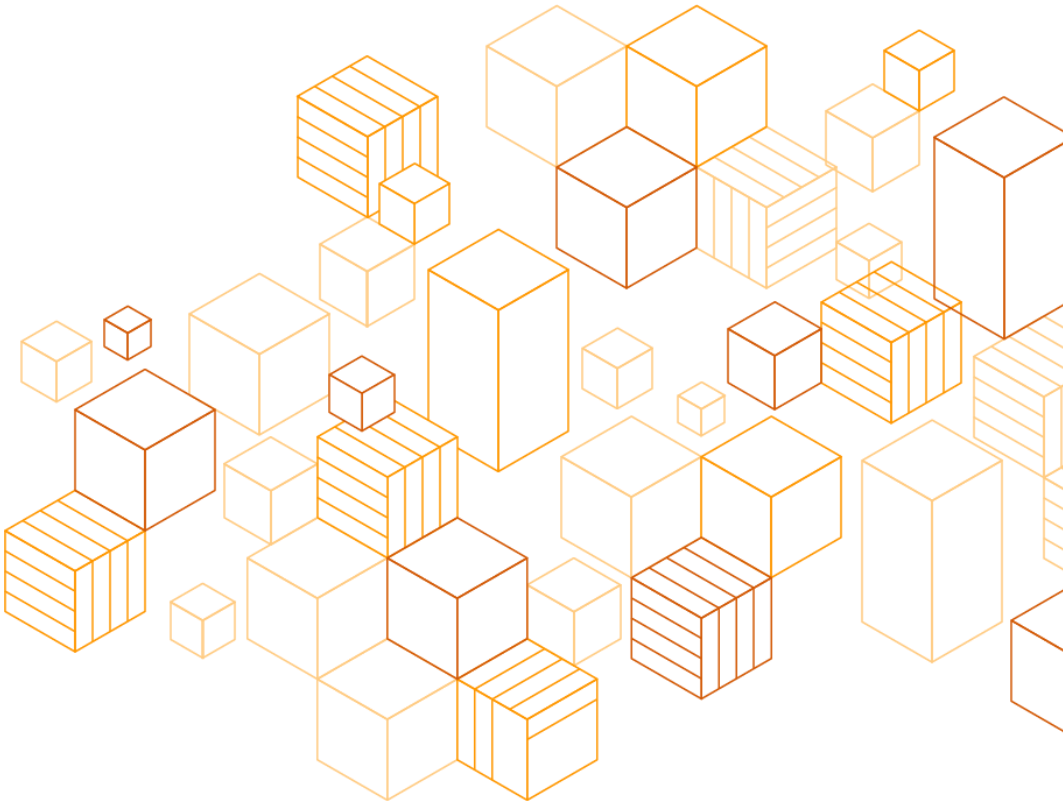
Create a Serverless Content Syndication Pipeline with AWS Step Functions

Technical Guide

Published March 11, 2021



Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

About this guide

The entertainment industry is undergoing a fundamental shift from traditional media concepts to digital on-demand offerings. As a consequence, many of our customers in the media and entertainment industry are looking for ways to increase the reach of their digital content. In a business model typically referred to as “syndication”, content is distributed to a large number of partners that integrate it into their own consumer products. These syndication workflows are typically event driven and stateless in nature. This guide outlines how customers can build a cost-efficient and scalable reference architecture based on the Amazon Web Services (AWS) serverless platform.

Overview

An important goal for many of our customers in the media and entertainment industry is to increase the reach of their digital content. In a business model typically referred to as “syndication”, content is distributed to a large number of partners that integrate it into their own consumer products. A typical example for such a business-to-business-to-consumer (B2B2C) model is a video streaming service that extends its reach by being integrated into cable provider set-top boxes.

