



AWS Compute Blog

Optimizing batch processing with custom checkpoints in AWS Lambda

by [James Beswick](#) | on 15 DEC 2020 | in [Amazon DynamoDB](#), [AWS Lambda](#), [Kinesis Data Streams](#), [Serverless](#) | [Permalink](#) | [Comments](#) | [Share](#)

[AWS Lambda](#) can process batches of messages from sources like [Amazon Kinesis Data Streams](#) or [Amazon DynamoDB Streams](#). In normal operation, the processing function moves from one batch to the next to consume messages from the stream.

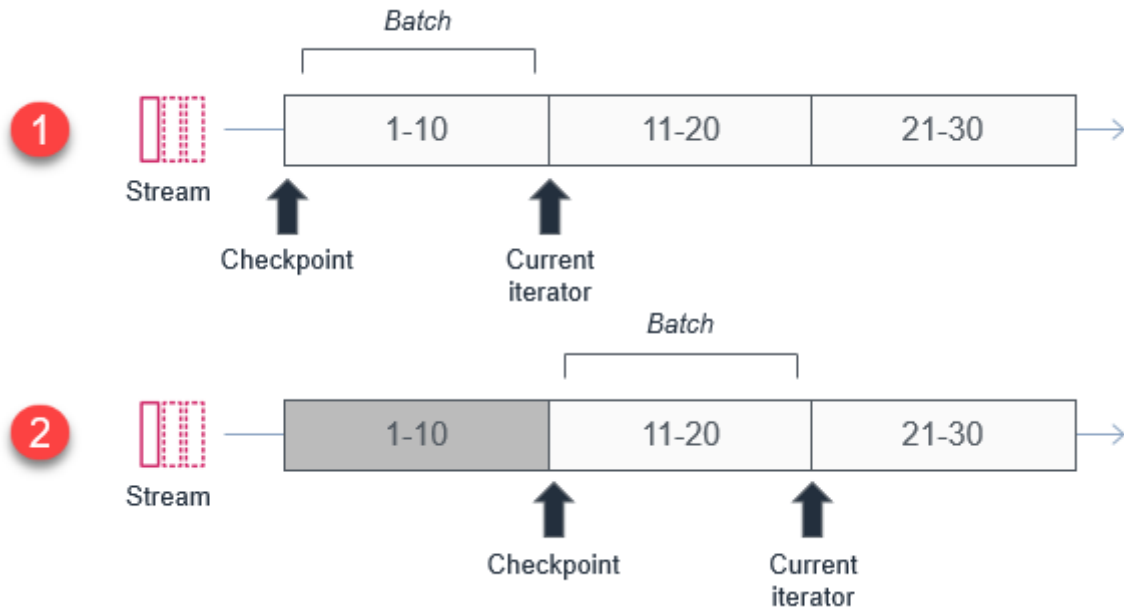
However, when an error occurs in one of the items in the batch, this can result in reprocessing some of the same messages in that batch. With the new custom checkpoint feature, there is now much greater control over how you choose to process batches containing failed messages.

This blog post explains the default behavior of batch failures and options available to developers to handle this error state. I also cover how to use this new checkpoint capability and show the benefits of using this feature in your stream processing functions.

Overview

When using a Lambda function to consume messages from a stream, the [batch size property](#) controls the maximum number of messages passed in each event.

