

Using Amazon EFS for AWS Lambda in your serverless applications

by [James Beswick](#) | on 18 JUN 2020 | in [Amazon EC2](#), [Amazon Elastic File System \(EFS\)](#), [Amazon VPC](#), [AWS Cloud9](#), [AWS Lambda](#), [AWS Serverless Application Model](#), [Serverless](#), [Technical How-To](#) | [Permalink](#) | [Comments](#) | [Share](#)

Serverless applications are event-driven, using ephemeral compute functions to integrate services and transform data. While [AWS Lambda](#) includes a 512-MB temporary file system for your code, this is an ephemeral scratch resource not intended for durable storage.

[Amazon EFS](#) is a fully managed, elastic, shared file system designed to be consumed by other AWS services, such as Lambda. With the release of Amazon EFS for Lambda, you can now easily share data across function invocations. You can also read large reference data files, and write function output to a persistent and shared store. There is no additional charge for using file systems from your Lambda function within the same VPC.

EFS for Lambda makes it simpler to use a serverless architecture to implement many common workloads. It opens new capabilities, such as building and importing large code libraries directly into your Lambda functions. Since the code is loaded dynamically, you can also ensure that the latest version of these libraries is always used by every new execution environment. For appending to existing files, EFS is also a preferred option to using [Amazon S3](#).

This blog post shows how to enable [EFS for Lambda](#) in your AWS account, and walks through some common use-cases.

Capabilities and behaviors of Lambda with EFS

EFS is built to scale on demand to petabytes of data, growing and shrinking automatically as files are written and deleted. When used with Lambda, your code has low-latency access to a file system where data is persisted after the function terminates.

EFS is a highly reliable NFS-based regional service, with all [data stored durably](#) across multiple Availability Zones. It is cost-optimized, due to no provisioning requirements, and no purchase commitments. It uses built-in lifecycle management to optimize between SSD-performance class and an infrequent access class that offer [92% lower cost](#).

EFS offers two [performance modes](#) – general purpose and MaxIO. General purpose is suitable for most Lambda workloads, providing lower operational latency and higher performance for individual files.

You also choose between two throughput modes – bursting and provisioned. The bursting mode uses a [credit system](#) to determine when a file system can burst. With bursting, your throughput is calculated based upon the amount of data you are storing. Provisioned throughput is useful when you need more throughput than provided by the bursting mode. Total throughput available is divided across the number of concurrent Lambda invocations.

The Lambda service mounts EFS file systems when the execution environment is prepared. This adds minimal latency when the function is invoked for the first time, often within hundreds of milliseconds. When the execution environment is already warm from previous invocations, the EFS mount is already available.

EFS can be used with [Provisioned Concurrency](#) for Lambda. When the reserved capacity is prepared, the Lambda service also configures and mounts EFS file system. Since Provisioned Concurrency executes any initialization code, any libraries or packages consumed from EFS at this point are downloaded. In this use-case, it's recommended to use provisioned throughput when configuring EFS.

The EFS file system is shared across Lambda functions as it scales up the number of concurrent executions. As files are written by one instance of a Lambda function, all other instances can access and modify this data,

