November 11th 2024

Thomas Feduk, Jr.
thomasfeduk@gmail.com

# Best Practices Whitepaper: AWS Kinesis & Lambda

- Implementing eventual consistency and disaster recovery methodologies in event-driven distributed systems with Kinesis and Lambdas.
- Ensuring exactly-once-processing with or without idempotency with reliable error recovery, re-try handling and DLQs.

## Table of Contents

**Goal of this paper**

Throughout my career, I've observed that many organizations using event-driven, distributed microservices on AWS lack a standardized approach to implementing the various and subtle details for their infrastructure integrations. A very frequently used, but inconsistently managed integration that I see is Kinesis streams with Lambda consumers. The frequent absence of formal standards has led to varied practices in integration methods, error handling, recovery, and infrastructure management—not only between teams but often even within projects on the same team. As such, the disaster recovery, retry, and failure handling must often be addressed for each project on a case-by-case basis, often with recovery processes developed only after a failure occurs. In some cases, recovery requires extensive manual intervention, such as investigating and scrapping from raw logs, resetting database values directly, or writing scripts to handle rebroadcasting missed events. In others, there may be partial automation, but the extent of this automation is often known only to the implementing team.

The core goal of this paper is to propose such a standard for specifically the AWS Kinesis+Lambda consumer model dictating best practices and
recommendations that projects can adopt organization wide regarding that address the following:

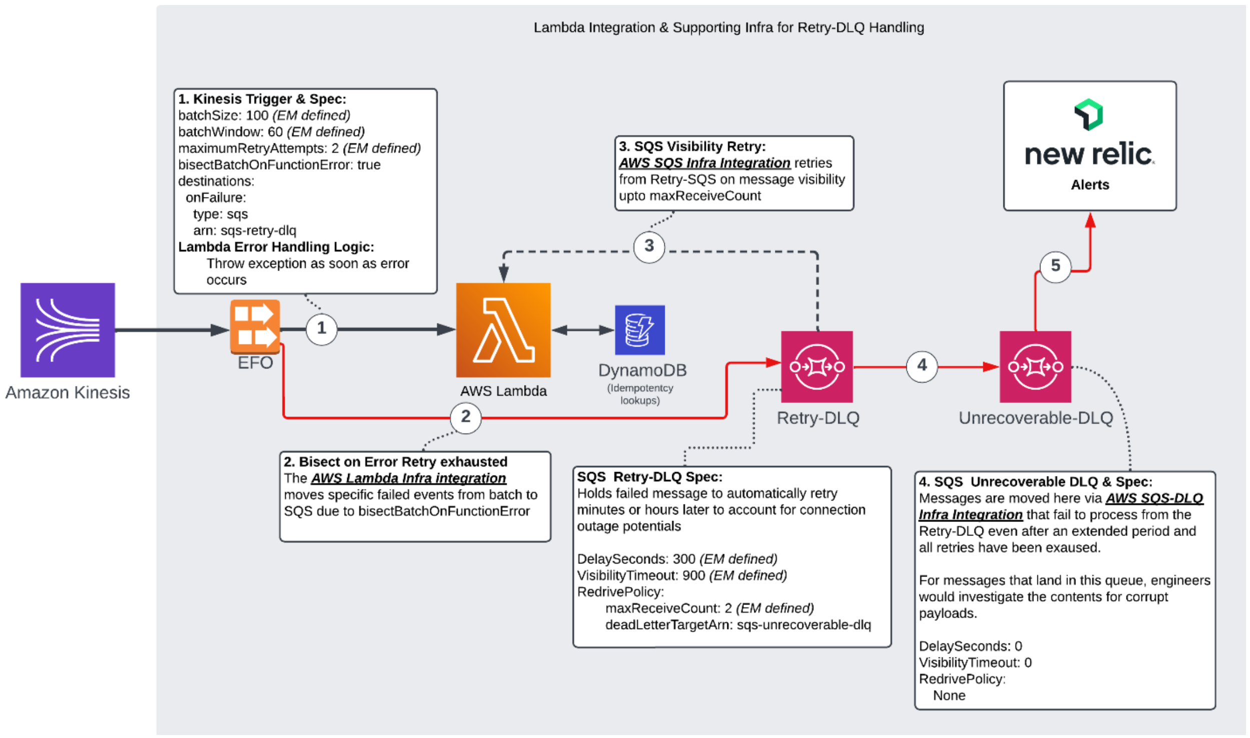
1. Kinesis Lambda integration and trigger specifications
2. Impotency handling
    a. Infrastructure controlled automatic retry and DLQ handling resulting in eventual consistency if idempotency is supported
    b. A combination of infrastructure and app-logic based retry and DLQ handling to achieve automated eventual consistency when idempotency is not supported.