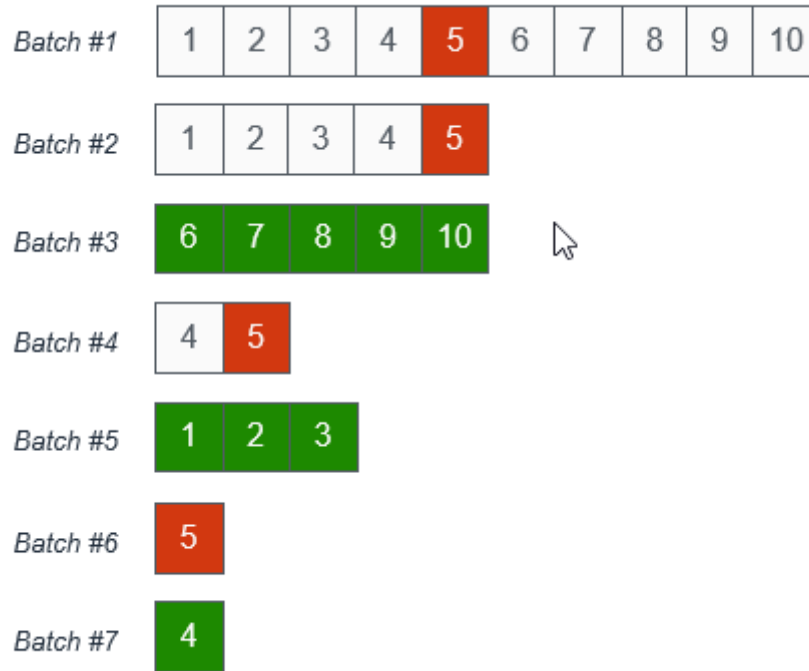
The stream manages two internal pointers: a checkpoint and a current iterator. The checkpoint is the last known item position that was successfully processed. The current iterator is the position in the stream for the next read operation. In a successful operation, here are two batches processed from a stream with a batch size of 10:



1. The first batch delivered to the Lambda function contains items 1–10. The function processes these items without error.
2. The checkpoint moves to item 11. The next batch delivered to the Lambda function contains items 11–20.

In default operation, the processing of the entire batch must succeed or fail. If a single item fails processing and the function returns an error, the batch fails. The entire batch is then retried until the maximum retries is reached. This can result in the same failure occurring multiple times and unnecessary processing of individual messages.

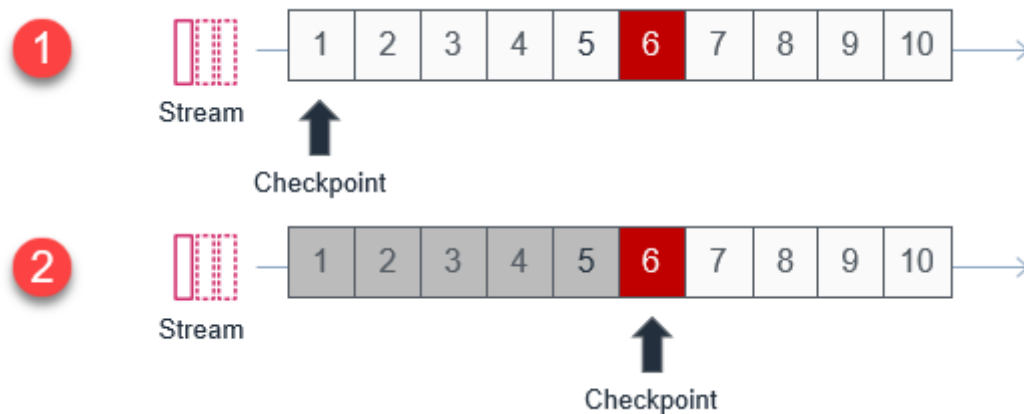
You can also enable the [BisectBatchOnFunctionError](#) property in the event source mapping. If there is a batch failure, the calling service splits the failed batch into two and retries the half-batches separately. The process continues recursively until there is a single item in a batch or messages are processed successfully. For example, in a batch of 10 messages, where item number 5 is failing, the processing occurs as follows:



1. Batch 1 fails. It's split into batches 2 and 3.
2. Batch 2 fails, and batch 3 succeeds. Batch 2 is split into batches 4 and 5.
3. Batch 4 fails and batch 5 succeeds. Batch 4 is split into batches 6 and 7.
4. Batch 6 fails and batch 7 succeeds.

While this provides a way to process messages in a batch with one failing message, it results in multiple invocations of the function. In this example, message number 4 is processed four times before succeeding.