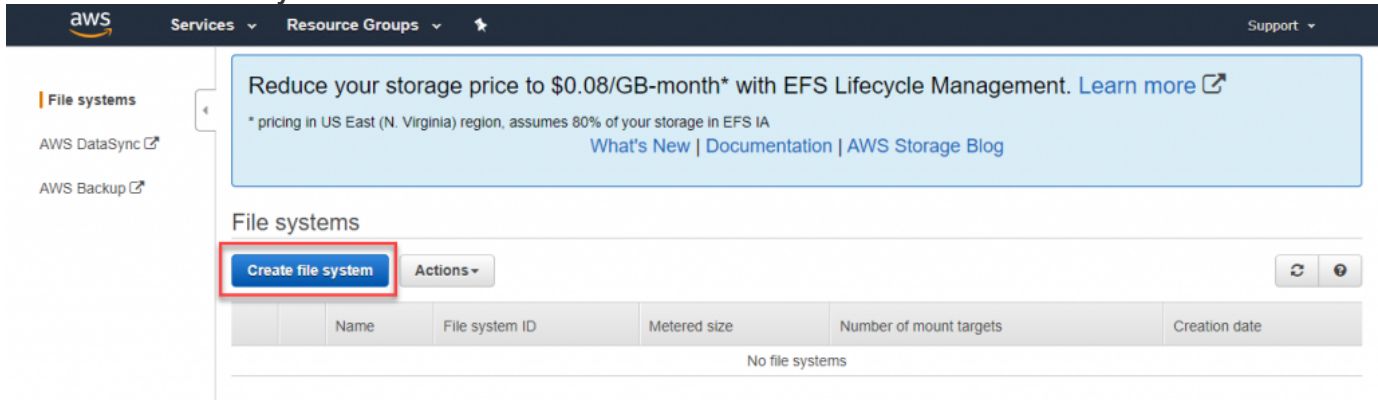
depending upon the access point permissions. The EFS file system scales with your Lambda functions, supporting up to 25,000 concurrent connections.

Creating an EFS file system

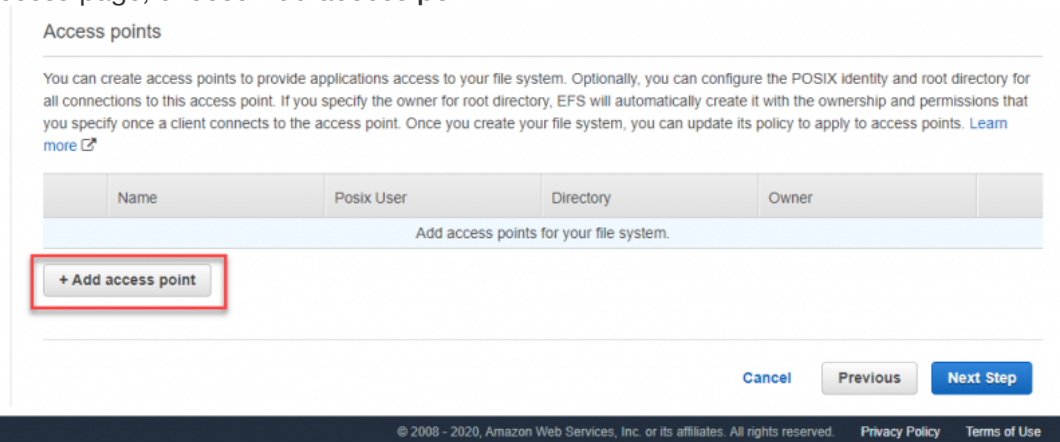
Configuring EFS for Lambda is straight-forward. I show how to do this in the [AWS Management Console](#) but you can also use the [AWS CLI](#), [AWS SDK](#), [AWS Serverless Application Model](#) (AWS SAM), and [AWS CloudFormation](#). EFS file systems are always created within a customer VPC, so Lambda functions using the EFS file system must all reside in the same VPC.

To create an EFS file system:

1. Navigate to the [EFS console](#).
2. Choose **Create File System**.



3. On the *Configure network access* page, select your preferred VPC. Only resources within this VPC can access this EFS file system. Accept the default mount targets, and choose **Next Step**.
4. On *Configure file system settings*, you can choose to enable [encryption of data at rest](#). Review this setting, then accept the other defaults and choose **Next Step**. This uses bursting mode instead of provisioned throughput.
5. On the *Configure client access* page, choose **Add access point**.



6. Enter the following parameters. This configuration creates a file system with open read/write permissions – read more about [settings to secure your access points](#). Choose **Next Step**.

