Video Transcoding on AWS Serverless using Lambda

V1.0

Tom Pflaum

TPFL Consulting

tomp@tpflconsulting.com

www.tpflconsulting.com

November, 5th 2024

Overview

There are many ways to transcode video on AWS: using on-prem transcode software on EC2, using AWS Elemental Media Convert, using 3rd part SaaS services, etc.

This article outlines the advantages and properties of using AWS' serverless compute platform Lambda to run ffmpeg.

Lambda and FFMPEG

"Serverless" means that you don't have to worry about spinning up compute instances (EC2 instances) to run a task. You define a trigger (in this example a watch folder on an S3 bucket) that will cause AWS to automatically start a Lambda function, which in this case will run ffmpeg. Once the transcoding finishes, the Lambda function terminates. (It can trigger other Lambda functions if future steps are required.)

There are three main advantages of Lambda vs. using EC2 instances:

- You get automatic scalability. Each file uploaded to the S3 bucket will trigger a separate Lambda function. Hundreds or even thousands of Lambda functions can run in parallel, at the same time.
- You don't have to worry about spinning up and down EC2 instances and associated storage making sure you are not wasting money on idling resources.
- The watch folder is always active without any charges from AWS.

The Lambda function used for this test is using the Python3 runtime environment. That means that when the Lambda function is started, it runs a Python script.

The compute environment for the Python script is by default 128Mbytes of main memory, 1 vCore and 512 Mbyte of Ephemeral (block) storage.

Since transcoding is compute intensive, I increased the memory to the maximum of 10240Mbytes of storage, which also increases the vCores to 6. (we don't need the storage, but the number of cores are tied to the memory).

To get access to ffmpeg, a statically linked version is provided as a Layer (which in this case is a ZIP file that gets extracted to /opt when the Lambda function is started.

Benchmarks

Source: MP4, 1920x1080, 30fps, duration 15 minutes

Output: MP4, x264, 1mbit/s, 600x360, 30fps

Lambda Performance (10240Mb memory, 6 vCores)

Startup time: 2 seconds

Transcode time: 199 seconds,

Frames per second: 135 fps

On prem workstation, 24 Threads (vCore)

Transcode time: 48 seconds, 562 fps

Rough math:

Workstation has 4x the vCores compared to the Lambda environment, and it is 4.5x faster.

Thus, the performance per vCore is similar to what I am seeing on an Intel Workstation (rough estimate.)

Cost:

Lamba:

Cost per ms compute 10240Gb (6 vCores):	$0.0000001667
Cost per ms, 5Gb epheral storage:	$0.0000000309
Billed duration for example:	219923 ms
Total cost transcoding 15 minutes of video:	$0.043

Using published AWS Elemental MediaConvert pricing:

1 minute of AVC, SD:	0.0075 per minute
Total cost transcoding 15 minutes of video:	$0.1125

Advantages

The performance of ffmpeg on Lambda is comparable to a 6 core on-prem system, which I consider reasonable from a turnaround time perspective.

Keep in mind that AWS is able to spin up MANNY of the ffmpeg-Lambda function at the same time. If you have, for example, a user-generated video workflow with lots of file being processed it will automatically scale up 100drets of concurrent transcodes.

When comparing to AWS MediaConvert, you are using a transcoder that you are already familiar with. You also control which version of ffmpeg is being used, and if other command line tools are needed, they can be easily added to the Layer.

From a pricing perspective, it is generally cheaper than other options.

Technical Description

To allow Lambda function to run ffmpeg, I added a Layer. A Layer is a ZIP file that gets extracted to /opt when the Lambda function starts. In this ZIP file contains a statically linked version of ffmpeg.

