Troubleshooting issues in Lambda

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- Troubleshoot configuration issues in Lambda
- Troubleshoot deployment issues in Lambda
- Troubleshoot invocation issues in Lambda
- Troubleshoot execution issues in Lambda
- Troubleshoot event source mapping issues in Lambda
- Troubleshoot networking issues in Lambda

Troubleshoot configuration issues in Lambda

Your function configuration settings can have an impact on the overall performance and behavior of your Lambda function. These may not cause actual function errors, but can cause unexpected timeouts and results.

The following topics provide troubleshooting advice for common issues that you might encounter related to Lambda function configuration settings.

Topics

- Memory configurations
- CPU-bound configurations
- Timeouts
- Memory leakage between invocations
- Asynchronous results returned to a later invocation

Memory configurations

You can configure a Lambda function to use between 128 MB and 10,240 MB of memory. By default, any function created in the console is assigned the smallest amount of memory. Many Lambda functions are performant at this lowest setting. However, if you are importing large code libraries or completing memory intensive tasks, 128 MB is not sufficient.

If your functions are running much slower than expected, the first step is to increase the memory setting. For memory-bound functions, this will resolve the bottleneck and may improve the performance of your function.

CPU-bound configurations

For compute-intensive operations, if your function experiences slower-than-expected performance, this may be due to your function being CPU-bound. In this case, the computational capacity of the function cannot keep pace with the work.

While Lambda doesn't allow you to modify CPU configuration directly, CPU is indirectly controlled via the memory settings. The Lambda service proportionally allocates more virtual CPU as you allocate more memory. At 1.8 GB memory, a Lambda function has an entire vCPU allocated, and above this level it has access to more than one vCPU core. At 10,240MB, it has 6 vCPUs available. In other words, you can improve performance by increasing the memory allocation, even if the function doesn’t use all of the memory.

Timeouts

Timeouts for Lambda functions can be set between 1 and 900 seconds (15 minutes). By default, the Lambda console sets this to 3 seconds. The timeout value is a safety valve that ensures functions do not run indefinitely. After the timeout value is reached, Lambda stops the function invocation.

If a timeout value is set close to the average duration of a function, this increases the risk that the function will time out unexpectedly. The duration of a function can vary based on the amount of data transfer and processing, and the latency of any services the function interacts with. Common causes of timeout include:

- When downloading data from S3 buckets or other data stores, the download is larger or takes longer than average.

- A function makes a request to another service, which takes longer to respond.
- The parameters provided to a function require more computational complexity in the function, which causes the invocation to take longer.

When testing your application, ensure that your tests accurately reflect the size and quantity of data, and realistic parameter values. Importantly, use datasets at the upper bounds of what is reasonably expected for your workload.

Additionally, implement upper-bound limits in your workload wherever practical. In this example, the application could use a maximum size limit for each file type. You can then test the performance of your application for a range of expected file sizes, up to and including the maximum limits.

Memory leakage between invocations

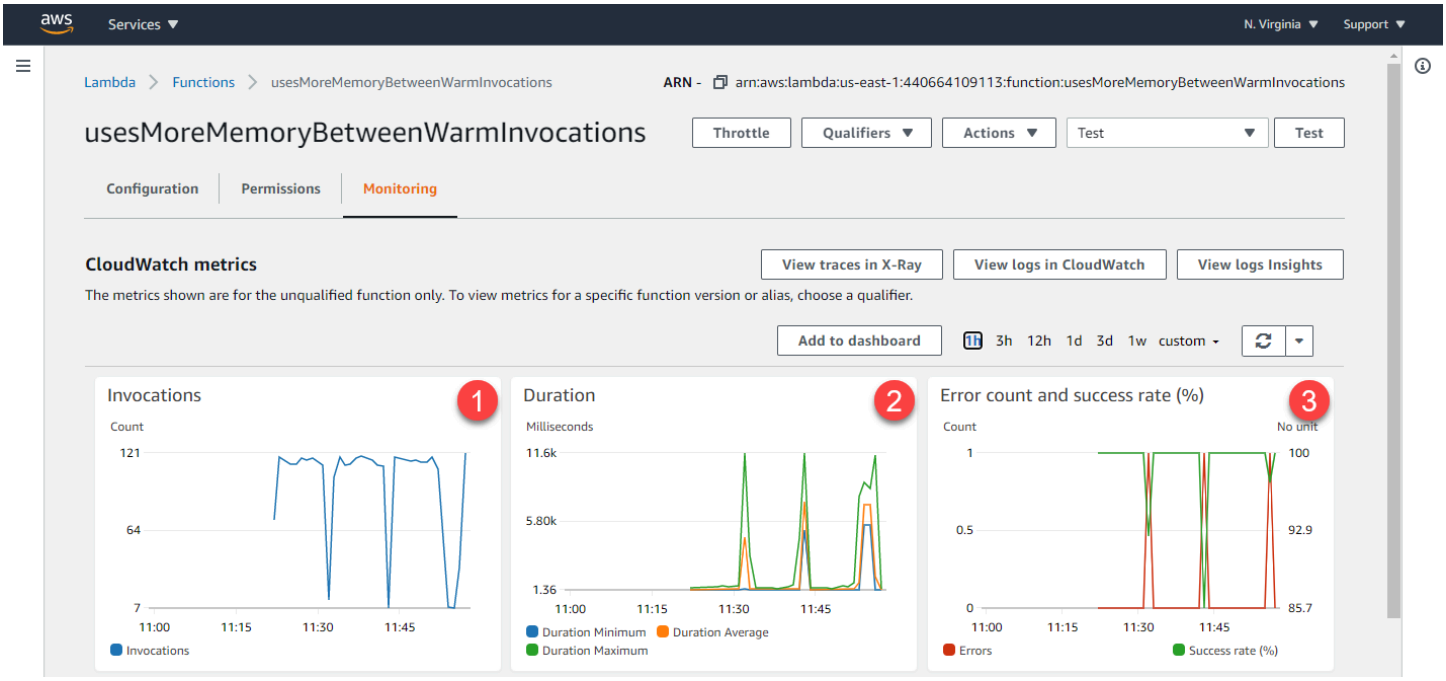
Global variables and objects stored in the INIT phase of a Lambda invocation retain their state between warm invocations. They are completely reset only when the execution environment is run for the first time (also known as a “cold start”). Any variables stored in the handler are destroyed when the handler exits. It’s best practice to use the INIT phase to set up database connections, load libraries, create caches, and load immutable assets.

When you use third-party libraries across multiple invocations in the same execution environment, check their documentation for usage in a serverless compute environment. Some database connection and logging libraries may save intermediate invocation results and other data. This causes the memory usage of these libraries to grow with subsequent warm invocations. If this is the case, you may find the Lambda function runs out of memory, even if your custom code is disposing of variables correctly.