## Proposed Solution

This paper outlines two different solutions to ensure exactly-once-processing of events with automated re-tries, error handling and DLQ support depending on the availability of idempotency in the subscribed stream:

1. Streams supporting idempotent-safe events
2. Streams that do not support idempotent-safe events

## 1. Idempotent Supported Streams



**1. Kinesis Trigger & Spec:**
batchSize: 100 *(EM defined)*
batchWindow: 60 *(EM defined)*
maximumRetryAttempts: 2 *(EM defined)*
bisectBatchOnFunctionError: true
destinations:
  onFailure:
    type: sqs
    arn: sqs-retry-dlq
**Lambda Error Handling Logic:**
Throw exception as soon as error occurs

**2. Bisect on Error Retry exhausted**
The ***AWS Lambda Infra integration*** moves specific failed events from batch to SQS due to bisectBatchOnFunctionError

**3. SQS Visibility Retry:**
***AWS SQS Infra Integration*** retries from Retry-SQS on message visibility upto maxReceiveCount

**SQS Retry-DLQ Spec:**
Holds failed message to automatically retry minutes or hours later to account for connection outage potentials

DelaySeconds: 300 *(EM defined)*
VisibilityTimeout: 900 *(EM defined)*
RedrivePolicy:
  maxReceiveCount: 2 *(EM defined)*
  deadLetterTargetArn: sqs-unrecoverable-dlq

**4. SQS Unrecoverable DLQ & Spec:**
Messages are moved here via ***AWS SQS-DLQ Infra Integration*** that fail to process from the Retry-DLQ even after an extended period and all retries have been exaused.

For messages that land in this queue, engineers would investigate the contents for corrupt payloads.

DelaySeconds: 0
VisibilityTimeout: 0
RedrivePolicy:
  None

## Integration Detail Overview

This approach utilizes pure infrastructure to handle all retires, retry delays, DLQ interaction and alerts. There is no app programming logic needed besides simply throwing an exception when an error occurs.

However, this approach only works if the messages are idempotent (by either an idempotency key, unique id, or as described in a  separate section below: it is possible to use the sequence+subsequence number combination as an idempotency key to facilitate the workflow described in this section).

## Integration Components

### Lambda

The Lambda Kinesis integration triggered via an EFO consumer (EFO if over 2 total consumers per stream) with **bisectBatchOnFunctionError=True**. This allows for all the infrastructure to handle retries, error handling and DLQ processing. **batchSize**, **maximumRetryAttempts** and **batchWindow** are set per the Engineer Manager's project needs and are only given recommended values in this paper that will likely be appropriate as the starting values for run-of-the-mill projects.

In addition to the Kinesis integration trigger, the Lambda should also be subscribed to the **Retry-DLQ** via a **visibility trigger** to handle Engineer Manager defined post-failure-retries. Thus the Lambda logic must be written to support both the **Kinesis-Lambda event object schema** and **SQS-Kinesis-DLQ event object schema**.