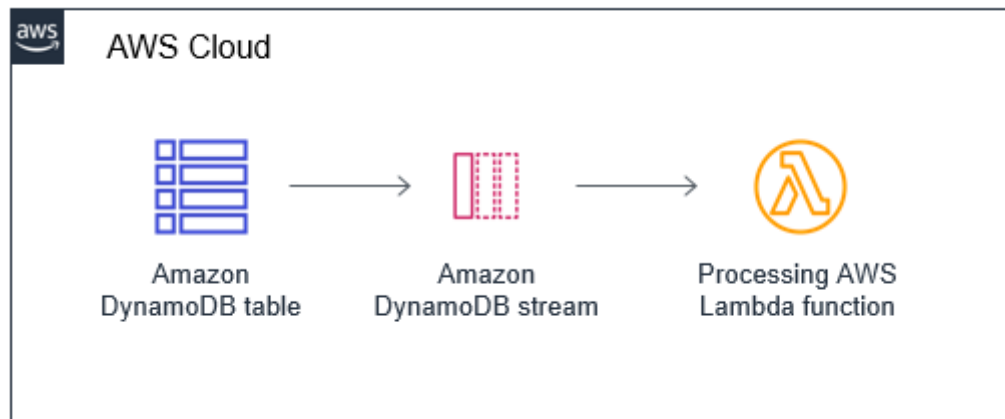
With the new custom checkpoint feature, you can return the sequence identifier for the failed messages. This provides more precise control over how to choose to continue processing the stream. For example, in a batch of 10 messages where the sixth message fails:



1. Lambda processes the batch of messages, items 1–10. The sixth message fails and the function returns the failed sequence identifier.
2. The checkpoint in the stream is moved to the position of the failed message. The batch is retried for only messages 6–10.

Existing stream processing behaviors

In the following examples, I use a DynamoDB table with a Lambda function that is invoked by the stream for the table. You can also use a Kinesis data stream if preferred, as the behavior is the same. The event source mapping is set to a batch size of 10 items so all the stream messages are passed in the event to a single Lambda invocation.



I use the following Node.js script to generate batches of 10 items in the table.

JavaScript

```
const AWS = require('aws-sdk')
AWS.config.update({ region: 'us-east-1' })
const docClient = new AWS.DynamoDB.DocumentClient()

const ddbTable = 'ddbTableName'
const BATCH_SIZE = 10

const createRecords = async () => {
  // Create envelope
  const params = {
    RequestItems: {}
  }
  params.RequestItems[ddbTable] = []

  // Add items to batch and write to DDB
  for (let i = 0; i < BATCH_SIZE; i++) {
    params.RequestItems[ddbTable].push({
      PutRequest: {
        Item: {
```

After running this script, there are 10 items in the DynamoDB table, which are then put into the DynamoDB stream for processing.

Scan: [Table] customCheckpointing: ID ^	
Scan	[Table] customCheckpointing: ID
+ Add filter	
Start search	
<input type="checkbox"/>	ID ⓘ
<input type="checkbox"/>	1607610250077
<input type="checkbox"/>	1607610250078
<input type="checkbox"/>	1607610250079
<input type="checkbox"/>	1607610250080
<input type="checkbox"/>	1607610250081
<input type="checkbox"/>	1607610250082
<input type="checkbox"/>	1607610250083
<input type="checkbox"/>	1607610250084
<input type="checkbox"/>	1607610250085
<input type="checkbox"/>	1607610250086

The processing Lambda function uses the following code. This contains a constant called `FAILED_MESSAGE_NUM` to force an error on the message with the corresponding index in the event batch:

JavaScript

```
exports.handler = async (event) => {
  console.log(JSON.stringify(event, null, 2))
  console.log('Records: ', event.Records.length)
  const FAILED_MESSAGE_NUM = 6

  let recordNum = 1
  let batchItemFailures = []

  event.Records.map((record) => {
    const sequenceNumber = record.dynamodb.SequenceNumber

    if ( recordNum === FAILED_MESSAGE_NUM ) {
      console.log('Error! ', sequenceNumber)
      throw new Error('kaboom')
    }
    console.log('Success: ', sequenceNumber)
    recordNum++
  })
}
```

```

    })
  }
}

```

The code uses the DynamoDB item's sequence number, which is provided in each record of the stream event:

```

{
  "Records": [
    {
      "eventID": "01ee18c571f7bf142376f84fee288f0d",
      "eventName": "INSERT",
      "eventVersion": "1.1",
      "eventSource": "aws:dynamodb",
      "awsRegion": "us-east-1",
      "dynamodb": {
        "ApproximateCreationDateTime": 1607610475,
        "Keys": {
          "ID": {
            "N": "1607610475212"
          }
        },
        "NewImage": {
          "ID": {
            "N": "1607610475212"
          }
        },
        "SequenceNumber": "34310000000006400636574",
        "SizeBytes": 20,
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN": "arn:aws:dynamodb:us-east-1:79e3fe26-69f3-4b4c-a8fe-979b44bc00de:table/customCheckpointing/stream/2020-12-09T18:21:16.022"
    }
  ]
}

