deliver better price performance for Spark and Hive workloads on Amazon EMR Serverless when compared with equivalent workloads running on the **x86_64** architecture option.

Launching new applications with Graviton support

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

AWS CLI

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

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

EMR Studio

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

Configuring existing applications to use Graviton

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

To convert an existing application from x86 to arm64

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

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

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

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

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

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

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

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