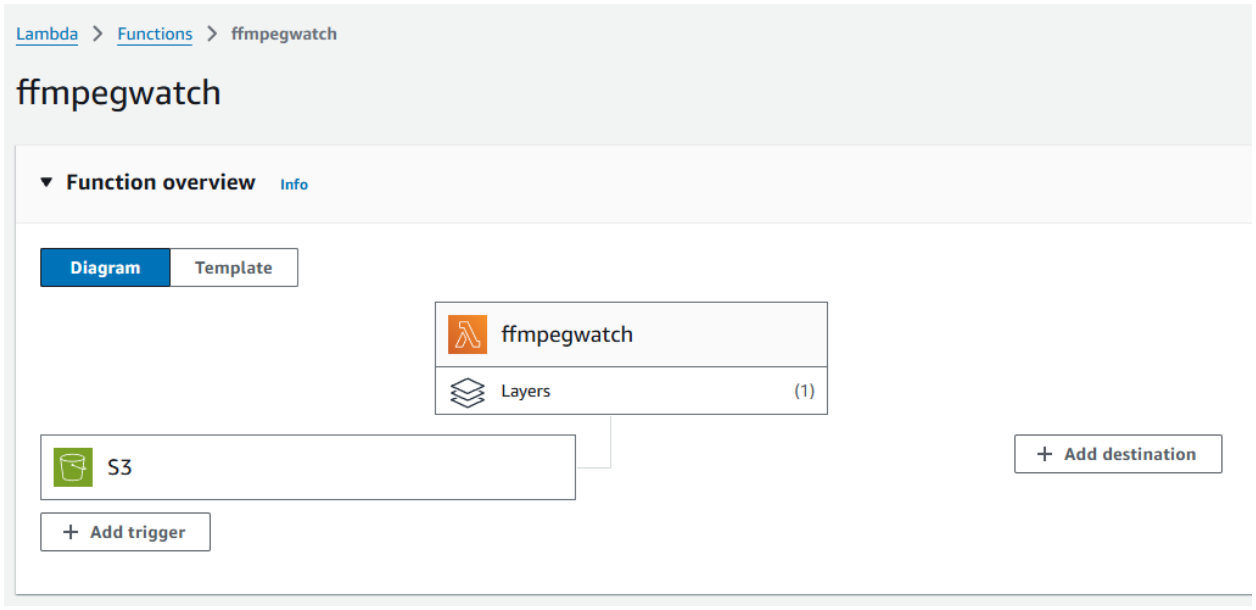
To start the Lambda function, I am using a watch folder (meaning a Bucket on S3) that will kick off the Lambda function when a new file complets uploading to the Bucket.

To get good performance on the Lambda function, you have to increase the memory available to the Lambda function. The number of cores is tied to the memory you specify. The maximum is 10240GB of memory, which will also increase the number of vCores to 6. (by default the memory is 512MB and a single core, so you really want to change it.) While the default network speed is not very high (I measured about 0.5Gbit/sec from S3) it is adequate for this use-case. (It is possible to increase the networking, but I did not look further into it).

The Code

I wrote the Lambda function using Python3.

- The function is started, receiving a message indicating which object (file) on S3 started the Lambda function.
- A signed URL for the S3 object (the video file) that started the Lambda function is generated (using Bodo3)
- A path to the output location of the transcoded file is hardcoded to be /temp/foo.mp4. It will be renamed when uploaded to S3.
- ffmpeg is started.
  The source is the signed URL generated earlier. ffmpeg can transcode files directly from S3 without having to localize the file.
  Encoding parameters are specified. In this case a 600x360 poxy size. Any parameter ffmpeg supports can be specified here. In this case a MP4 file with the default bit-rate will be generated.
  The output location is /tmp/foo.mp4
- ffmpeg finishes the transcode is complete, the output file is uploaded to a different S3 Bucket sing Bodo3 (as AWS warns: don't use the watch folder Bucket, as this would result in an infinite loop.)



Code:

```
import json
import os
import boto3
import urllib.parse

def lambda_handler(event, context):
    # boto3 client
    s3 = boto3.client("s3")
    # Get source bucket and key (file)
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'], encoding='utf-8')
    # Create a signed URL for the source file
    s3_url = s3.generate_presigned_url(
        'get_object',
        Params= {'Bucket': bucket_name, 'Key': key},
        ExpiresIn=60*15  # URL expiration time in seconds
    )

    # Hardcoded output file. The storge will be new for every invocation
    lambda_output_file_path = "/tmp/foo.mp4"

    # transcoding
    os.system(
        f"/opt/ffmpeglib/ffmpeg -t 900 -i \"{s3_url}\" -s 600x360 {lambda_output_file_path}"
    )

    # uploading transcoded file to a different bucket
    output_bucket_name = "myoutput_bucket"

    s3.upload_file(
        Bucket=output_bucket_name,
        Key=lambda_output_file_path.split("/")[-1],
        Filename=lambda_output_file_path,
    )
    return {"statusCode": 200, "body": json.dumps("ffmpeg completed.")}
```

Further Discussions

More details on how to run ffmpeg in lambda function can be found in this AWS blog post by Abbas Nemer:

[Processing user-generated content using AWS Lambda and FFmpeg | AWS for M&E Blog]

One of the limitations of the approach outlined in this paper is that the output file needs to fit in the ephemeral storage (block storage) allocated to the Lambda function. Increasing the storage will increase the cost and it is limited to 10GB.

One way to avoid this problem is to "stream" the ffmpeg output directly to S3. You can find a discussion of this approach here:

[amazon s3 - Stream ffmpeg transcoding result to S3 - Stack Overflow]

However, many video formats (in particular MP4 and MOV) require the header information to be updated once transcoding is completed. This is not something that can be done using the "pipe" output of FFMPEG. This results in files that are usable by some programs but not by others.

In general, I recommend not using this approach when the output file exceeds 10GB and use a different approach instead.