This issue affects invocations occurring in warm execution environments. For example, the following code creates a memory leak between invocations. The Lambda function consumes additional memory with each invocation by increasing the size of a global array:

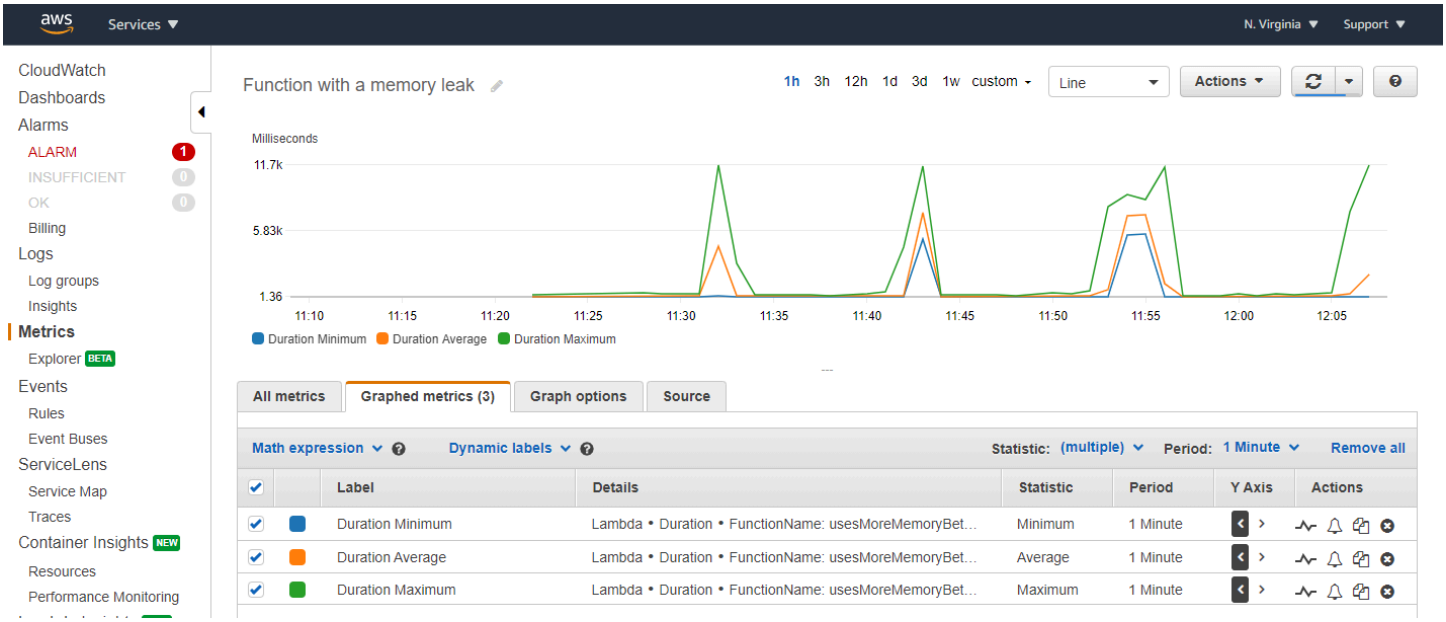
```
let a = []

exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

Configured with 128 MB of memory, after invoking this function 1000 times, the **Monitoring** tab of the Lambda function shows the typical changes in invocations, duration, and error counts when a memory leak occurs:



1. **Invocations** – A steady transaction rate is interrupted periodically as the invocations take longer to complete. During the steady state, the memory leak is not consuming all of the function's allocated memory. As performance degrades, the operating system is paging local storage to accommodate the growing memory required by the function, which results in fewer transactions being completed.
2. **Duration** – Before the function runs out of memory, it finishes invocations at a steady double-digit millisecond rate. As paging occurs, the duration takes an order of magnitude longer.
3. **Error count** – As the memory leak exceeds allocated memory, eventually the function errors due to the computation exceeding the timeout, or the execution environment stops the function.

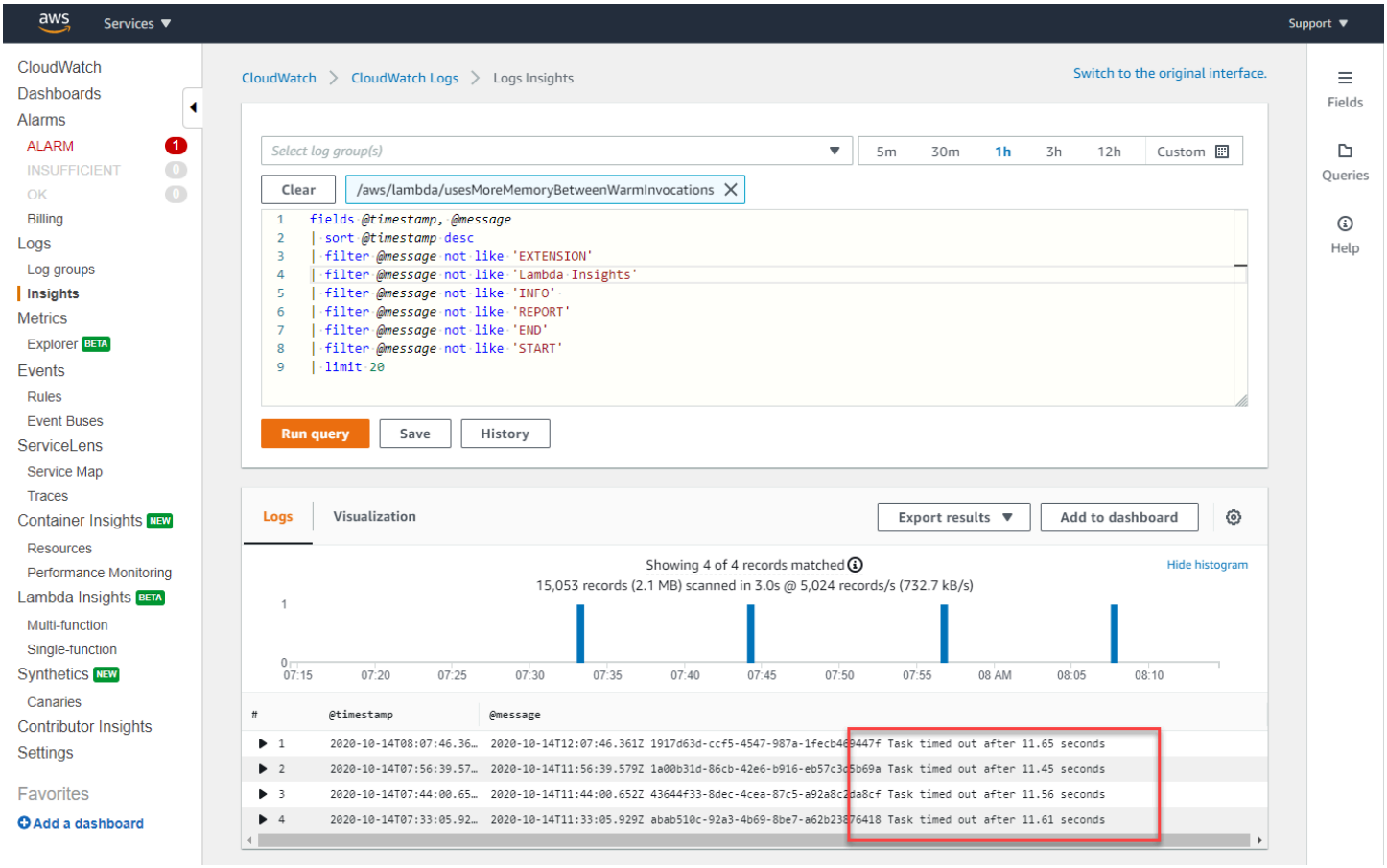
After the error, Lambda restarts the execution environment, which explains why all three graphs show a return to the original state. Expanding the CloudWatch metrics for duration provides more detail for the minimum, maximum and average duration statistics:



To find the errors generated across the 1000 invocations, you can use the CloudWatch Insights query language. The following query excludes informational logs to report only the errors:

```
fields @timestamp, @message
| sort @timestamp desc
| filter @message not like 'EXTENSION'
| filter @message not like 'Lambda Insights'
| filter @message not like 'INFO'
| filter @message not like 'REPORT'
| filter @message not like 'END'
| filter @message not like 'START'
```

When run against the log group for this function, this shows that timeouts were responsible for the periodic errors:



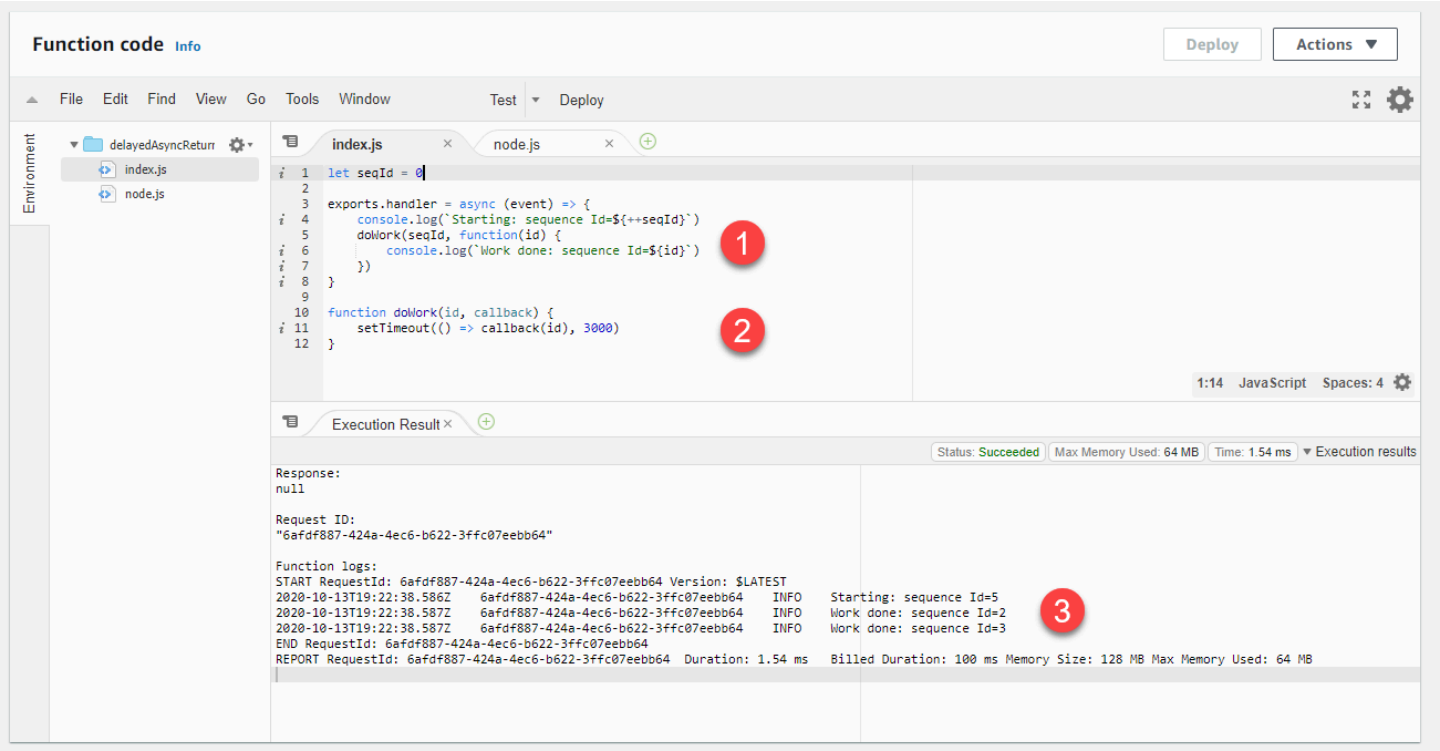
Asynchronous results returned to a later invocation

For function code that uses asynchronous patterns, it's possible for the callback results from one invocation to be returned in a future invocation. This example uses Node.js, but the same logic can apply to other runtimes using asynchronous patterns. The function uses the traditional callback syntax in JavaScript. It calls an asynchronous function with an incremental counter that tracks the number of invocations:

```
let seqId = 0

exports.handler = async (event, context) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    doWork(seqId, function(id) {
        console.log(`Work done: sequence Id=${id}`)
    })
}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)
```

