ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article Open Access

Strategic Approaches to AWS Lambda Error Resilience: Insights into Sync and Async Invocation Dynamics

Balasubrahmanya Balakrishna

Senior Lead Software Engineer, Richmond, VA, USA

ABSTRACT

This technical paper focuses on efficient error-handling techniques within Lambda functions and offers helpful insights into connecting AWS Lambda with an Application Load Balancer (ALB). Engineers can find practical solutions in the discussion, including illustrated code snippets and focusing on synchronous and asynchronous invocation types.

The article examines the architectural concerns of using an AWS Lambda as the backend for an ALB. With the ALB-to-Lambda architecture in mind, methods for improving error resilience in Lambda functions in sync and async invoke types are discussed.

The report also discusses the subtle differences between throttle and error, two vital metrics. Using real-world examples, engineers will get the skills to implement reliable error-handling procedures adapted to various invocation kinds. The insights offered serve as a brief yet thorough reference for maximizing the performance of AWS Lambda behind an ALB, guaranteeing efficient error management, and tackling the particular difficulties brought forth by sync and async invocations. This tool provides engineers with practical approaches that will enable them to build robust serverless applications in the AWS cloud.

***Corresponding author**
Balasubrahmanya Balakrishna, Senior Lead Software Engineer, Richmond, VA, USA.

Received: March 05, 2022; **Accepted:** March 16, 2022; **Published:** March 24, 2022

Keywords: AWS Lambda, ALB to Lambda, Lambda Invocation Types, Async, Sync, X-Ray, Metrics, Logs, PowerTools for Python, AWS SigV4 Signature

Background

Swift Security Review: AWS Lambda API Frontend by ALB

In the case of ALB to Lambda pattern, shown in Figure 1a below, the 1MB payload limit is a notable constraint. It represents an AWS hard limit that might initially appear arbitrary but has specific reasons behind its existence. Let's delve into the reasons for this limit.

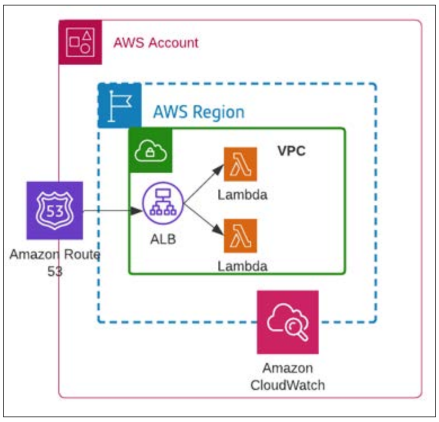


Figure 1a: ALB Fronted AWS Lambda Architecture

Every invocation of AWS APIs mandates generating and including an AWS SigV4 signature in the request [1]. This process involves utilizing your AWS ID and Secret keys to compute an HMAC hash, thereby authenticating your call. The process remains consistent when invoking Lambda-includes all calls to the Lambda Invoke API action, spanning SDK usage, CLI commands, and even interactions from other AWS services like the Application Load Balancer (ALB) [2].

The signing process comprises four steps, as shown in Figure 1b, culminating in adding the HMAC signature to the request header. Step 1 involves including the entire request payload in the calculation, while Step 4 appends the final calculated signature to the Authorization header [3]. This process utilizes the computationally expensive SHA256 hashing algorithm, encountering performance degradation for payloads exceeding 1MB.

Consequently, AWS has imposed a strict payload size limit of 1MB for Lambda functions.

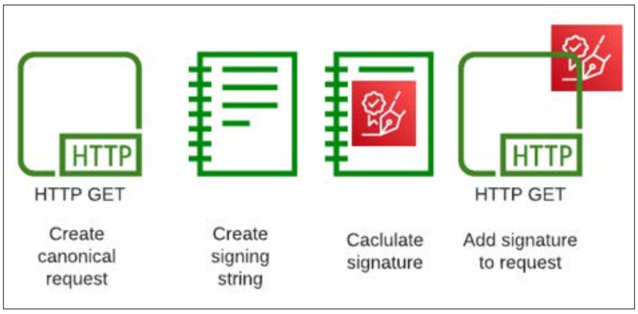


Figure 1b: Signing API Requests Process Overview

A crucial consideration lies in comprehending the importance of restricting the payload to 1MB, especially in the architectural pattern where synchronous traffic moves from ALB to Lambda. This understanding is essential in designing the function and effectively preventing throttling errors.