As each message in the batch is processed **successfully**:

a. **The idempotency key** should be written to persistent storage with an Engineering Manager's desired **TTL** for the project.
   - This prevents re-processing of the message if another message in the batch has a failure due to the **bisectBatchOnFunctionError** handling.

b. The message's **sequence number** should be logged when failure occurs.

        - In the event of a severe disaster such as the re-try handling or DLQ itself failing, this allows for tools such as the **Kinesis-SLR** to scrape and re-play exactly the missed messages in any sequence number or timestamp range.

When **errors do occur** when a message is being processed, (invalid message payload, connectivity issues to another service, internal handling fault, etc.):

a. The function code should **log** the error details including messages sequence number.
b. Return an **exception** back to the infrastructure.
c. This will result in the bisect error handling splitting the batch and re-trying until all other messages are processed successfully or the resulting failed message(s) will be automatically sent to the SQS-Retry DLQ.

## DynamoDB

A DynamoDB (or any storage mechanism such as RDS, Redis etc.) is used to store the idempotency keys of successfully processed messages in the event a batch is re-tried due to **bisectBatchOnFunctionError** and if failures occur processing other messages. To save on storage/bloat, a **TTL** is recommended to be used on the idempotency keys that is Engineering Manager defined based on the project's allowable recovery window in the event re-tries do occur. Some re-tries may be allowable only within a maximum of a 1 hour window, while others may be allowable a day or more later.

## Retry-DLQ

This is the "active worker" DLQ that receives individual messages that have been bisected out of the batch. It is recommended the **same Lambda** that is subscribed to the Kinesis stream is also subscribed to this SQS via a visibility trigger. Messages that are delivered into this queue from the AWS Kinesis infrastructure messages are automatically re-played back to the Lambda, delayed based on the **DelaySeconds**, **VisibilityTimeout** and **maxReceiveCount** options configured for the DLQ. These values are Engineering Manager defined based on the project's allowable recovery window. Some projects may want to wait 10 minutes before the first try, then an hour for upto 2 subsequent reties.

If the **maxReceiveCount** value is reached, then all re-tries have been exhausted and the message is then delivered to the **Unrecoverable-DLQ** as dictated via the **RedrivePolicy** where it will wait until an engineer can investigate and manually re-play the message or purge it based on the findings and situation.

## Unrecoverable-DLQ

This is an "inactive" queue that simply receives and stores messages that have exhausted all re-try attempts and deemed to be unrecoverable via the automatic infrastructure handling process. New Relic Alarms should be attached to this queue as any messages that land here will by design **not** be automatically processed. Once this queue becomes populated, it will fall to an engineer to determine the cause for failure, and if appropriate, manually move the messages back to the Retry-DLQ to re-play them once the problem has been resolved, or purge the messages as needed.

## Flows

*(Step numbers correspond to chart numbers above)*

### Successfully Processed Event

1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
    a. Log the sequence number for the current message about to be processed
    b. Check the DynamoDB to determine if message was already processed, if so, log the identification and abort
    c. Message is processed successfully
    d. Log the message's sequence number (and if aggregation is used, sub-sequence number)
    e. Store the message's idempotency key in the DynamoDB