While this is an effective way to reach a wider audience, it often requires customizations to deliver content according to partner specifications. This adds complexity and requires source modifications such as:

- Image transformation (orientation, overlay images, filtering)
- Video formats (codecs, resolutions, cut versions, ad stitching)
- Delivery protocols (Amazon Simple Storage Service [Amazon S3], FTP, Signiant, Aspera)
- Metadata format (XML, JSON, XLS, CSV)

A characteristic of these syndication workflows is that they are event-driven and stateless. The systems responsible for handling syndication workflows can take advantage of these characteristics by using a serverless architecture. Instead of paying for servers running idle for a significant portion of time, the AWS serverless platform enables users to design an event-driven architecture that has a number of benefits compared to a traditional approach. These benefits include:

- **Cost-effectiveness** — Syndication applications in a traditional server setup idle a significant amount of time while they wait for a processing task to arrive. Being able to consume only the required compute capacity when it is needed improves the cost-effectiveness of the whole solution.

- **Scalability** — Traffic patterns in syndication applications are typically spiky. Ingest or update operations tend to concentrate on certain dates during which a large amount of content becomes available. The spiky traffic patterns get amplified by the nature of workload where one input leads to many different outputs. An example of this spike in traffic is a new season of a popular TV show on a video streaming service. With *n* episodes in the season and *m* partners, this content item leads to *n*m* invocations of the same process at the same time. The result is that systems are either over-provisioned to account for hard-to-predict peak loads, or are incapable of keeping up with demand.
- **Maintainability** — With an increasing number of partners, system complexity grows and becomes harder to maintain, because onboarding new partners requires changes to the existing application. This dependency reduces the velocity of onboarding new partners, as deployment of those new features must be coordinated with the existing system.

  This guide discusses an architecture that addresses these limitations by using AWS serverless technologies to build an event-driven syndication workflow that is cost-efficient, highly scalable, and easy to maintain.

Before you begin

While this guide includes detailed instructions and sample code, the intended audience is already familiar with AWS. This guide assumes an understanding of core concepts like service roles and does not explain these aspects in detail.

Other prerequisites for this guide include:

- An AWS account.
- Installed and authenticated AWS Command Line Interface (AWS CLI).
- Installed and set up AWS Cloud Development Kit (AWS CDK) for TypeScript.

Cost

One of the major advantages of the solution outlined in this document is its cost effectiveness compared with traditional solutions, which typically run idle servers that wait for new assets to process.

In contrast, with AWS Lambda, you pay only for the compute time you consume, so you're never paying for over-provisioned infrastructure. Three main consumption-driven factors contribute to the cost of this solution:

1. **Running business logic with AWS Lambda**. You are charged for every millisecond your code runs and the number of times your code is triggered. AWS Lambda automatically scales your application by running code in response to each event. **See AWS Lambda Pricing to learn more**.
2. **Video transcoding with AWS Elemental MediaConvert**. This workflow uses AWS Elemental MediaConvert with on-demand pricing to transcode video files. You pay only for what you use and there are no minimum fees. Charges are based on per-minute usage in each output. See AWS Elemental MediaConvert Pricing to learn **more**.
3. **Storing files on Amazon Simple Storage Service (Amazon S3)**. This guide uses Amazon S3 to store both input and output artifacts of our workflows. With Amazon S3, you pay **only** for what you use. Amazon S3 **enables** you to scale your storage resources up and down to meet fluctuating demands, without upfront investments or resource procurement cycles. See Amazon S3 Pricing to learn more.

To ensure that you are not billed for any accidental consumption of resources you created as part of this guide, be sure to clean up the stack you created by running `cdk destroy`.