Introduction

Comprehending the impact of errors in Lambda code is vital for understanding how AWS manages Lambda executions.

Recognizing the two modes of Lambda invocation-Sync and Async-is crucial, given their distinct built-in retry behaviors.

Distinguishing Throttles from Errors lies at the heart of Lambda error handling. AWS makes a clear distinction between these two scenarios and offers separate metrics: Throttles and Errors-the Throttles metric increases when there is inadequate concurrency to invoke the function. Throttled instances leave the function uninvoked, and no code gets executed. Throttles trigger a 429/ Rate Exceeded error; significantly, they do not contribute to the count of Invocations or Errors.

Errors stem from either code issues or uncaught exceptions in the Lambda runtime. AWS attempts to invoke the function in such cases, executing a code segment. When synchronously invoking a function, the execution timeline concludes, and it becomes the client's responsibility to retry the invocationillustrated in the above API pattern (Figure 1b).

In contrast, asynchronous function invocation by AWS includes a default of two retries. These retries signify that AWS will automatically make two additional attempts to invoke the function. You have control over the number of retries and the maximum age of each retry.

Exploring Error Handling in the ALB to Lambda API Pattern for both Sync and Async Invocations

Sync Invoke with Errors

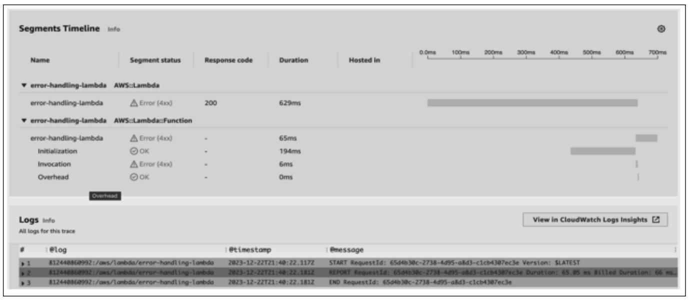
In this scenario, the function deliberately induces an uncaught exception, followed by a synchronous invocation. This invocation triggers a singular execution of the function. The function code and the associated error are below (Figure 2a, 2b and 2c):

```python
import json
print('Loading function')
def lambda_handler(event, context):
    raise Exception('Something went wrong')

```

Figure 2a: Sync Function Code

```
~ aws lambda invoke \
  --function-name error-handling-lambda \
  --invocation-type RequestResponse \
  --cli-binary-format raw-in-base64-out \
  response.json

~ cat response.json
{"errorMessage": "Something went wrong", "errorType": "Exception", "requestId": "65d4b30c-2738-4d95-a8d3
-c1cb4307ec3e", "stackTrace": ["  File \"/var/task/lambda_function.py\", line 4, in lambda_handler\n
raise Exception('Something went wrong')\n"]}%
~ 
```

Figure 2b: Sync Function Invocation



Name	Segment status	Response code	Duration	Hosted in
▼ error-handling-lambda AWS::Lambda				
error-handling-lambda	⚠ Error (4xx)	200	629ms	
▼ error-handling-lambda AWS::Lambda::Function				
error-handling-lambda	⚠ Error (4xx)	-	65ms	
Initialization	✓ OK	-	194ms	
Invocation	⚠ Error (4xx)	-	6ms	
Overhead	✓ OK	-	0ms	

```
#    @log                                              @timestamp                @message
▸ 1  812440860992:/aws/lambda/error-handling-lambda    2023-12-22T21:40:22.117Z  START RequestId: 65d4b30c-2738-4d95-a8d3-c1cb4307ec3e Version: $LATEST
▸ 2  812440860992:/aws/lambda/error-handling-lambda    2023-12-22T21:40:22.181Z  REPORT RequestId: 65d4b30c-2738-4d95-a8d3-c1cb4307ec3e Duration: 65.05 ms Billed Duration: 66 ms…
▸ 3  812440860992:/aws/lambda/error-handling-lambda    2023-12-22T21:40:22.181Z  END RequestId: 65d4b30c-2738-4d95-a8d3-c1cb4307ec3e
```