### Recoverable Event Failure

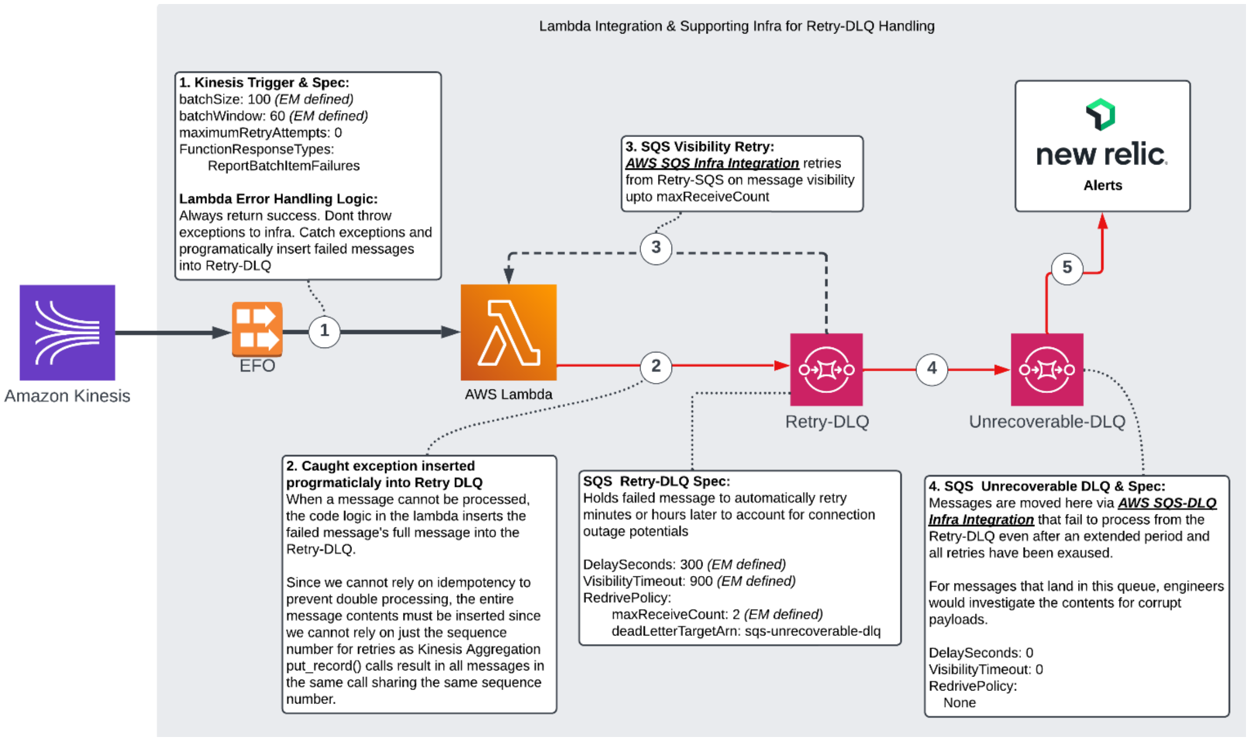
1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
    a. Log the sequence number for the current message about to be processed
    b. Check the DynamoDB to determine if message was already processed, if so, log the identification and abort

   c. Message begins to process but encounters an error. Throw an exception detailing the error that occurred, ensuring to include the sequence/subsequence number.
2. Kinesis Integration moves the failed message to the Retry-DLQ
   a. Once the Kinesis Integration sees the exception the exception thrown from the lambda, it beings to bisect the batch in half until the single failed message is identified.
   b. Once the failed message is identified, the infrastructure will automatically write the message's meta data (sequence number, timestamp etc) into an SQS message inserted into the Retry-DLQ.
3. The Retry-DLQ will retry the message
   a. The Retry-DLQ will wait **DelaySeconds**, then make visible the newly inserted message
   b. The Lambda subscribed to the Retry-DLQ visibility trigger will automatically be invoked and sent the message for processing
   c. If the Lambda successfully process the message:
      i. Jump to step **1.a** of **"Successfully Processed Event"** above.
      ii. The Retry-DLQ will purge the message due to no error return by the lambda
   d. If the message fails to be processed:
      i. The Lambda will throw an exception detailing the error that occurred, ensuring to include the sequence/subsequence number.
      ii. The message will remain in-flight in the SQS-DLQ due to the exception being detected and repeat the Retry-DLQ Lambda process after the **VisibilityTimeout** duration, up to **maxReceiveCount** times.