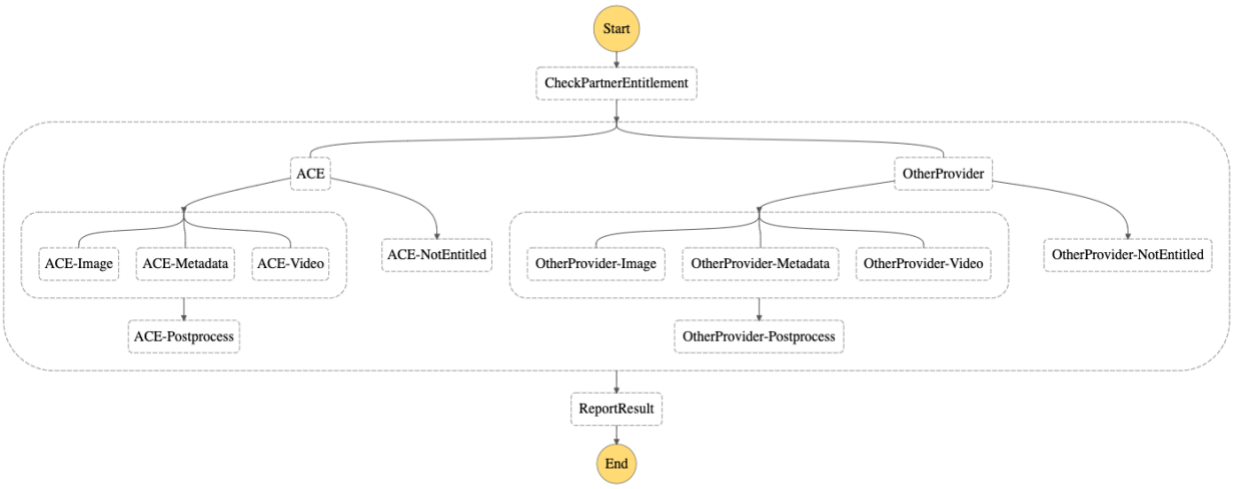
S3 buckets that contain objects are not deleted when a stack is destroyed. If you tested the workflow, this is likely the case. For those buckets you can run the following **destructive and irreversible** AWS CLI command:

```
aws s3 rb s3://<bucket-name> --force
```

Architecture overview

This example implements a complete workflow for a fictitious media company called Example Corp. Media *(ECM)* that syndicates video assets to their partner AnyCompany Entertainment (ACE). The workflow starts with an upload of the source files to an S3 bucket. It ends with the video, image, and metadata of the video asset being processed according to partner specs and stored in a second S3 bucket that serves as the delivery destination.

This guide focuses on the workflow of ExampleCorp to AnyCompany, and also briefly covers how additional partners can be integrated into the syndication process. The AnyCompany branch is fully functional, while the OtherPartner is a stub implementation that you can extend with your own business logic.