Figure 2c: Sync Function Error (X-Ray)

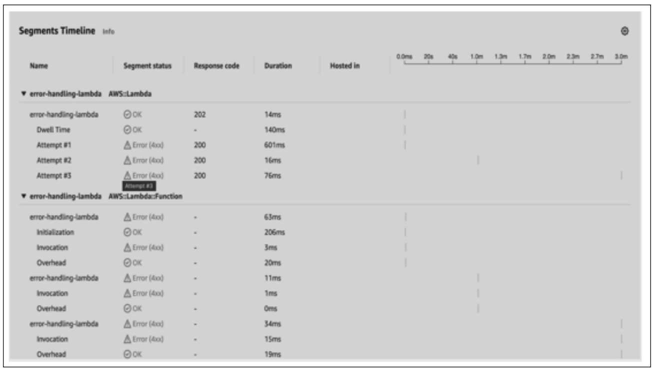
Async Invokes with Errors

Now invoke the same function asynchronously using the AWS CLI. Simply modify the invocation-type flag to 'Event,' as shown in Figure 3a:

```
~     aws lambda invoke \
  --function-name error-handling-lambda \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key1": "value1","key2": "value2","key3": "value3"}' \
  response.json
~  cat response.json
~ 
```

Figure 3a: Async Invocation

A confirmation of the asynchronous invocation is evident from the X-ray trace, where the response code is 202, and there is an observable "Dwell time." Following this, AWS automatically retries the invocation twice. AWS employs an exponential backoff strategy, introducing longer wait intervals between retries. The initial retry occurs after 45 seconds, and the second retry occurs approximately 3 minutes later, as seen in Figure 3b.


Segments Timeline Info

Name	Segment status	Response code	Duration	Hosted in
▼ error-handling-lambda AWS::Lambda				
error-handling-lambda	OK	202	14ms	
Dwell Time	OK	-	140ms	
Attempt #1	Error (4xx)	200	601ms	
Attempt #2	Error (4xx)	200	16ms	
Attempt #3	Error (4xx)	200	76ms	
▼ error-handling-lambda AWS::Lambda::Function				
error-handling-lambda	Error (4xx)	-	63ms	
Initialization	OK	-	206ms	
Invocation	Error (4xx)	-	3ms	
Overhead	OK	-	20ms	
error-handling-lambda	Error (4xx)	-	11ms	
Invocation	Error (4xx)	-	1ms	
Overhead	OK	-	0ms	
error-handling-lambda	Error (4xx)	-	34ms	
Invocation	Error (4xx)	-	15ms	
Overhead	OK	-	19ms	

Figure 3b: Async Function Error (X-Ray)

Figure 3c displays the logs linked with each of the three invocation attempts. It is noteworthy that the RequestId stays consistent across all three attempts.


```
@message
START RequestId: 17038144-378d-4531-9338-53836c33948d Version: $LATEST
REPORT RequestId: 17038144-378d-4531-9338-53836c33948d Duration: 62.92 ms Billed Durati
END RequestId: 17038144-378d-4531-9338-53836c33948d
START RequestId: 17038144-378d-4531-9338-53836c33948d Version: $LATEST
REPORT RequestId: 17038144-378d-4531-9338-53836c33948d Duration: 10.97 ms Billed Durati
END RequestId: 17038144-378d-4531-9338-53836c33948d
START RequestId: 17038144-378d-4531-9338-53836c33948d Version: $LATEST
REPORT RequestId: 17038144-378d-4531-9338-53836c33948d Duration: 34.41 ms Billed Durati
END RequestId: 17038144-378d-4531-9338-53836c33948d
```

Figure 3c: Async Invocation: Logs

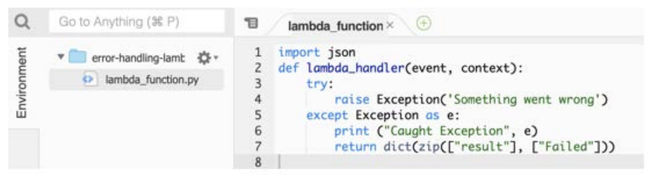
Asynchronous invocations delegate retry logic responsibility to AWS, offering a potent mechanism for minimizing the overall execution duration of your function. In this scenario, implementing exponential retry in your function code could have extended the total duration to approximately 3 minutes, resulting in additional costs. Opting to let the function fail quickly and enabling AWS to manage the reinvoke process proves to be a more efficient and cost-effective approach.