```

In the default configuration of the event source mapping, the failure of message 6 causes the whole batch to fail. The entire batch is then retried multiple times. This appears in the [CloudWatch Logs](#) for the function:

20:50:17	START RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de Version: \$LATEST
20:50:17	2020-12-09T20:50:17.549Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO {"Records": [{"eventID": "7838c821f0cf9c00bd0718cea7bbef7e"}]}
20:50:17	2020-12-09T20:50:17.549Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Records: 9
20:50:17	2020-12-09T20:50:17.549Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18100000000026768641749
20:50:17	2020-12-09T20:50:17.549Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18200000000026768641793
20:50:17	2020-12-09T20:50:17.550Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18300000000026768641794
20:50:17	2020-12-09T20:50:17.550Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18400000000026768641795
20:50:17	2020-12-09T20:50:17.550Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18500000000026768641796
20:50:17	2020-12-09T20:50:17.550Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Error! 18600000000026768641797
20:50:17	2020-12-09T20:50:17.559Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de ERROR Invoke Error ("errorType": "Error", "errorMessage": "kaboom", "stack": "Error: kaboom\n at /var/task/index.js:1:1\n at processTicksAndRejections (node:internal/process/task_queues:93:5)"})
20:50:17	END RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de
20:50:17	REPORT RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de Duration: 15.75 ms Billed Duration: 16 ms Memory Size: 128 MB Max Memory Used: 128 MB
20:50:17	START RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de Version: \$LATEST
20:50:17	2020-12-09T20:50:17.780Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO {"Records": [{"eventID": "7838c821f0cf9c00bd0718cea7bbef7e"}]}
20:50:17	2020-12-09T20:50:17.780Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Records: 9
20:50:17	2020-12-09T20:50:17.780Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18100000000026768641749
20:50:17	2020-12-09T20:50:17.780Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18200000000026768641793
20:50:17	2020-12-09T20:50:17.780Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18300000000026768641794
20:50:17	2020-12-09T20:50:17.781Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18400000000026768641795
20:50:17	2020-12-09T20:50:17.781Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Success: 18500000000026768641796
20:50:17	2020-12-09T20:50:17.781Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de INFO Error! 18600000000026768641797
20:50:17	2020-12-09T20:50:17.781Z 79e3fe26-69f3-4b4c-a8fe-979b44bc00de ERROR Invoke Error ("errorType": "Error", "errorMessage": "kaboom", "stack": "Error: kaboom\n at /var/task/index.js:1:1\n at processTicksAndRejections (node:internal/process/task_queues:93:5)"})
20:50:17	END RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de
20:50:17	REPORT RequestId: 79e3fe26-69f3-4b4c-a8fe-979b44bc00de Duration: 2.23 ms Billed Duration: 3 ms Memory Size: 128 MB Max Memory Used: 128 MB

Next, I enable the [bisect-on-error](#) feature in the function's event trigger. The first invocation fails as before but this causes two subsequent invocations with batches of five messages. The original batch is bisected. These batches complete processing successfully.

START RequestId: 8697e8f4-f83d-46f7-b10b-41fad5f697f7 Version: \$LATEST

2020-12-10T20:07:50.329Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO {"Records": [{ "eventID": "c095fb7acb45fb6137c84da5

2020-12-10T20:07:50.349Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Records: 10

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Success: 5108100000000004160121415

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Success: 5108200000000004160121416

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Success: 5108300000000004160121417

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Success: 5108400000000004160121418

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Success: 5108500000000004160121419

2020-12-10T20:07:50.350Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 INFO Error! 5108600000000004160121420

2020-12-10T20:07:50.351Z 8697e8f4-f83d-46f7-b10b-41fad5f697f7 ERROR Invoke Error {"errorType":"Error","errorMessage":"ka

END RequestId: 8697e8f4-f83d-46f7-b10b-41fad5f697f7

REPORT RequestId: 8697e8f4-f83d-46f7-b10b-41fad5f697f7 Duration: 52.17 ms Billed Duration: 53 ms Memory Size: 128 MB M