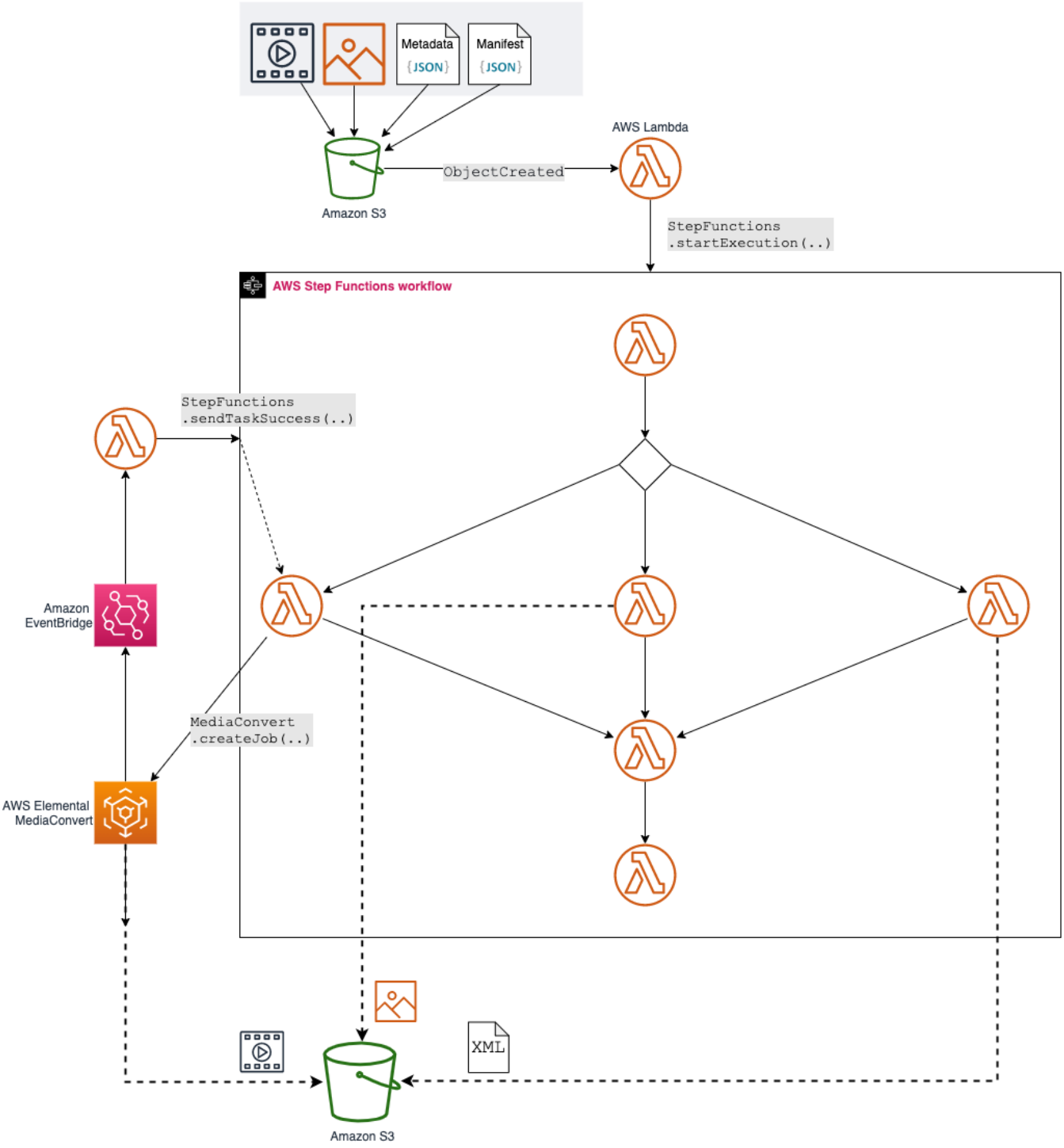
Syndication workflow

The preceding syndication diagram depicts the workflow built with AWS Step Functions, a serverless function orchestrator that makes it easy to sequence AWS Lambda functions and multiple AWS Services into business-critical applications.

With Step Functions, you can create and run a series of checkpointed and event-driven workflows that maintain the application state. The output of one step acts as an input to the next. Each step in your application runs in order, as defined by your business logic. Step Functions automatically manages error handling, retry logic, and state. With its built-in operational controls, Step Functions manages sequencing, error handling, retry logic, and state, removing significant operational burdens from your team.

You will use the AWS Cloud Development Kit (CDK) with TypeScript to express the following architecture as Infrastructure as Code, and employ further serverless technologies like AWS Lambda or and AWS Elemental MediaConvert for our business processes. Using the CDK enables you to apply software engineering best practices (such as code reusability, Test Driven Development, and type safety) not only to your application code, but extend them to the infrastructure definition as well.

The code snippets used in this guide are simplified to illustrate key concepts. The complete and functional code is available on GitHub. After you deploy the CDK project, the following architecture will be created in your AWS account:



Architecture of the content syndication workflow

Procedural sections

Ingest

In syndication workflows, the goal is to distribute media files from a single source to multiple independent destinations. This process typically starts by source files being made available for distribution in the ingest phase.

Here, the workflow is triggered by uploading files to Amazon S3. Amazon S3 is object storage built to store and retrieve any amount of data from anywhere on the internet. It's a simple storage service that offers an extremely durable, highly available, and infinitely scalable data storage infrastructure at low cost.

To upload your data to Amazon S3, you must first create an S3 bucket to store the source files that go into your workflow as input. You then configure this source bucket to call an AWS Lambda function for every created object. AWS Lambda lets you run code without provisioning or managing servers, and you pay only for the compute time you consume.