## Unrecoverable Event Failure

1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
   a. *(Same steps as Recoverable Event Failure step 1)*
2. Kinesis Integration moves the failed message to the Retry-DLQ
   a. *(Same steps as Recoverable Event Failure step 2)*
3. The Retry-DLQ will retry the message
   a. *(Same steps as Recoverable Event Failure step 3)*
4. Once all re-tries have been exhausted, move the failed message to the Unrecoverable-DLQ
   a. Once the **maxReceiveCount** has been reached on the message on the Retry-DLQ, the message is then automatically moved into the Unrecoverable-DLQ
   b. New Relic fires an alert as it is subscribed to the activity in this queue
   c. At this point it will fall to an engineer to determine the cause for failure and what manual handling should be done.

## 2. Idempotent Unsupported Streams



**1. Kinesis Trigger & Spec:**
batchSize: 100 *(EM defined)*
batchWindow: 60 *(EM defined)*
maximumRetryAttempts: 0
FunctionResponseTypes:
ReportBatchItemFailures

**Lambda Error Handling Logic:**
Always return success. Dont throw exceptions to infra. Catch exceptions and programatically insert failed messages into Retry-DLQ

**3. SQS Visibility Retry:**
***AWS SQS Infra Integration*** retries from Retry-SQS on message visibility upto maxReceiveCount

**2. Caught exception inserted progrmaticlaly into Retry DLQ**
When a message cannot be processed, the code logic in the lambda inserts the failed message's full message into the Retry-DLQ.

Since we cannot rely on idempotency to prevent double processing, the entire message contents must be inserted since we cannot rely on just the sequence number for retries as Kinesis Aggregation put_record() calls result in all messages in the same call sharing the same sequence number.

**SQS Retry-DLQ Spec:**
Holds failed message to automatically retry minutes or hours later to account for connection outage potentials

DelaySeconds: 300 *(EM defined)*
VisibilityTimeout: 900 *(EM defined)*
RedrivePolicy:
maxReceiveCount: 2 *(EM defined)*
deadLetterTargetArn: sqs-unrecoverable-dlq

**4. SQS Unrecoverable DLQ & Spec:**
Messages are moved here via ***AWS SQS-DLQ Infra Integration*** that fail to process from the Retry-DLQ even after an extended period and all retries have been exaused.

For messages that land in this queue, engineers would investigate the contents for corrupt payloads.

DelaySeconds: 0
VisibilityTimeout: 0
RedrivePolicy:
None

### Integration Detail Overview

This solution can be used for any type of message and stream regardless of idempotency as it uses a mix of application logic for DLQ insertions and SQS infrastructure handling for all retires, retry delays, and alerts.

### Integration Components

### Lambda

The Lambda Kinesis integration triggered via an EFO consumer (EFO if over 2 total consumers per stream) with the configuration:

- **bisectBatchOnFunctionError=False**
- **MaximumRetryAttempts=0**
- **ReportBatchItemFailures=True**

The above values ensure **exactly-once-processing** as disable any infrastructure based re-try mechanisms at the batch level which cannot be used due to the lack of idempotent safe messages (**bisectBatchOnFunctionError** and **MaximumRetryAttempts**). We make use of **ReportBatchItemFailures** to allow for application logic to be written into the Lambda **to handle Retry-DLQ insertion** and cursor advancement in the stream. This is accomplished by reporting all messages have succeeded to the Kinesis integration regardless of message failures as failures are addressed via application logic in the Lambda exclusively.

**batchSize**, and **batchWindow** are set per the Engineer Manager's project needs and are only given recommended values in this paper that will likely be appropriate as the starting values for run-of-the-mill projects.

In addition to the Kinesis integration trigger, the Lambda should also be subscribed to the **Retry-DLQ** via a visibility trigger to handle Engineer Manager defined post-failure-retries. Thus the Lambda logic must be written to support both the Kinesis-**Lambda event object schema** and **custom-SQS-DLQ event object schema**.