```
}
```

When invoked several times in succession, the results of the callbacks occur in subsequent invocations:



1. The code calls the `doWork` function, providing a callback function as the last parameter.
2. The `doWork` function takes some period of time to complete before invoking the callback.
3. The function's logging indicates that the invocation is ending before the `doWork` function finishes execution. Additionally, after starting an iteration, callbacks from previous iterations are being processed, as shown in the logs.

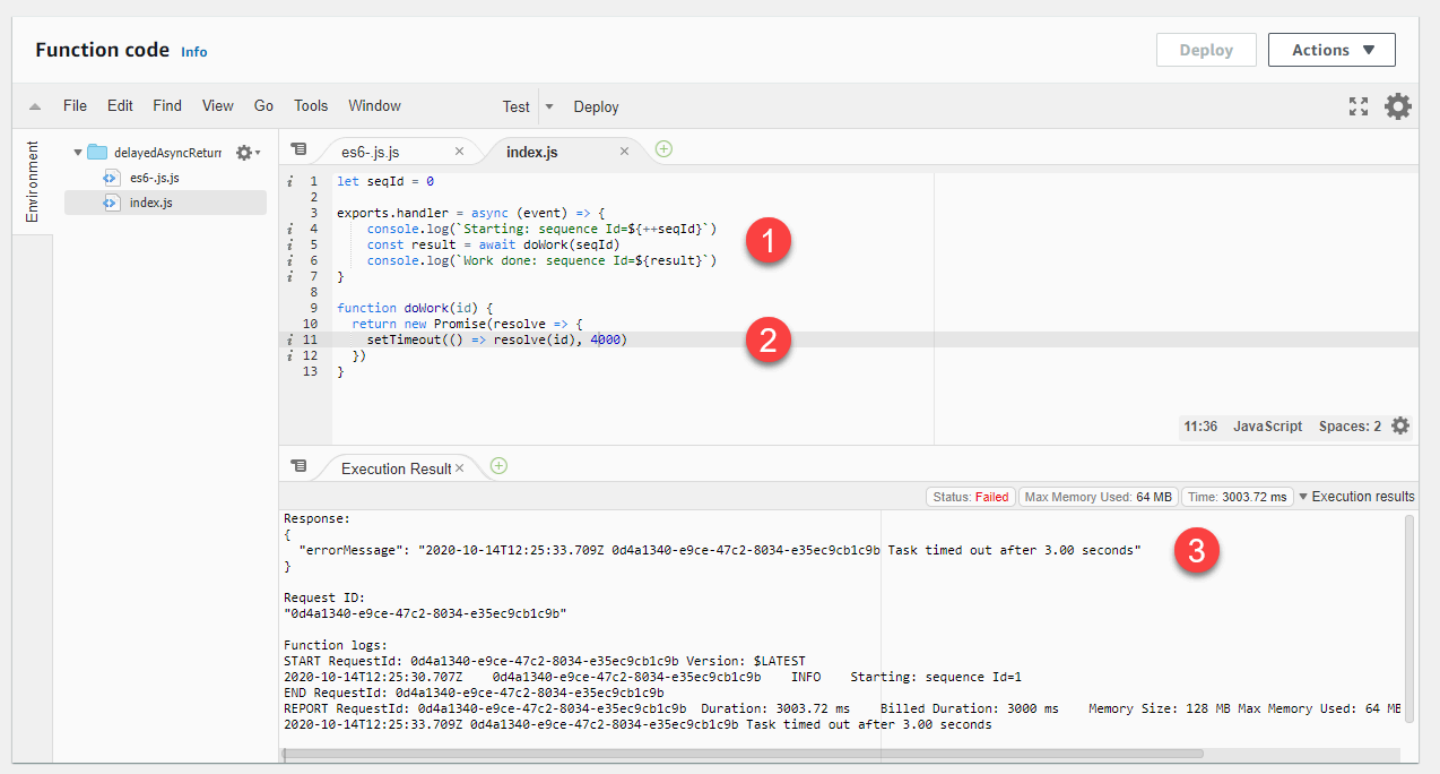
In JavaScript, asynchronous callbacks are handled with an event loop. Other runtimes use different mechanisms to handle concurrency. When the function's execution environment ends, Lambda freezes the environment until the next invocation. After it resumes, JavaScript continues processing the event loop, which in this case includes an asynchronous callback from a previous invocation. Without this context, it can appear that the function is running code for no reason, and returning arbitrary data. In fact, it is really an artifact of how runtime concurrency and the execution environments interact.

This creates the potential for private data from a previous invocation to appear in a subsequent invocation. There are two ways to prevent or detect this behavior. First, JavaScript provides the

async and await keywords to simplify asynchronous development and also force code execution to wait for an asynchronous call to complete. The function above can be rewritten using this approach as follows:

```
let seqId = 0
exports.handler = async (event) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    const result = await doWork(seqId)
    console.log(`Work done: sequence Id=${result}`)
}

function doWork(id) {
  return new Promise(resolve => {
    setTimeout(() => resolve(id), 4000)
  })
}
```

Using this syntax prevents the handler from exiting before the asynchronous function is finished. In this case, if the callback takes longer than the Lambda function's timeout, the function will throw an error, instead of returning the callback result in a later invocation:

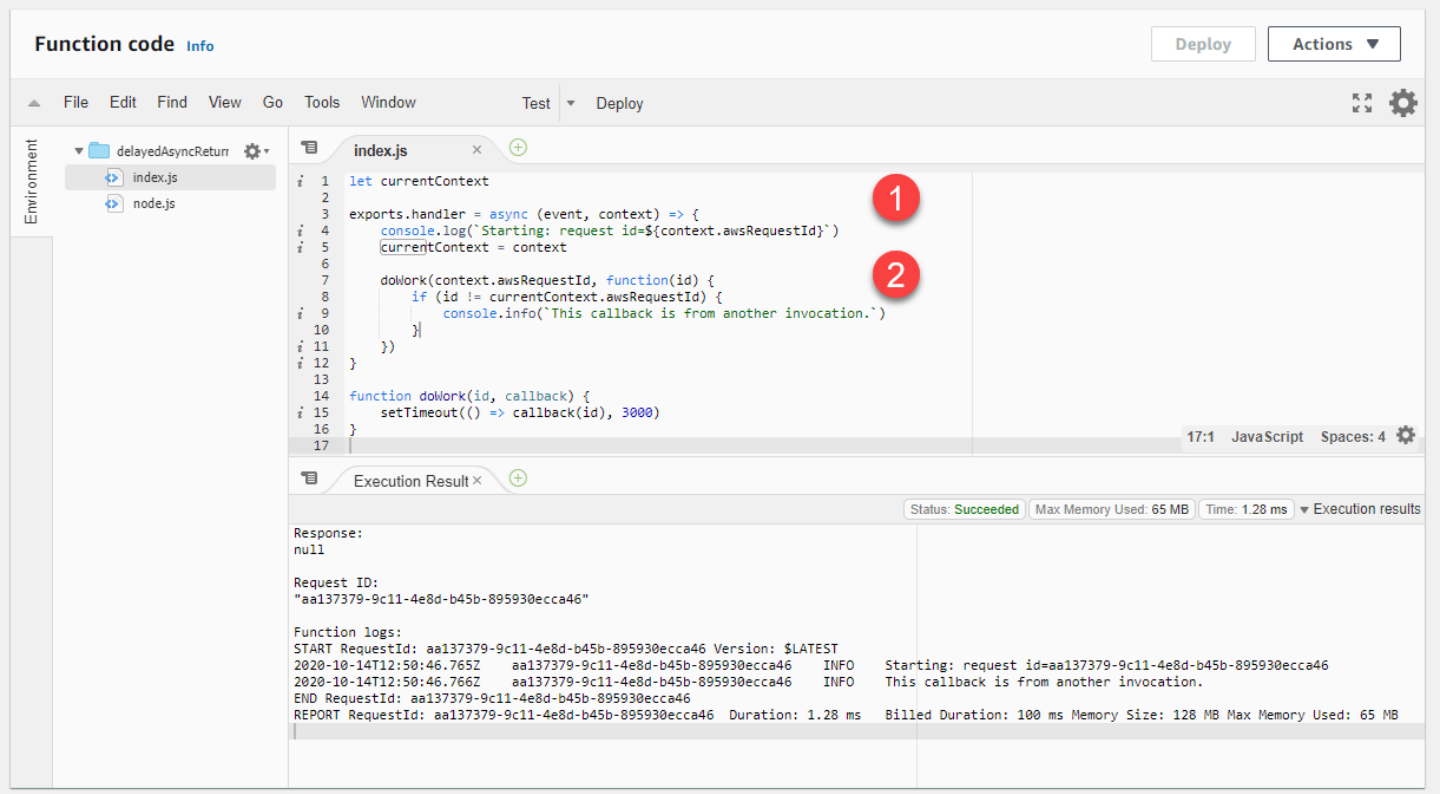


1. The code calls the asynchronous `doWork` function using the await keyword in the handler.

2. The `doWork` function takes some period of time to complete before resolving the promise.
3. The function times out because `doWork` takes longer than the timeout limit allows and the callback result is not returned in a later invocation.

Generally, you should make sure any background processes or callbacks in the code are complete before the code exits. If this is not possible in your use case, you can use an identifier to ensure that the callback belongs to the current invocation. To do this, you can use the *awsRequestId* provided by the context object. By passing this value to the asynchronous callback, you can compare the passed value with the current value to detect if the callback originated from another invocation:

```
let currentContext

exports.handler = async (event, context) => {
    console.log(`Starting: request id=$\{context.awsRequestId}`)
    currentContext = context

    doWork(context.awsRequestId, function(id) {
        if (id != currentContext.awsRequestId) {
            console.info(`This callback is from another invocation.`)
        }
    })

}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)

}
```



1. The Lambda function handler takes the context parameter, which provides access to a unique invocation request ID.
2. The `awsRequestId` is passed to the doWork function. In the callback, the ID is compared with the `awsRequestId` of the current invocation. If these values are different, the code can take action accordingly.

Troubleshoot deployment issues in Lambda

When you update your function, Lambda deploys the change by launching new instances of the function with the updated code or settings. Deployment errors prevent the new version from being used and can arise from issues with your deployment package, code, permissions, or tools.

When you deploy updates to your function directly with the Lambda API or with a client such as the AWS CLI, you can see errors from Lambda directly in the output. If you use services like AWS CloudFormation, AWS CodeDeploy, or AWS CodePipeline, look for the response from Lambda in the logs or event stream for that service.

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- General: Permission is denied / Cannot load such file
- General: Error occurs when calling the UpdateFunctionCode
- Amazon S3: Error Code PermanentRedirect.
- General: Cannot find, cannot load, unable to import, class not found, no such file or directory
- General: Undefined method handler
- General: Lambda code storage limit exceeded
- Lambda: Layer conversion failed
- Lambda: InvalidParameterValueException or RequestEntityTooLargeException
- Lambda: InvalidParameterValueException
- Lambda: Concurrency and memory quotas
- Lambda: Invalid alias configuration for provisioned concurrency

General: Permission is denied / Cannot load such file

Error: *EACCES: permission denied, open '/var/task/index.js'*

Error: *cannot load such file -- function*

Error: *[Errno 13] Permission denied: '/var/task/function.py'*

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the chmod command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

General: Error occurs when calling the UpdateFunctionCode

Error: *An error occurred (RequestEntityTooLargeException) when calling the UpdateFunctionCode operation*

When you upload a deployment package or layer archive directly to Lambda, the size of the ZIP file is limited to 50 MB. To upload a larger file, store it in Amazon S3 and use the S3Bucket and S3Key parameters.

> **Note**
>
> When you upload a file directly with the AWS CLI, AWS SDK, or otherwise, the binary ZIP file is converted to base64, which increases its size by about 30%. To allow for this, and the size of other parameters in the request, the actual request size limit that Lambda applies is larger. Due to this, the 50 MB limit is approximate.

Amazon S3: Error Code PermanentRedirect.

Error: *Error occurred while GetObject. S3 Error Code: PermanentRedirect. S3 Error Message: The bucket is in this region: us-east-2. Please use this region to retry the request*

When you upload a function's deployment package from an Amazon S3 bucket, the bucket must be in the same Region as the function. This issue can occur when you specify an Amazon S3 object in a

call to UpdateFunctionCode, or use the package and deploy commands in the AWS CLI or AWS SAM CLI. Create a deployment artifact bucket for each Region where you develop applications.

General: Cannot find, cannot load, unable to import, class not found, no such file or directory

Error: *Cannot find module 'function'*

Error: *cannot load such file -- function*

Error: *Unable to import module 'function'*

Error: *Class not found: function.Handler*

Error: *fork/exec /var/task/function: no such file or directory*

Error: *Unable to load type 'Function.Handler' from assembly 'Function'.*

The name of the file or class in your function's handler configuration doesn't match your code. See the following section for more information.

General: Undefined method handler

Error: *index.handler is undefined or not exported*

Error: *Handler 'handler' missing on module 'function'*

Error: *undefined method `handler' for #<LambdaHandler:0x000055b76ccebf98>*

Error: *No public method named handleRequest with appropriate method signature found on class function.Handler*

Error: *Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function'*

The name of the handler method in your function's handler configuration doesn't match your code. Each runtime defines a naming convention for handlers, such as *`filename.methodname`*. The handler is the method in your function's code that the runtime runs when your function is invoked.

For some languages, Lambda provides a library with an interface that expects a handler method to have a specific name. For details about handler naming for each language, see the following topics.

- Building Lambda functions with Node.js

- Building Lambda functions with Python
- Building Lambda functions with Ruby
- Building Lambda functions with Java
- Building Lambda functions with Go
- Building Lambda functions with C#
- Building Lambda functions with PowerShell

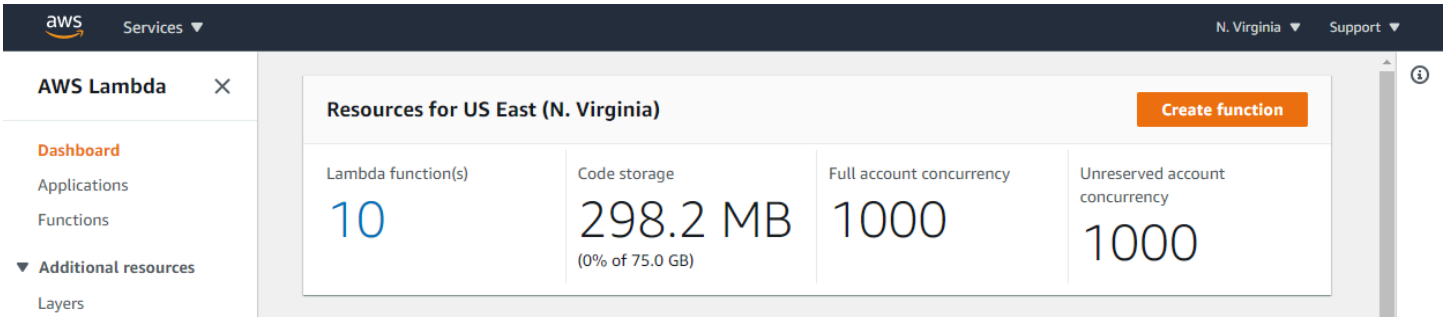
General: Lambda code storage limit exceeded

Error: *Code storage limit exceeded.*

Lambda stores your function code in an internal S3 bucket that's private to your account. Each AWS account is allocated 75 GB of storage in each Region. Code storage includes the total storage used by both Lambda functions and layers. If you reach the quota, you receive a *CodeStorageExceededException* when you attempt to deploy new functions.

Manage the storage space available by cleaning up old versions of functions, removing unused code, or using Lambda layers. In addition, it's good practice to use separate AWS accounts for separate workloads to help manage storage quotas.

You can view your total storage usage in the Lambda console, under the **Dashboard** submenu:



Lambda: Layer conversion failed

Error: *Lambda layer conversion failed. For advice on resolving this issue, see the Troubleshoot deployment issues in Lambda page in the Lambda User Guide.*

When you configure a Lambda function with a layer, Lambda merges the layer with your function code. If this process fails to complete, Lambda returns this error. If you encounter this error, take the following steps:

- Delete any unused files from your layer
- Delete any symbolic links in your layer
- Rename any files that have the same name as a directory in any of your function's layers

Lambda: InvalidParameterValueException or RequestEntityTooLargeException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided exceeded the 4KB limit. String measured: {"A1":"uSFeY5cyPiPn7AtnX5BsM...*

Error: *RequestEntityTooLargeException: Request must be smaller than 5120 bytes for the UpdateFunctionConfiguration operation*

The maximum size of the variables object that is stored in the function's configuration must not exceed 4096 bytes. This includes key names, values, quotes, commas, and brackets. The total size of the HTTP request body is also limited.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

In this example, the object is 39 characters and takes up 39 bytes when it's stored (without white space) as the string `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}`. Standard ASCII characters in environment variable values use one byte each. Extended ASCII and Unicode characters can use between 2 bytes and 4 bytes per character.

Lambda: InvalidParameterValueException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided contains reserved keys that are currently not supported for modification.*

Lambda reserves some environment variable keys for internal use. For example, `AWS_REGION` is used by the runtime to determine the current Region and cannot be overridden. Other variables, like `PATH`, are used by the runtime but can be extended in your function configuration. For a full list, see Defined runtime environment variables.

Lambda: Concurrency and memory quotas

Error: *Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value*

Error: *'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008*

These errors occur when you exceed the concurrency or memory quotas for your account. New AWS accounts have reduced concurrency and memory quotas. To resolve errors related to concurrency, you can request a quota increase. You cannot request memory quota increases.

- **Concurrency:** You might get an error if you try to create a function using reserved or provisioned concurrency, or if your per-function concurrency request (PutFunctionConcurrency) exceeds your account's concurrency quota.
- **Memory:** Errors occur if the amount of memory allocated to the function exceeds your account's memory quota.

Lambda: Invalid alias configuration for provisioned concurrency

Error: *Invalid alias configuration for provisioned concurrency*

This error occurs when you try to update a Lambda function's code or configuration while an alias with provisioned concurrency is pointing to a version that has issues. Lambda pre-initializes execution environments for provisioned concurrency, and if these environments can't be properly initialized due to code errors, resource constraints, or affected stack and alias, the deployment fails. If you encounter this issue, take the following steps:

1. **Roll back the alias:** Temporarily update the alias to point to the previously working version.