Amazon S3 has a native integration with Lambda that enables you to run code as a response to events in S3. In this case, you use a Lambda function to evaluate if all required files for the source asset are already uploaded to the bucket. Files uploaded to the bucket must to adhere to the following structure:
`s3://<bucketname>/<assetId>/*`

In particular, the Lambda function expects a file `s3://<bucketname>/<assetId>/manifest.json` that describes where to find video, image, and metadata information for this asset.

```
{
  "Video": "video.mp4",
  "Image": "image.jpg",
  "Metadata": "metadata.json"
}
```

If Lambda determines that the bundle is complete, it starts the syndication workflow by calling `StepFunctions.startExecution` via the AWS SDK for JavaScript.

```
StepFunctions.startExecution({
    input: JSON.stringify({..}),
    name: `S3UploadTriggeredExecution${Date.now()}`,
```

```
    stateMachineArn: "arn:aws:.."
}).promise();
```

The following CDK code defines an S3 bucket, Lambda function, and Step Function state machine. It also defines permissions and describes relationships between the components. S3 requires permission to invoke the Lambda function on your behalf for new S3 events; the Lambda function requires permission to read from the S3 bucket and start the state machine.

```
const stateMachine = .. // Defined elsewhere

const objectCreatedLambdaHandler = new lambda.Function(this,
`FileUploadLambda`, {
   runtime: lambda.Runtime.NODEJS_12_X,
    code: lambda.Code.fromAsset(..),
   // More config params
});

objectCreatedLambdaHandler.addToRolePolicy(new
iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ['states:StartExecution'],
    resources: [stateMachine.stateMachineArn]
}))

sourceBucket.addObjectCreatedNotification(new
s3notifications.LambdaDestination(objectCreatedLambdaHandler))
sourceBucket.grantRead(objectCreatedLambdaHandler)
```

Processing

Identify relevant partners

Typically, not every partner is entitled to receive every asset, so the first step in your workflow is to determine which partner-specific processing steps are actually required. In a real-world scenario, this information is typically retrieved from a license management system, but for this sample, the Lambda function returns a static result:

```
function DetermineRelevantDestination(event: any) {
    event.Destinations = {
        ACE: true,
        OtherProvider: false
    };
```

```
    return event;
}
```

You then use a Step Functions Choice state to evaluate the output of this Lambda function and determine which path to follow. In this case, the check for Provider A passes, and the state machine enters the `ProviderACEParallelProcessing` flow. For `OtherProvider`, the check fails, and the branch for Provider B ends with no action required.

```
isProviderEnabled(providerName: string) {
    return
sfn.Condition.booleanEquals(`$.Destinations.${providerName}`,
true);
}

const noActionRequiredForACE = new sfn.Pass(this,
"NoActionRequiredACE", {
    result: Result.fromObject(..)
})

const noActionRequiredForOther = new sfn.Pass(this,
"NoActionRequiredOther", {
    result: Result.fromObject(..)
})

const ProviderACEParallelProcessing = // Processing steps for ACE
const ProviderOtherParallelProcessing = // Processing steps for
OtherProvider

const isProviderACE = new sfn.Choice(this, 'ProviderACE')
    .when(isProviderEnabled("ACE"), ProviderACEParallelProcessing)
    .otherwise(noActionRequiredForACE);

const isProviderOther = new sfn.Choice(this, 'ProviderOther')
    .when(isProviderEnabled("OtherProvider"),
ProviderOtherParallelProcessing)
    .otherwise(noActionRequiredForOther);
```

Parallel processing of video, image, and metadata

After determining the relevant partners for a video asset, the workflow now starts processing those assets for each partner. ACE assumes the following specification:

- **Video** — must be transcoded to 720p using the H265 codec.

- **Metadata** — must be converted to XML.
- **Image** — must be black and white with an AWS logo in the bottom-left corner.