Access points

You can create access points to provide applications access to your file system. Optionally, you can configure the POSIX identity and root directory for all connections to this access point. If you specify the owner for root directory, EFS will automatically create it with the ownership and permissions that you specify once a client connects to the access point. Once you create your file system, you can update its policy to apply to access points. [Learn more](#)

Name	Posix User	Directory	Owner
LambdaEFS	1000 : 1000	/efs	1000 : 1000 (777)

Name

LambdaEFS

User ID

1000

Group ID

1000

Secondary Group IDs

Path

/efs

Owner User ID

1000

Owner Group ID

1000

Permissions

777

+ Add access point

- On the *Review and create* page, check your settings and choose **Create File System**.
- In the EFS console, you see the new file system and its configuration. Wait until the *Mount target state* changes to *Available* before proceeding to the next steps.

Alternatively, you can use CloudFormation to create the EFS access point. With the `AWS::EFS::AccessPoint` resource, the preceding configuration is defined as follows:

YAML

```
AccessPointResource:
  Type: 'AWS::EFS::AccessPoint'
  Properties:
    FileSystemId: !Ref FileSystemResource
    PosixUser:
      Uid: "1000"
      Gid: "1000"
    RootDirectory:
      CreationInfo:
```

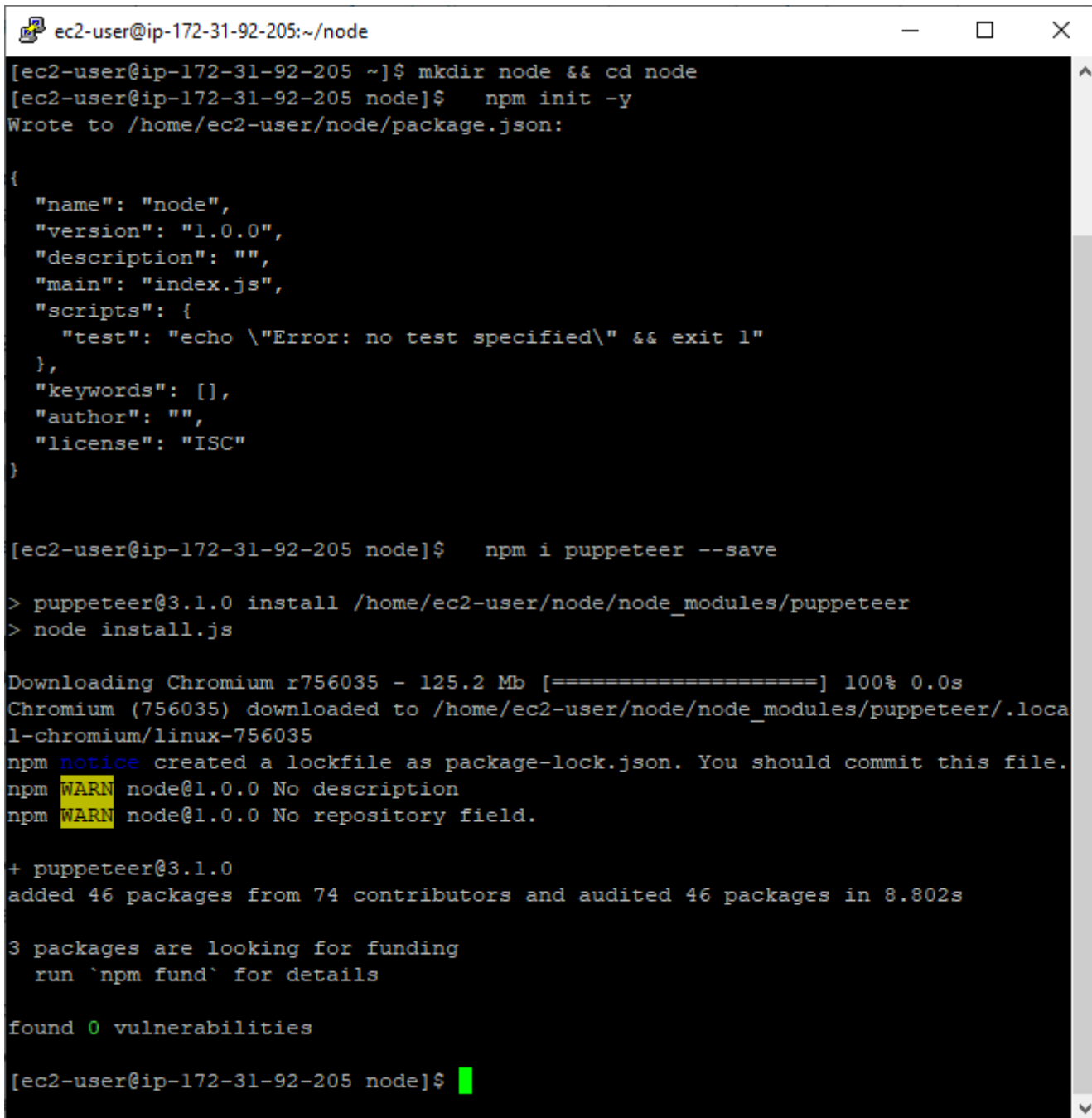
For more information, see the example setup template in the [code repository](#).