```
aws lambda update-alias \
 --function-name <function-name> \
 --name <alias-name> \
 --function-version <known-good-version>
```

2. **Fix Lambda initialization code:** Ensure the initialization code that runs outside the handler doesn't have any uncaught exceptions and initialize the clients and connections.
3. **Redeploy safety:** Deploy fixed code and publish a new version. Then, update alias to point to the fixed version. Optionally, re-enable provisioned concurrency, if necessary.

If using AWS CloudFormation, update stack definition `FunctionVersion:!GetAtt version.Version` so that the alias points to the working version:

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

Troubleshoot invocation issues in Lambda

When you invoke a Lambda function, Lambda validates the request and checks for scaling capacity before sending the event to your function or, for asynchronous invocation, to the event queue. Invocation errors can be caused by issues with request parameters, event structure, function settings, user permissions, resource permissions, or limits.

If you invoke your function directly, you see any invocation errors in the response from Lambda. If you invoke your function asynchronously with an event source mapping or through another service, you might find errors in logs, a dead-letter queue, or a failed-event destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

For a list of error types that the `Invoke` operation can return, see Invoke.

Topics

- Lambda: Function times out during Init phase (Sandbox.Timedout)
- IAM: lambda:InvokeFunction not authorized
- Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint)
- Lambda: Operation cannot be performed ResourceConflictException
- Lambda: Function is stuck in Pending
- Lambda: One function is using all concurrency
- General: Cannot invoke function with other accounts or services
- General: Function invocation is looping
- Lambda: Alias routing with provisioned concurrency
- Lambda: Cold starts with provisioned concurrency
- Lambda: Cold starts with new versions
- Lambda: Unexpected Node.js exit in runtime (Runtime.NodejsExit)
- EFS: Function could not mount the EFS file system
- EFS: Function could not connect to the EFS file system
- EFS: Function could not mount the EFS file system due to timeout
- Lambda: Lambda detected an IO process that was taking too long
- Container: CodeArtifactUserException errors
- Container: InvalidEntrypoint errors

Lambda: Function times out during Init phase (Sandbox.Timedout)

Error: *Task timed out after 3.00 seconds*

When the Init phase times out, Lambda initializes the execution environment again by re-running the `Init` phase when the next invoke request arrives. This is called a suppressed init. However, if your function is configured with a short timeout duration (generally around 3 seconds), the suppressed init might not complete during the allocated timeout duration, causing the `Init` phase to time out again. Alternatively, the suppressed init completes but does not leave enough time for the Invoke phase to complete, causing the `Invoke` phase to time out.

To reduce timeout errors, use one or more of the following strategies:

- **Increase the function timeout duration**: Extend the timeout to give the `Init` and `Invoke` phases time to complete successfully.
- **Increase the function memory allocation**: More memory also means more proportional CPU allocation, which can speed up both the `Init` and `Invoke` phases.
- **Optimize the function initialization code**: Reduce the time needed for initialization to ensure that the the `Init` and `Invoke` phase can complete within the configured timeout.

IAM: lambda:InvokeFunction not authorized

Error: *User: arn:aws:iam::123456789012:user/developer is not authorized to perform: lambda:InvokeFunction on resource: my-function*

Your user, or the role that you assume, must have permission to invoke a function. This requirement also applies to Lambda functions and other compute resources that invoke functions. Add the AWS managed policy **AWSLambdaRole** to your user, or add a custom policy that allows the `lambda:InvokeFunction` action on the target function.

> **Note**
>
> The name of the IAM action (`lambda:InvokeFunction`) refers to the `Invoke` Lambda API operation.

For more information, see Managing permissions in AWS Lambda .

Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint)

Error: *Couldn't find valid bootstrap(s): [/var/task/bootstrap /opt/bootstrap]*

This error typically occurs when the root of your deployment package doesn't contain an executable file named `bootstrap`. For example, if you're deploying a `provided.al2023` function with a .zip file, the `bootstrap` file must be at the root of the .zip file, not in a directory.

Lambda: Operation cannot be performed ResourceConflictException

Error: *ResourceConflictException: The operation cannot be performed at this time. The function is currently in the following state: Pending*

When you connect a function to a virtual private cloud (VPC) at the time of creation, the function enters a `Pending` state while Lambda creates elastic network interfaces. During this time, you can't invoke or modify your function. If you connect your function to a VPC after creation, you can invoke it while the update is pending, but you can't modify its code or configuration.

For more information, see Lambda function states .

Lambda: Function is stuck in Pending

Error: *A function is stuck in the `Pending` state for several minutes.*

If a function is stuck in the `Pending` state for more than six minutes, call one of the following API operations to unblock it:

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda cancels the pending operation and puts the function into the `Failed` state. You can then attempt another update.

Lambda: One function is using all concurrency

Issue: *One function is using all of the available concurrency, causing other functions to be throttled.*

To divide your AWS account's available concurrency in an AWS Region into pools, use reserved concurrency. Reserved concurrency ensures that a function can always scale to its assigned concurrency, and that it doesn't scale beyond its assigned concurrency.

General: Cannot invoke function with other accounts or services

Issue: *You can invoke your function directly, but it doesn't run when another service or account invokes it.*

You grant other services and accounts permission to invoke a function in the function's resource-based policy. If the invoker is in another account, that user must also have permission to invoke functions.

General: Function invocation is looping

Issue: *Function is invoked continuously in a loop.*

This typically occurs when your function manages resources in the same AWS service that triggers it. For example, it's possible to create a function that stores an object in an Amazon Simple Storage Service (Amazon S3) bucket that's configured with a notification that invokes the function again. To stop the function from running, reduce the available concurrency to zero, which throttles all future invocations. Then, identify the code path or configuration error that caused the recursive invocation. Lambda automatically detects and stops recursive loops for some AWS services and SDKs. For more information, see the section called “Recursive loop detection”.

Lambda: Alias routing with provisioned concurrency

Issue: *Provisioned concurrency spillover invocations during alias routing.*

Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid spillover invocations by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

Lambda: Cold starts with provisioned concurrency

Issue: *You see cold starts after enabling provisioned concurrency.*

When the number of concurrent executions on a function is less than or equal to the configured level of provisioned concurrency, there shouldn't be any cold starts. To help you confirm if provisioned concurrency is operating normally, do the following:

- Check that provisioned concurrency is enabled on the function version or alias.

  > **Note**
  >
  > Provisioned concurrency is not configurable on the unpublished version of the function ($LATEST).

- Ensure that your triggers invoke the correct function version or alias. For example, if you're using Amazon API Gateway, check that API Gateway invokes the function version or alias with provisioned concurrency, not $LATEST. To confirm that provisioned concurrency is being used,

you can check the ProvisionedConcurrencyInvocations Amazon CloudWatch metric. A non-zero value indicates that the function is processing invocations on initialized execution environments.

- Determine whether your function concurrency exceeds the configured level of provisioned concurrency by checking the ProvisionedConcurrencySpilloverInvocations CloudWatch metric. A non-zero value indicates that all provisioned concurrency is in use and some invocation occurred with a cold start.
- Check your invocation frequency (requests per second). Functions with provisioned concurrency have a maximum rate of 10 requests per second per provisioned concurrency. For example, a function configured with 100 provisioned concurrency can handle 1,000 requests per second. If the invocation rate exceeds 1,000 requests per second, some cold starts can occur.

Lambda: Cold starts with new versions

Issue: *You see cold starts while deploying new versions of your function.*

When you update a function alias, Lambda automatically shifts provisioned concurrency to the new version based on the weights configured on the alias.

Error: *KMSDisabledException: Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Please check the function's KMS key settings.*

This error can occur if your AWS Key Management Service (AWS KMS) key is disabled, or if the grant that allows Lambda to use the key is revoked. If the grant is missing, configure the function to use a different key. Then, reassign the custom key to recreate the grant.

Lambda: Unexpected Node.js exit in runtime (Runtime.NodejsExit)