## Retry-DLQ

This is the "active worker" DLQ that receives individual failed messages that have been inserted It is recommended the **same Lambda** that is subscribed to the Kinesis stream is also subscribed to this SQS via a visibility trigger. Messages that are delivered into this queue from the Lambda application failure handling are automatically re-played back to the Lambda, delayed based on the **DelaySeconds**, **VisibilityTimeout** and **maxReceiveCount** options configured for this DLQ These values are Engineering Manager defined based on the project's allowable recovery window. Some projects may want to wait 10 minutes before the first try, then an hour for up to 2 subsequent reties.

If the **maxReceiveCount** value is reached, then all re-tries have been exhausted and the message is then delivered to the **Unrecoverable DLQ** as dictated via the **RedrivePolicy** where it will wait until an engineer can investigate and manually re-play the message or purge it based on the findings and situation.

## Unrecoverable-DLQ

This is an "inactive" queue that simply receives and stores messages that have exhausted all re-try attempts and deemed to be unrecoverable via the automatic infrastructure handling process. New Relic Alarms should be attached to this queue as any messages that land here will by design not be automatically processed. Once this queue becomes populated, it will fall to an engineer to determine the cause for failure, and if appropriate, manually move the messages back to the Retry-DLQ to re-play them once the problem has been resolved, or purge the messages as needed.

## Flows

*(Step numbers correspond to chart numbers above)*

### Successfully Processed Event

1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
    a. Log the sequence number for the current message about to be processed
    b. Message is processed successfully
    c. Log the message's sequence number (and if aggregation is used, sub-sequence number)
    d. Once all messages in the batch have been processed, return an empty **batchItemFailures** empty list to communicate to the infrastructure that all messages were processed successfully

### Recoverable Event Failure

1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
    a. Log the sequence number for the current message about to be processed
    b. Message begins to process but encounters an error. Throw an exception within the iteration loop the error detailing what occurred, ensuring to include the sequence/subsequence number.
2. At the batch iteration handler level, capture the thrown exception and log the exception details.
    a. Application logic inserts the failed message into the message into the Retry-DLQ then continue onto the next message.
    b. Once all messages in the batch have been processed, return an empty **batchItemFailures** empty list to communicate to the infrastructure that all messages were processed successfully
3. The Retry-DLQ will retry the message
    a. The Retry-DLQ will wait **DelaySeconds**, then make visible the newly inserted message
    b. The Lambda subscribed to the Retry-DLQ visibility trigger will automatically be invoked and sent the message for processing
    c. If the Lambda successfully process the message:
        i. Jump to step **1.a** of **"Successfully Processed Event"** above.
        ii. The Retry-DLQ will purge the message due to no error return by the lambda
    d. If the message fails to be processed:
        i. The Lambda will throw an exception to the SQS trigger detailing the error that occurred, ensuring to include the sequence/subsequence number.
        ii. The message will remain in-flight in the SQS-DLQ due to the exception being detected and repeat the **Retry-DLQLambda** process after the **VisibilityTimeout** duration, up to **maxReceiveCount** times

Unrecoverable Failure

1. Lambda is invoked via the Kinesis trigger and begins iterating through each message
    a. *(Same steps as Recoverable Event Failure step 1)*
2. Lambda application logic moves the failed message to the Retry-DLQ
    a. *(Same steps as Recoverable Event Failure step 2)*
3. The Retry-DLQ will retry the message
    a. *(Same steps as Recoverable Event Failure step 3)*
4. Once all re-tries have been exhausted, move the failed message to the Unrecoverable-DLQ
    a. Once the **maxReceiveCount** has been reached on the message on the Retry-DLQ, the message is then automatically moved into the Unrecoverable-DLQ
    b. New Relic fires an alert as it is subscribed to the activity in this queue
    c. At this point it will fall to an engineer to determine the cause for failure and what manual handling should be done.