Working with AWS Cloud9 and Amazon EC2

You can mount EFS access points on [Amazon EC2](#) instances. This can be useful for browsing file systems contents and downloading files from other locations. The [EFS console](#) shows customized mount instructions directly under each created file system:

Bash

```
mkdir node && cd node
npm init -y
```

A terminal window titled 'ec2-user@ip-172-31-92-205:~/node' showing the execution of 'mkdir node && cd node' and 'npm init -y'. It displays the generated package.json file. Then, 'npm i puppeteer --save' is run, showing the download of puppeteer@3.1.0 and its dependencies. The terminal output includes a notice about a lockfile, warnings about missing description and repository fields, and a summary of installed packages and vulnerabilities.

```
ec2-user@ip-172-31-92-205:~/node
[ec2-user@ip-172-31-92-205 ~]$ mkdir node && cd node
[ec2-user@ip-172-31-92-205 node]$ npm init -y
Wrote to /home/ec2-user/node/package.json:

{
  "name": "node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

[ec2-user@ip-172-31-92-205 node]$ npm i puppeteer --save

> puppeteer@3.1.0 install /home/ec2-user/node/node_modules/puppeteer
> node install.js

Downloading Chromium r756035 - 125.2 Mb [=====] 100% 0.0s
Chromium (756035) downloaded to /home/ec2-user/node/node_modules/puppeteer/.local-chromium/linux-756035
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN node@1.0.0 No description
npm WARN node@1.0.0 No repository field.

+ puppeteer@3.1.0
added 46 packages from 74 contributors and audited 46 packages in 8.802s

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

[ec2-user@ip-172-31-92-205 node]$
```

You can then use this package from a Lambda function connected to this folder in the EFS file system. You include the Puppeteer package with the mount path in the require declaration:

```
const puppeteer = require
('/mnt/efs/node/node_modules/puppeteer')
```

In Node.js, to avoid changing declarations manually, you can add the EFS mount path to the Node.js module search path by using [app-module-path](#). Lambda functions [support a range of other runtimes](#), including Python, Java, and Go. Many other runtimes offer similar ways to add the EFS path to the list of default package locations.

There is an important difference between using packages in EFS compared with Lambda layers. When you use Lambda layers to include packages, these are downloaded to an immutable code package. Any changes to the underlying layer do not affect existing functions published using that layer.

Since EFS is a dynamic binding, any changes or upgrades to packages are available immediately to the Lambda function when the execution environment is prepared. This means you can output a build process to an EFS mount, and immediately consume any new versions of the build from a Lambda function.