Transformation of the image

As a next step, create three Lambda functions that process video, image, and metadata according to ACE specs. All of those functions require permissions to read from the source bucket and write to the destination bucket.

```
const theFunction = new lambda.Function(this,
`CDKFunctionIdentifier`, {
   runtime: lambda.Runtime.NODEJS_12_X,
    code: lambda.Code.fromAsset(..),
   // ..
});

sourceBucket.grantRead(theFunction)
providerABucket.grantReadWrite(theFunction)
```

Images and metadata

Images and metadata are handled in a similar fashion, with a synchronous Lambda function each. Each of these Lambda functions contains the business logic required to transform the source data into the partner-specific formats. This sample code depends on two external libraries to process the image and metadata:

- **Jimp** is an image processing library for Node written entirely in JavaScript, with zero native dependencies.

- **xml-js** is a convert utility that allows us to convert XML to JSON in a simple way.

The recommended way to manage those dependencies are Lambda Layers. AWS released Lambda Layers to provide a mechanism to externally package dependencies that can be shared across multiple Lambda functions. Lambda layers reduce lines of code and size of application artifacts, and simplify dependency management.

With the CDK, creating a Lambda Layer and adding it to a Lambda function can be done in just a few lines of code.

```
const dependencies = new lambda.LayerVersion(this, '..', {
   code: lambda.Code.fromAsset(path.join(__dirname, '..', '..',
'src', 'lib')),
    compatibleRuntimes: [lambda.Runtime.NODEJS_12_X]
   // ..
});

new lambda.Function(this, `..`, {
   runtime: lambda.Runtime.NODEJS_12_X,
   code: lambda.Code.fromAsset(..),
   layers: [dependencies],
   // ..
});
```

This snippet describes that the content of folder `src/lib` should be used for this Lambda Layer. AWS Lambda searches for subfolders that match the compatible runtimes, in this case `src/lib/nodejs`. All dependencies found (`node_modules` for `nodejs`) for compatible runtimes are injected into the runtime environment of the Lambda functions to which the Lambda Layer is attached.

For the Lambda function that transforms the image, it's sensible to increase the function timeout (`timeout: cdk.Duration.minutes(2)`) to avoid timeouts while processing large images. You should also increase the allocated memory to 1024MB to speed up the computation. To find the best parameters for your Lambda functions, consider tools like AWS Lambda Power Tooling.

The code for metadata and image processing in this example is similar and follows the same pattern:

1. Get input from the state machine via the `event` parameter.
2. Retrieve the file from the source bucket via the AWS SDK.
3. Create a new file by applying respective transformation.

4. Save the new file to the output bucket via AWS SDK.
5. Return the output to the state machine.

```
function ProcessFile(event: any) {
    const file = await S3.getObject({
        Bucket: event.bucketName,
        Key: event.objectKey
    }).promise();

    const transformedFile = // do required transformation

    await S3.putObject({
        Body: transformedFile,
        Bucket: OUTPUT_BUCKET_NAME,
        Key: event.objectKey
    }).promise();

    return {
        AssetId: event.assetId,
        Bucket: OUTPUT_BUCKET_NAME,
        Key: event.objectKey,
        Type: "Image" // respectively "Metadata"
    };
}
```

Video

In contrast to metadata and image, the transcoding of a video file is a long-running task that is not well suited to be handled by a synchronous process for two reasons:

- Lambda functions have a maximum timeout of 15 minutes, and transcoding jobs can (and most of the time will) run longer than that.
- Transcoding video files with AWS Elemental MediaConvert is an asynchronous operation. While the acknowledgement for the retrieval of a transcoding job is synchronous, the results of a transcoding job are communicated via events in Amazon EventBridge.

Amazon EventBridge is a serverless event bus that makes it easy to connect applications together. EventBridge delivers a stream of real-time data from event sources and routes that data to targets like AWS Lambda.