Insights On: Caught Exceptions

Structuring code to capture all errors and avoid surfacing any exceptions to the Lambda runtime.

This approach is essential in specific cases, such as when a function supports an API. In these instances, the function must provide a response, define precise error modes, and, most importantly, safeguard against revealing implementation details [4].

A simple example is outlined below in Figure 4a, and invocation of the function is shown in Figure 4b:



```python
import json
def lambda_handler(event, context):
    try:
        raise Exception('Something went wrong')
    except Exception as e:
        print ("Caught Exception", e)
        return dict(zip(["result"], ["Failed"]))
```

Figure 4a: Caught Exceptions: Function

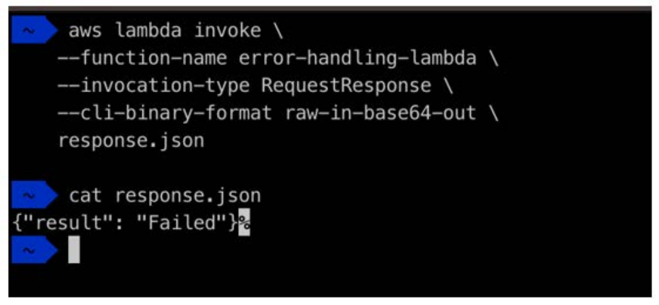

```
~ aws lambda invoke \
  --function-name error-handling-lambda \
  --invocation-type RequestResponse \
  --cli-binary-format raw-in-base64-out \
  response.json

~ cat response.json
{"result": "Failed"}%
~
```

Figure 4b: Caught Exceptions: Sync Invocation

As evident from this X-ray trace, shown in Figure 5a, Lambda does not categorize this as an error, and as a result, the Errors metric remains unaffected. The code captures the exception and responds with a descriptive message, demonstrating a deliberate error-handling strategy.


Segments Timeline Info

Name	Segment status	Response code	Duration	Hosted in
▼ error-handling-lambda AWS::Lambda				
error-handling-lambda	OK	200	8ms	
▼ error-handling-lambda AWS::Lambda::Function				
error-handling-lambda	OK	-	5ms	
Invocation	OK	-	1ms	
Overhead	OK	-	3ms	

Figure 5a: Caught Exceptions X-Ray

Consider an asynchronous invocation-where the primary distinction lies in the response code being 202 instead of 200. Lambda notably interprets this as a successful invocation and refrains from initiating retry attempts, as depicted in Figure 6a and Figure 6b.

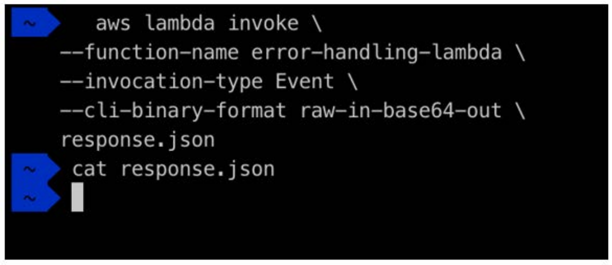

```
~  aws lambda invoke \
  --function-name error-handling-lambda \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  response.json
~ cat response.json
~
```

Figure 6a: Caught Exceptions: Invocation

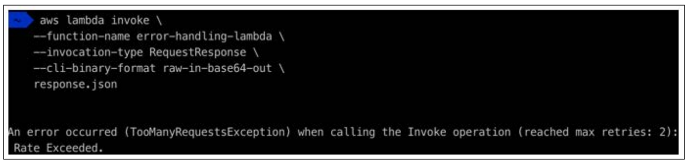

Segments Timeline Info

Name	Segment status	Response code	Duration	Hosted in
▼ error-handling-lambda AWS::Lambda				
error-handling-lambda	OK	202	20ms	
Dwell Time	OK	-	124ms	
Attempt #1	OK	200	502ms	
▼ error-handling-lambda AWS::Lambda::Function				
error-handling-lambda	OK	-	70ms	
Initialization	OK	-	210ms	
Invocation	OK	-	1ms	
Overhead	OK	-	0ms	