START RequestId: cb934eb4-4ba6-4543-86cb-db489f79b951 Version: \$LATEST

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO {"Records": [{ "eventID": "c095fb7acb45fb6137c84c

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Records: 5

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Success: 5108100000000004160121415

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Success: 5108200000000004160121416

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Success: 5108300000000004160121417

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Success: 5108400000000004160121418

2020-12-10T20:07:50.399Z cb934eb4-4ba6-4543-86cb-db489f79b951 INFO Success: 5108500000000004160121419

END RequestId: cb934eb4-4ba6-4543-86cb-db489f79b951

REPORT RequestId: cb934eb4-4ba6-4543-86cb-db489f79b951 Duration: 2.26 ms Billed Duration: 3 ms Memory Size: 128 MB M

START RequestId: e892bdbb-930e-460d-921d-4259d8a4877c Version: \$LATEST

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO {"Records": [{ "eventID": "bc7143a73b195475ebb5

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Records: 5

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Success: 5108600000000004160121420

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Success: 5108700000000004160121421

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Success: 5108800000000004160121422

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Success: 5108900000000004160121423

2020-12-10T20:07:50.446Z e892bdbb-930e-460d-921d-4259d8a4877c INFO Success: 5109000000000004160121424

END RequestId: e892bdbb-930e-460d-921d-4259d8a4877c

REPORT RequestId: e892bdbb-930e-460d-921d-4259d8a4877c Duration: 1.55 ms Billed Duration: 2 ms Memory Size: 128 MB M

Configuring a custom checkpoint

Finally, I enable the custom checkpoint feature. This is configured in the Lambda function console by selecting the “Report batch item failures” check box in the DynamoDB trigger:

Add trigger

Trigger configuration

DynamoDB
aws database nosql

DynamoDB table
Choose or enter the ARN of a DynamoDB table.

arn:aws:dynamodb:us-east-1:780018668030:table/customCheckpointing

Report batch item failures
Allow your function to return a partial successful response for a batch of records.
☒

In order to read from the DynamoDB trigger, your execution role must have proper permissions.

☒ **Enable trigger**
Enable the trigger now, or create it in a disabled state for testing (recommended).

Cancel Add

I update the processing Lambda function with the following code:

JavaScript

```
exports.handler = async (event) => {
  console.log(JSON.stringify(event, null, 2))
  console.log('Records: ', event.Records.length)
  const FAILED_MESSAGE_NUM = 4

  let recordNum = 1
  let sequenceNumber = 0

  try {
    event.Records.map((record) => {
      sequenceNumber = record.dynamodb.SequenceNumber

      if ( recordNum === FAILED_MESSAGE_NUM ) {
        throw new Error('kaboom')
      }
      console.log('Success: ', sequenceNumber)
      recordNum++
    })
  }
```



```
} catch (err) {
```

In this version of the code, the processing of each message is wrapped in a `try...catch` block. When processing fails, the function stops processing any remaining messages. It returns the sequence number of the failed message in a JSON object:

JSON

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": sequenceNumber
    }
  ]
}
```

The calling service then updates the checkpoint value with the sequence number provided. If the *batchItemFailures* array is empty, the caller assumes all messages have been processed correctly. If the *batchItemFailures* array contains multiple items, the lowest sequence number is used as the checkpoint.

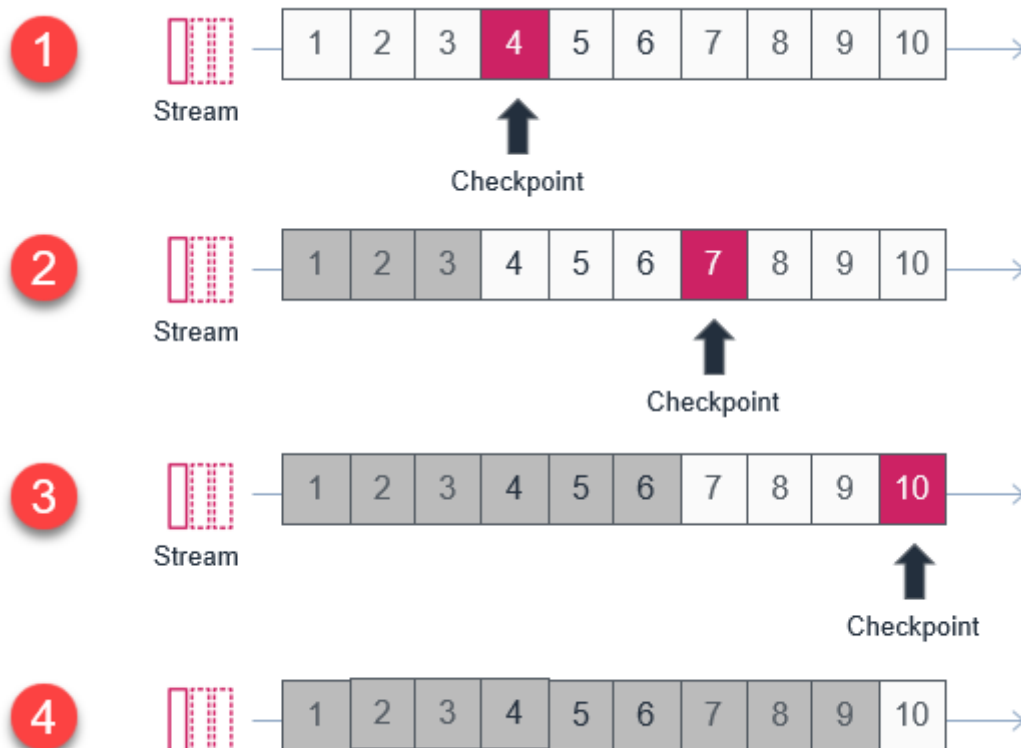
In this example, I also modify the `FAILED_MESSAGE_NUM` constant to 4 in the Lambda function. This causes the fourth message in every batch to throw an error. After adding 10 items to the DynamoDB table, the CloudWatch log for the processing function shows:

```

START RequestId: cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 Version: $LATEST
2020-12-10T19:20:34.030Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO {"Records": [{"eventID": "d8a5dbc789d8c520d4937d6a259ad93b", "eventNam
2020-12-10T19:20:34.044Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO Records: 10
2020-12-10T19:20:34.044Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO Success: 4313000000000000728645557
2020-12-10T19:20:34.044Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO Success: 43131000000000000728645558
2020-12-10T19:20:34.063Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO Success: 43132000000000000728645559
2020-12-10T19:20:34.063Z cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 INFO Failure: 43133000000000000728645560
END RequestId: cac547f5-ec20-42e4-a7c7-c9b87d38e1b5
REPORT RequestId: cac547f5-ec20-42e4-a7c7-c9b87d38e1b5 Duration: 36.02 ms Billed Duration: 37 ms Memory Size: 128 MB Max Memory Used: 65 MB
START RequestId: 0550aa2d-e36e-4b37-8f6d-685dd6915668 Version: $LATEST
2020-12-10T19:20:34.083Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO {"Records": [{"eventID": "37a21210caadd02cdfcfcc35358601", "eventNam
2020-12-10T19:20:34.083Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO Records: 7
2020-12-10T19:20:34.084Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO Success: 43133000000000000728645560
2020-12-10T19:20:34.084Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO Success: 43134000000000000728645561
2020-12-10T19:20:34.084Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO Success: 43135000000000000728645562
2020-12-10T19:20:34.084Z 0550aa2d-e36e-4b37-8f6d-685dd6915668 INFO Failure: 43136000000000000728645563
END RequestId: 0550aa2d-e36e-4b37-8f6d-685dd6915668
REPORT RequestId: 0550aa2d-e36e-4b37-8f6d-685dd6915668 Duration: 5.16 ms Billed Duration: 6 ms Memory Size: 128 MB Max Memory Used: 66 MB
START RequestId: 60bb324b-64ee-43a0-b4a9-40196716ccc5 Version: $LATEST
2020-12-10T19:20:34.126Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO {"Records": [{"eventID": "ba75bc66452dc45c5602504ec53381ca", "eventNam
2020-12-10T19:20:34.126Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO Records: 4
2020-12-10T19:20:34.127Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO Success: 43136000000000000728645563
2020-12-10T19:20:34.127Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO Success: 43137000000000000728645564
2020-12-10T19:20:34.127Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO Success: 43138000000000000728645565
2020-12-10T19:20:34.127Z 60bb324b-64ee-43a0-b4a9-40196716ccc5 INFO Failure: 43139000000000000728645566
END RequestId: 60bb324b-64ee-43a0-b4a9-40196716ccc5
REPORT RequestId: 60bb324b-64ee-43a0-b4a9-40196716ccc5 Duration: 1.82 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 66 MB
START RequestId: 43e67198-9bd6-4556-9838-1566be83e8cd Version: $LATEST
2020-12-10T19:20:34.182Z 43e67198-9bd6-4556-9838-1566be83e8cd INFO {"Records": [{"eventID": "d6bbe4582825f903f9b5162030ccc673", "eventNam
2020-12-10T19:20:34.182Z 43e67198-9bd6-4556-9838-1566be83e8cd INFO Records: 1
2020-12-10T19:20:34.182Z 43e67198-9bd6-4556-9838-1566be83e8cd INFO Success: 43139000000000000728645566
END RequestId: 43e67198-9bd6-4556-9838-1566be83e8cd
REPORT RequestId: 43e67198-9bd6-4556-9838-1566be83e8cd Duration: 1.84 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 66 MB

```

This is how the stream of 10 messages has been processed using the custom checkpoint:



1. In the first invocation, all 10 messages are in the batch. The fourth message throws an error. The function returns this position as the checkpoint.
2. In the second invocation, messages 4–10 are in the batch. Message 7 throws an error and its sequence number is returned as the checkpoint.
3. In the third invocation, the batch contains messages 7–10. Message 10 throws an error and its sequence number is now the returned checkpoint.
4. The final invocation contains only message 10, which is successfully processed.

Using this approach, subsequent invocations do not receive messages that have been successfully processed previously.

Conclusion

The default behavior for stream processing in Lambda functions enables entire batches of messages to succeed or fail. You can also use batch bisecting functionality to retry batches iteratively if a single message fails. Now with custom checkpoints, you have more control over handling failed messages.

This post explains the three different processing modes and shows example code for handling failed messages. Depending upon your use-case, you can choose the appropriate mode for your workload. This can help reduce unnecessary Lambda invocations and prevent reprocessing of the same messages in batches containing failures.

To learn more about how to use this feature, read the developer documentation for [DynamoDB](#) and [Kinesis Streams](#). To learn more about building with serverless technology, visit [Serverless Land](#).

TAGS: [serverless](#)



AWS Podcast

Subscribe for weekly AWS news and interviews

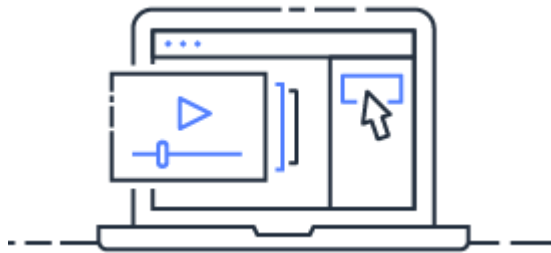
[Learn more »](#)



AWS Partner Network

Find an APN member to support your cloud business needs

[Learn more »](#)



AWS Training & Certifications

Free digital courses to help you develop your skills

[Learn more »](#)

Resources

[Serverless Computing and Applications](#)

[Amazon Container Services](#)

[AWS Messaging](#)




[Cloud Compute with AWS](#)

[Desktop and Application Streaming](#)

Follow

 [Twitter](#)

 [Facebook](#)

-  [LinkedIn](#)
-  [Twitch](#)
-  [Email Updates](#)



New Launches From re:Invent

Discover the latest services and features from AWS

[Visit the News Blog »](#)

Related Posts

[Using AWS Lambda for streaming analytics](#)

[Detecting sensitive data in DynamoDB with Macie](#)

[BandLab welcomes users by the millions with AWS](#)

[ICYMI: Serverless pre:Invent 2020](#)

[Detect change points in your event data stream using Amazon Kinesis Data Streams, Amazon DynamoDB and AWS Lambda](#)

[Integrating Amazon ElastiCache with other AWS services: The serverless way](#)

[New – Export Amazon DynamoDB Table Data to Your Data Lake in Amazon S3, No Code Writing Required](#)

[How to deliver headless commerce in retail](#)