## Alternative Considerations for Idempotent Unsupported streams

## Sequence Number as Idempotency key

It is important to note, that as an alternative to the section above in utilizing a mixture of application logic to handle the Retry-DLQ insertions to ensure exactly-once-processing, it is possible to use the **sequence+subsequence** number combination as an idempotency key to treat **any** Kinesis stream as **idempotent supported**. However, this comes with the caveat that utilizing this system will **necessarily and intrinsically tie** the AWS Kinesis infrastructure to the idempotency key and that reliance will be lost for messages which have expired due to the stream's data retention period configuration.

## Custom Checkpoints

**Custom checkpoints** have already been considered and disregarded as a viable alternative given it cannot be handled via purely infrastructure. They require all handling logic and DLQ insertion to be implemented in the application code layer and thus provide no benefit over the "non-idempotent safe" handling process already proposed in this document. It was identified that there is simply no mechanism after research and extensive testing to allow for the infrastructure itself to auto-dlq the failed message and advance the cursor past the failed message in the stream.

## Known Risks/Drawbacks for consideration aligned against counterpoint benefits

**Idempotent Supported Streams**

- **Benefits** of the **idempotent supported** recommendation: This methodology is considered industry standard in many production environments as **all error tracking, handling, retires and DLQ insertions** are addressed **purely by the infrastructure**. Absolutely zero engineer time, overhead or logic needs to be incorporated at the application source level. This saves substantial engineering costs, training overhead, simplifies alerts as they can target the unrecoverable-dlq directly and removes user error from retry-dlq logic implementation.

- **Drawbacks** to the **idempotent supported** recommendation: **Increased AWS operating costs** and **concurrency quotas** are the largest concerns. Due to utilize of the Kinesis integration option **bisectBatchOnFunctionError**, even with retries set to **0**, at the stream integration level, a single bad message in a batch can result in **round(log(batchSize))** invocations as each batch is split into a new batch and as a subsequent invocation. This will repeat until the singular message is eventually bisected out, isolated and DLQ'd. In a best case scenario with a single failed message in a batch of 100, this can result in 7 total invocations. If the lambda is dependent on an external failed service that is entirely unreachable, a 5 message batch will result in every every message being invoked through split batch at least **round(log(batchSize))** times until each message is individually DLQ'd, resulting in a total of **9** invocations: **12345,123,45,12,3,1,2,4,5**

## Idempotent Unsupported Streams

- **Benefits** of the i**dempotent unsupported** recommendation: **Optimized AWS operating costs** and **arbitrary logic/handling flexibility** are the largest benefits. When all logic and handling of message failures, identification, DLQ handling is done at the application code level and can be hyper optimized. As such, it is possible to ensure only a **single lambda invocation** occurs per batch at the stream integration level regardless of number of record failures. Additionally, **any arbitrary mechanism and logic** for retry handling, dlqs and recovery can be utilized (e.g. databases, s3, other streams etc.) This proposal recommends the duel retry-dlq and unrecoverable-dlq methodology as a standard which relies on the infrastructure re-try handling of SQS dlqs once the application logic identifies and processed a failed message for the first time.

- **Drawbacks** to the **idempotent unsupported** recommendation: **Additional engineering overhead, training, implementing reliable /consistent DLQ handling at the application code level**, **and logic code duplication across multiple projects** are the biggest concerns. The code duplication aspect can often be addressed via Lambda Layers or shared libraries pulled in during deployments. However, those layers/libraries require overhead to maintain, update and ensure cross-project compatibility. If on the other hand, each time a new service is launched fresh retry and dlq handing is written specific to that project, it does introduce a statistically non-zero chance of failures being introduced in the implementation logic for that new service. It also requires substantial engineer investment/investigation/fixes when errors occur in addition to building another implementation from scratch as the handling logic can get very nuanced. Even if copy/pasted from another project, once that code handling logic is committed into project's own repository as opposed to being handled via a library/layer, what was once-common-code used for handling of all the different services can diverge significantly over time as each project begins to add exceptions and special handling for their own project's purposes.