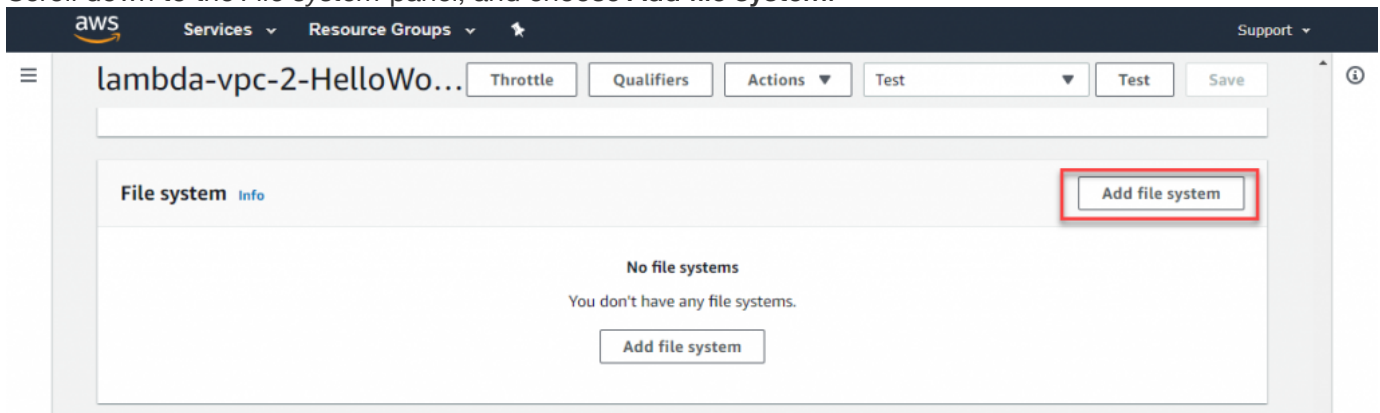
Configuring AWS Lambda to use EFS

Lambda functions that access EFS must run from within a VPC. Read this guide to learn more about [setting up Lambda functions to access resources from a VPC](#). There are also [sample CloudFormation templates](#) you can use to configure private and public VPC access.

The execution role for Lambda function must provide access to the VPC and EFS. For development and testing purposes, this post uses the `AWSLambdaVPCAccessExecutionRole` and `AmazonElasticFileSystemClientFullAccess` managed policies in IAM. For production systems, you should use [more restrictive policies](#) to control access to EFS resources.

Once your Lambda function is configured to use a VPC, next configure EFS in Lambda:

1. Navigate to the [Lambda console](#) and select your function from the list.
2. Scroll down to the *File system* panel, and choose **Add file system**.



3. In the *File system* configuration:
 - From the *EFS file system* dropdown, select the required file system. From the *Access point* dropdown, choose the required EFS access point.
 - In the *Local mount path*, enter the path your Lambda function uses to access this resource. Enter an absolute path.
 - Choose **Save**.

[Lambda](#) > [Functions](#) > [EFS test](#) > Add file system

Add file system

File system

You can associate an existing Amazon Elastic File System (Amazon EFS) file system with your function. Visit the Amazon EFS console to [create a new file system](#).

EFS file system
Choose an existing EFS file system to use with your Lambda function.

console-1d97edb3-75e7-40fa-a41b-0721efa7577e

arn:aws:elasticfilesystem:us-east-1:780018668030:file-system/fs-d74f2054

Owner: 780018668030 Throughput Mode: bursting

fs-d74f2054 ▼

Access point
An access point that is used to mount a network file system and integrates with IAM to control access.

LambdaEFS

arn:aws:elasticfilesystem:us-east-1:780018668030:access-point/fsap-0ce0f8d77682aa6df

POSIX uid: 1000 POSIX gid: 1000 Remote path: /test

fsap-0ce0f8d77682aa6df ▼

Local mount path
Only absolute paths are supported.

/mnt/efs

Cancel

Save

The *File system* panel now shows the configuration of the EFS mount, and the function is ready to use EFS. Alternatively, you can use an [AWS Serverless Application Model \(SAM\)](#) template to add the EFS configuration to a function resource:

YAML

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      ...
      FileSystemConfigs:
```

To learn more, see the [SAM documentation](#) on this feature.

Example applications

You can view and download these examples from this [GitHub repository](#). To deploy, follow the instructions in the repo's [README.md](#) file.

1. Processing large video files

The [first example](#) uses EFS to process a 60-minute MP4 video and create screenshots for each second of the recording. This uses the [FFmpeg](#) Linux package to process the video. After copying the MP4 to the EFS file location, invoke the Lambda function to create a series of JPG frames. This uses the following code to execute FFmpeg and pass the EFS mount path and input file parameters:

JavaScript