Figure 6b: Caught Exceptions: X-Ray

Insights On: Throttles

Let's examine the scenario when throttling occurs. It's crucial to note that throttles are metered independently and do not contribute to the counts of Invokes or Errors. In the context of a synchronous invocation throttled by AWS, the attempt results in a 429 status code, concluding without any subsequent retries, as shown in Figure 7a and Figure 7b.



```
~ aws lambda invoke \
  --function-name error-handling-lambda \
  --invocation-type RequestResponse \
  --cli-binary-format raw-in-base64-out \
  response.json


An error occurred (TooManyRequestsException) when calling the Invoke operation (reached max retries: 2):
 Rate Exceeded.
```

Figure 7a: Throttles: Sync Invocation

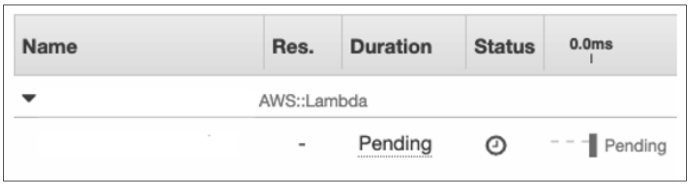


Figure 7b: Throttles: Sync Invocation X-Ray

Upon conducting an asynchronous invocation of the same function, as depicted in Figure 8, observers note a characteristic "Dwell time." This invocation stays in a Pending status until reaching the default maximum event age of 6 hours. It is worth noting that this maximum event age is configurable, spanning from 1 minute to 6 hours.

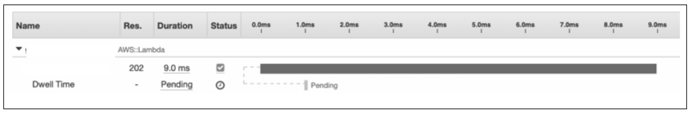


Figure 8: Throttles: Async Invocation X-Ray

Approximately 9 minutes later, account concurrency levels permit AWS to autonomously attempt the execution of the function, as shown in Figure 9. Subsequently, Attempt #1 achieves success around the 9-minute mark.

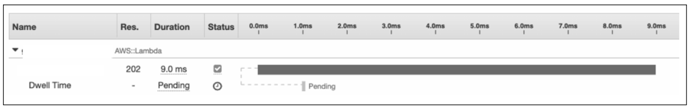


Figure 9: Throttles: Async Invocation Resume X-Ray

Conclusion

To sum up, this technical investigation clarifies the complex workings of AWS Lambda, especially when used with an Application Load Balancer (ALB). The thorough examination addresses various topics, including error-handling techniques and the subtle differences between synchronous and asynchronous calls. Breaking down Errors and throttles, focusing on how they differ and affect Lambda operations.

The paper highlights the value of asynchronous invocations in shifting retry logic to AWS and offers valuable insights into the complexities of AWS Lambda error handling. This idea is very effective, cutting execution time and minimizing possible expenses.

Furthermore, analyzing asynchronous invocations unveils their unique features, including exponential backoff automatic retries and dwell time. The significance of distinct error modes and preventing implementation information leakage in API through examples.

The paper gives engineers practical methods for maximizing Lambda performance while it explores concurrency issues and payload limitations. Integrating X-Ray traces and log analysis provides an additional layer of visibility, making comprehending the nuances at play easier.

In essence, this tech paper serves as a valuable resource for architects and engineers navigating the intricacies of AWS Lambda, providing practical insights, efficient error-handling strategies, and optimization techniques for building resilient and scalable serverless applications within the AWS environment [5].

References

1. Signing AWS API requests. AWS Identity and Access Management-User Guide https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-signing.html.
2. Invoke. AWS Lambda-Developer Guide https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html.
3. Create a signed AWS API request. AWS Identity and Access Management-User Guide https://docs.aws.amazon.com/IAM/latest/UserGuide/create-signed-request.html.
4. Werner Vogels (2021) AWS re:Invent 2021. YouTube https://www.youtube.com/watch?v=8_Xs8Ik0h1w&t=4278s.
5. Working with Lambda function metrics. AWS Lambda-Developer Guide https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html.