Issue: *Lambda runtime client detected an unexpected Node.js exit code.*

This error occurs when your function exits before all Promises are settled, for example due to a code bug. It can also occur when Node.js detects a deadlock that prevents Promises from being settled. This error affects only async style handlers, not callback-style handlers.

Affected runtimes: Node.js 18 and later.

To resolve this issue:

1. Check your function code for unsettled promises in async handlers.
2. Ensure all promises are properly settled (resolved or rejected) before the function completes.

3. Review your code for potential race conditions in asynchronous operations.

For more information about Node.js exit codes and process termination, see the Node.js documentation.

EFS: Function could not mount the EFS file system

Error: *EFSMountFailureException: The function could not mount the EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd.*

The mount request to the function's file system was rejected. Check the function's permissions, and confirm that its file system and access point exist and are ready for use.

EFS: Function could not connect to the EFS file system

Error: *EFSMountConnectivityException: The function couldn't connect to the Amazon EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd. Check your network configuration and try again.*

The function couldn't establish a connection to the function's file system with the NFS protocol (TCP port 2049). Check the security group and routing configuration for the VPC's subnets.

If you get these errors after updating your function's VPC configuration settings, try unmounting and remounting the file system.

EFS: Function could not mount the EFS file system due to timeout

Error: *EFSMountTimeoutException: The function could not mount the EFS file system with access point {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} due to mount time out.*

The function could connect to the function's file system, but the mount operation timed out. Try again after a short time and consider limiting the function's concurrency to reduce load on the file system.

Lambda: Lambda detected an IO process that was taking too long

EFSIOException: This function instance was stopped because Lambda detected an IO process that was taking too long.

A previous invocation timed out and Lambda couldn't terminate the function handler. This issue can occur when an attached file system runs out of burst credits and the baseline throughput is insufficient. To increase throughput, you can increase the size of the file system or use provisioned throughput.

Container: CodeArtifactUserException errors

Error: *CodeArtifactUserPendingException error message*

The CodeArtifact is pending optimization. The function transitions to the Active state when Lambda completes the optimization. HTTP response code 409.

Error: *CodeArtifactUserDeletedException error message*

The CodeArtifact is scheduled to be deleted. HTTP response code 409.

Error: *CodeArtifactUserFailedException error message*

Lambda failed to optimize the code. You need to correct the code and upload it again. HTTP response code 409.

Container: InvalidEntrypoint errors

Error: *Runtime.ExitError or "errorType": "Runtime.InvalidEntrypoint"*

Verify that the ENTRYPOINT to your container image includes the absolute path as the location. Also verify that the image does not contain a symlink as the ENTRYPOINT.

Error: *You are using an CloudFormation template, and your container ENTRYPOINT is being overridden with a null or empty value.*

Review the ImageConfig resource in the CloudFormation template. If you declare an `ImageConfig` resource in your template, you must provide non-empty values for all three of the properties.

Troubleshoot execution issues in Lambda

When the Lambda runtime runs your function code, the event might be processed on an instance of the function that's been processing events for some time, or it might require a new instance to be initialized. Errors can occur during function initialization, when your handler code processes the event, or when your function returns (or fails to return) a response.

Function execution errors can be caused by issues with your code, function configuration, downstream resources, or permissions. If you invoke your function directly, you see function errors in the response from Lambda. If you invoke your function asynchronously, with an event source mapping, or through another service, you might find errors in logs, a dead-letter queue, or an on-failure destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

When your function code or the Lambda runtime return an error, the status code in the response from Lambda is 200 OK. The presence of an error in the response is indicated by a header named `X-Amz-Function-Error`. 400 and 500-series status codes are reserved for invocation errors.

Topics

- Lambda: Remote debugging with Visual Studio Code
- Lambda: Execution takes too long
- Lambda: Unexpected event payload
- Lambda: Unexpectedly large payload sizes
- Lambda: JSON encoding and decoding errors
- Lambda: Logs or traces don't appear
- Lambda: Not all of my function's logs appear
- Lambda: The function returns before execution finishes
- Lambda: Running an unintended function version or alias
- Lambda: Detecting infinite loops
- General: Downstream service unavailability
- AWS SDK: Versions and updates
- Python: Libraries load incorrectly
- Java: Your function takes longer to process events after updating to Java 17 from Java 11
- Kafka: Error handling and retry configuration issues

Lambda: Remote debugging with Visual Studio Code

Issue: *Difficulty troubleshooting complex Lambda function behavior in the actual AWS environment*

Lambda provides a remote debugging feature through the AWS Toolkit for Visual Studio Code. For set up and general instructions, see Remotely debug Lambda functions with Visual Studio Code.

For detailed instructions on troubleshooting, advanced use cases, and region availability, see Remote debugging Lambda functions in the AWS Toolkit for Visual Studio Code User Guide.

Lambda: Execution takes too long

Issue: *Function execution takes too long.*

If your code takes much longer to run in Lambda than on your local machine, it may be constrained by the memory or processing power available to the function. Configure the function with additional memory to increase both memory and CPU.

Lambda: Unexpected event payload

Issue: *Function errors related to malformed JSON or inadequate data validation.*

All Lambda functions receive an event payload in the first parameter of the handler. The event payload is a JSON structure that may contain arrays and nested elements.

Malformed JSON can occur when provided by upstream services that do not use a robust process for checking JSON structures. This occurs when services concatenate text strings or embed user input that has not been sanitized. JSON is also frequently serialized for passing between services. Always parse JSON structures both as the producer and consumer of JSON to ensure that the structure is valid.

Similarly, failing to check for ranges of values in the event payload can result in errors. This example shows a function that calculates a tax withholding:

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

This function uses a salary and tax rate from the event payload to perform the calculation. However, the code fails to check if the attributes are present. It also fails to check data types, or ensure boundaries, such as ensuring that the tax percentage is between 0 and 1. As a result, values outside of these bounds produce nonsensical results. An incorrect type or missing attribute causes a runtime error.

Create tests to ensure that your function handles larger payload sizes. The maximum size for a Lambda event payload is 1 MB. Depending upon the content, larger payloads may mean more items passed to the function or more binary data embedded in a JSON attribute. In both cases, this can result in more processing for a Lambda function.

Larger payloads can also cause timeouts. For example, a Lambda function processes one record per 100 ms and has a timeout of 3 seconds. Processing is successful for 0-29 items in the payload. However, once the payload contains more than 30 items, the function times out and throws an error. To avoid this, ensure that timeouts are set to handle the additional processing time for the maximum number of items expected.

Lambda: Unexpectedly large payload sizes

Issue: *Functions are timing out or causing errors due to large payloads.*

Larger payloads can cause timeouts and errors. We recommend creating tests to ensure that your function handles your largest expected payloads, and ensuring the function timeout is properly set.

In addition, certain event payloads can contain pointers to other resources. For example, a Lambda function with 128 MB of memory may perform image processing on a JPG file stored as an object in S3. The function works as expected with smaller image files. However, when a larger JPG file is provided as input, the Lambda function throws an error due to running out of memory. To avoid this, the test cases should include examples from the upper bounds of expected data sizes. The code should also validate payload sizes.

Lambda: JSON encoding and decoding errors

Issue: *NoSuchKey exception when parsing JSON inputs.*

Check to ensure you are processing JSON attributes correctly. For example, for events generated by S3, the `s3.object.key` attribute contains a URL encoded object key name. Many functions process this attribute as text to load the referenced S3 object:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

This code works with the key name `james.jpg` but throws a `NoSuchKey` error when the name is `james beswick.jpg`. Since URL encoding converts spaces and other characters in a key name, you must ensure that functions decode keys before using this data:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "))
}).promise()
```