To account for this asynchronicity, configure your state machine to halt until the results of the associated transcoding job are retrieved. Do this by configuring the `ProcessVideo` step in the state machine as a Callback Task.

```
const lambdaHandler = new lambda.Function(this, `..`, {..});

const videoLambdaTask = new tasks.LambdaInvoke(this,
`ProcessProviderAVideo`, {
   lambdaFunction: lambdaHandler,
    payload: sfn.TaskInput.fromObject({
        token: sfn.JsonPath.taskToken,
       // ..
    }),
    integrationPattern: sfn.IntegrationPattern.WAIT_FOR_TASK_TOKEN
});
```

This setting configures this task to be asynchronous:

```
integrationPattern: sfn.IntegrationPattern.WAIT_FOR_TASK_TOKEN
```

Step Functions then generates a unique Task Token and holds the execution. It resumes after it either reaches a defined timeout, or the Task Token is used to submit the results of the asynchronous process back to the waiting task. This is depicted in the architectural diagram in the Architecture overview section of this document.

The Lambda function uses the AWS SDK to trigger a transcoding job in AWS Elemental MediaConvert. MediaConvert enables you to add up to ten different key-value pairs of user data to each transcoding job. This enables you to attach the task token to the transcoding job and later link it back to your state machine.

```
const params = {
   JobTemplate: JOB_TEMPLATE_NAME,
    Queue: MEDIA_CONVERT_QUEUE_ARN,
    Role: MEDIA_CONVERT_ROLE_ARN,
    Settings: {
        // Transcoding settings
    },
    UserMetadata: {
        StepFunctionTaskToken: event.token
    }
};

await MediaConvert
        .createJob(params)
        .promise();
```

Once a MediaConvert job finishes, an EventBridge event is triggered, and you can again react on this event by running code with AWS Lambda. The following snippet defines the required Events Rule, wires it to the Lambda function, and grants permissions that allow it to perform the `states:SendTask*` action on the state machine.

```
const transcodingFinishedLambda = // The Lambda function

new events.Rule(this, 'TranscodingFinished', {
    ruleName: "MediaConvertForSyndicationFinished",
    eventPattern: {
        source: ["aws.mediaconvert"],
        detailType: ["MediaConvert Job State Change"]
    },
    targets: [new
eventtargets.LambdaFunction(transcodingFinishedLambda)]
});

transcodingFinishedLambda.addToRolePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ['states:SendTask*'],
    resources: [stateMachine.stateMachineArn]
}))
```

When a MediaConvert job finishes, `transcodingFinishedLambda` is called with the details of the job. These details contain the user data that was submitted when creating the job. This information can now be used to resume the state machine and appropriately handle success or failure of the transcoding job.

```
const token = event.detail.userMetadata.StepFunctionTaskToken;

if (event.detail.status === "COMPLETE") {
    await StepFunctions.sendTaskSuccess({
        output: // JSON payload for the next step in the
statefunction, i.e. video path
        taskToken: token
    }).promise();
}

if (event.detail.status === "STATUS_UPDATE" || event.detail.status
=== "PROGRESSING") {
    await StepFunctions.sendTaskHeartbeat({
        taskToken: token
    }).promise();
}
```

```
if (event.detail.status === "ERROR" || event.detail.status ===
"CANCELED") {
    await StepFunctions.sendTaskFailure({
        error: event.detail.errorMessage,
        taskToken: token
    }).promise();
}
```

Post-processing

After all three parallel processing steps succeed, you will find three generated output files in the destination bucket. The task after a parallel processing flow retrieves the output of all preceding parallel tasks as its input. This enables you to collect the output from the previous processing steps and proceed with post-processing the data. Typical post-processing steps are encrypting files, packaging a zip bundle, or starting distribution of the files.

In this implementation, the code calculates the checksums of the created files and ends the partner-specific workflow with the following response:

```
{
    Output: {
        Bucket: // output bucket,
        Checksums: // checksums,
        Files: // files
    },
    Provider: "ACE",
    Status: "PROCESS_OK"
};
```

As discussed in the beginning of this document, not every partner has licenses for every content. When the state machine determined that a certain partner is not entitled to receive the content (OtherPartner in this case), the workflow ends at the `Identify relevant partners` task and returns a similar response.