```
const os = require('os')

const inputFile = process.env.INPUT_FILE
const efsPath = process.env.EFS_PATH

const { exec } = require('child_process')

const execPromise = async (command) => {
  console.log(command)
  return new Promise((resolve, reject) => {
    const ls = exec(command, function (error, stdout, stderr) {
      if (error) {
        console.log('Error: ', error)
        reject(error)
      }
      console.log('stdout: ', stdout);
      console.log('stderr: ', stderr);
    })
    resolve(stdout)
  })
}
```

In this example, the process writes more than 2000 individual JPG files back to the EFS file system during a single invocation:

```
[ec2-user@ip-172-31-92-205 test]$ ls video.MP4
video.MP4
[ec2-user@ip-172-31-92-205 test]$ ls *.jpg | wc -l
2254
[ec2-user@ip-172-31-92-205 test]$
```

2. Archiving large numbers of files

Using the output from the first application, the [second example](#) creates a single archive file from the JPG files. The code uses the Node.js [archiver](#) package for processing:

JavaScript

```
const outputFile = process.env.OUTPUT_FILE
const efsPath = process.env.EFS_PATH

const fs = require('fs')
const archiver = require('archiver')

// The Lambda handler
exports.handler = function (event) {

  const output = fs.createWriteStream(`${efsPath}/${c
  const archive = archiver('zip', {
    zlib: { level: 9 } // Sets the compression level.
  })

  output.on('close', function() {
    console.log(archive.pointer() + ' total bytes')
  })
```

After executing this Lambda function, the resulting ZIP file is written back to the EFS file system:

```
[ec2-user@ip-172-31-92-205 test]$ ls video.MP4
video.MP4
[ec2-user@ip-172-31-92-205 test]$ ls *.jpg | wc -l
2254
[ec2-user@ip-172-31-92-205 test]$ ls -la video.zip
-rw-rw-r-- 1 ec2-user ec2-user 16006957 May 28 12:43 video.zip
[ec2-user@ip-172-31-92-205 test]$
```

3. Unzipping archives with a large number of files

The [last example](#) shows how to unzip an archive containing many files. This uses the Node.js [unzipper](#) package for processing:

JavaScript

```
const inputFile = process.env.INPUT_FILE
const efsPath = process.env.EFS_PATH
const destinationDir = process.env.DESTINATION_DIR

const fs = require('fs')
const unzipper = require('unzipper')

// The Lambda handler
exports.handler = function (event) {
```

Once this Lambda function is executed, the archive is unzipped into a destination direction in the EFS file system. This example shows the screenshots unzipped into the *frames* subdirectory:

```
[ec2-user@ip-172-31-92-205 test]$ ls *.jpg | wc -l
2254
[ec2-user@ip-172-31-92-205 test]$ ls -la video.zip
-rw-rw-r-- 1 ec2-user ec2-user 16006957 May 28 12:43 video.zip
[ec2-user@ip-172-31-92-205 test]$ cd frames/mnt/efs/
[ec2-user@ip-172-31-92-205 efs]$ ls -la | wc -l
2257
[ec2-user@ip-172-31-92-205 efs]$
```

Conclusion

EFS for Lambda allows you to share data across function invocations, read large reference data files, and write function output to a persistent and shared store. After configuring EFS, you provide the Lambda function with an access point ARN, allowing you to read and write to this file system. Lambda securely connects the function instances to the EFS mount targets in the same Availability Zone and subnet.

EFS opens a range of potential new use-cases for Lambda. In this post, I show how this enables you to access large code packages and binaries, and process large numbers of files. You can interact with the file system via EC2 or AWS Cloud9 and pass information to and from your Lambda functions.

EFS for Lambda is supported at launch in APN Partner solutions, including [Epsagon](#), [Lumigo](#), [Datadog](#), [HashiCorp Terraform](#), and [Pulumi](#). To learn more about how to use EFS for Lambda, see the [AWS News Blog post](#) and [read the documentation](#).