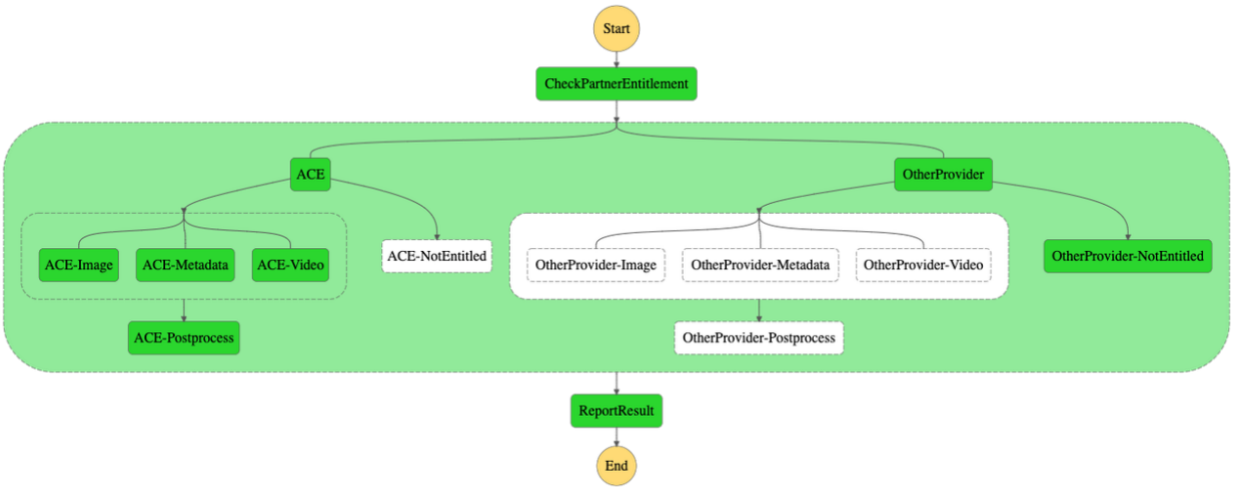
```
{
    Provider: "OtherProvider",
    Status: "IGNORED"
}
```

For this guide, you implemented a simple Lambda function that gathers these provider-specific responses for the complete workflow. Typical use cases at this stage are to keep track of the status of the syndication or notify content operators on changes in the delivery.

Testing the workflow

In this guide, you learned the CDK code required to create the required resources, grant required permissions, and wiring connect it together. After deploying the stack by following the instructions in the GitHub repo, you now have a fully functioning workflow that you can test by uploading files to the source bucket.

The GitHub repository contains a folder called fixtures that contains sample files and a script to upload them to your source bucket. Run source `upload.sh <YOUR_BUCKET_NAME>` to run the script and trigger your state machine. You can now watch in the web console how the state machine runs each task and creates video, image, and metadata files in the process.



Workflow diagram

Source code

The code snippets used in this guide are simplified to illustrate key concepts, but the complete and functional code is available on GitHub. Follow the instructions in the repository to set up a fully functioning syndication workflow.

Conclusion

In this guide, you used AWS Step Functions to define a serverless workflow to syndicate video content. The guide showed a lean solution that is cost-efficient and requires zero upfront investment. It only incurs costs for the actual usage, with no idle resources running, when no content is syndicated.

Horizontally scalability of the solution is virtually unlimited, as more content simply triggers more parallel runs of the state machine.

Adding new content partners to the solution is a non-invasive change to the existing workflows, and only requires adding another path in the provider selection step. Using the CDK enables you to completely automate provisioning of the required AWS resources, making the solution repeatable, maintainable, and resilient to human error.

Contributors

Contributors to this document include:

- Markus Ziller, Senior Solutions Architect, Amazon Web Services

Document revisions

Date	Description
March 11, 2021	First publication