Lambda: Logs or traces don't appear

Issue: *Logs don't appear in CloudWatch Logs.*

Issue: *Traces don't appear in AWS X-Ray.*

Your function needs permission to call CloudWatch Logs and X-Ray. Update its execution role to grant it permission. Add the following managed policies to enable logs and tracing.

- **AWSLambdaBasicExecutionRole**
- **AWSXRayDaemonWriteAccess**

When you add permissions to your function, perform a trivial update to its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

> **Note**
>
> It may take 5 to 10 minutes for logs to show up after a function invocation.

Lambda: Not all of my function's logs appear

Issue: *Function logs are missing in CloudWatch Logs, even though my permissions are correct*

If your AWS account reaches its CloudWatch Logs quota limits, CloudWatch throttles function logging. When this happens, some of the logs output by your functions may not appear in CloudWatch Logs.

If your function outputs logs at too high a rate for Lambda to process them, this can also cause log outputs not to appear in CloudWatch Logs. When Lambda can't send logs to CloudWatch at the rate your function produces them, it drops logs to prevent the execution of your function from slowing down. Expect to consistently observe dropped logs when your log throughput exceeds 2 MB/s for a single log stream.

If your function is configured to use JSON formatted logs, Lambda tries to send a `logsDropped` event to CloudWatch Logs when it drops logs. However, when CloudWatch throttles your function's logging, this event might not reach CloudWatch Logs, so you won't always see a record when Lambda drops logs.

To check if your AWS account has reached its CloudWatch Logs quota limits, do the following:

1. Open the Service Quotas console.
2. In the navigation pane, choose **AWS services**.
3. From the **AWS services** list, search for Amazon CloudWatch Logs.
4. In the **Service quotas** list, choose the `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second` and `PutLogEvents throttle limit in transactions per second` quotas to view your utilization.

You can also set CloudWatch alarms to alert you when your account utilization exceeds a limit you specify for these quotas. See Create a CloudWatch alarm based on a static threshold to learn more.

If the default quota limits for CloudWatch Logs aren't enough for your use case, you can request a quota increase.

Lambda: The function returns before execution finishes

Issue: (Node.js) *Function returns before code finishes executing*

Many libraries, including the AWS SDK, operate asynchronously. When you make a network call or perform another operation that requires waiting for a response, libraries return an object called a promise that tracks the progress of the operation in the background.

To wait for the promise to resolve into a response, use the `await` keyword. This blocks your handler code from executing until the promise is resolved into an object that contains the

response. If you don't need to use the data from the response in your code, you can return the promise directly to the runtime.

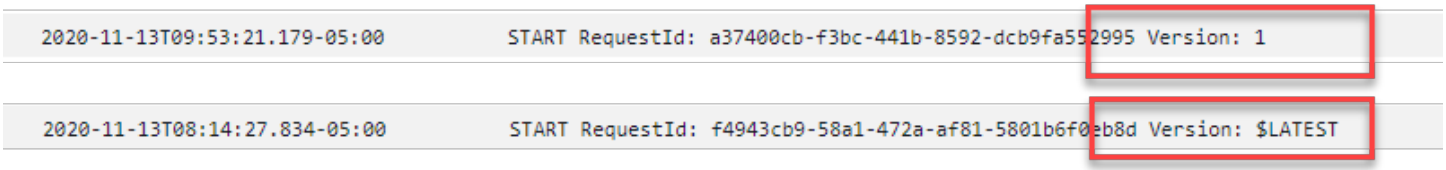
Some libraries don't return promises but can be wrapped in code that does. For more information, see Define Lambda function handler in Node.js.

Lambda: Running an unintended function version or alias

Issue: *Function version or alias not invoked*

When you publish new Lambda functions in the console or using AWS SAM, the latest code version is represented by $LATEST. By default, invocations that don't specify a version or alias automatically targets the $LATEST version of your function code.

If you use specific function versions or aliases, these are immutable published versions of a function in addition to $LATEST. When troubleshooting these functions, first determine that the caller has invoked the intended version or alias. You can do this by checking your function logs. The version of the function that was invoked is always shown in the START log line:

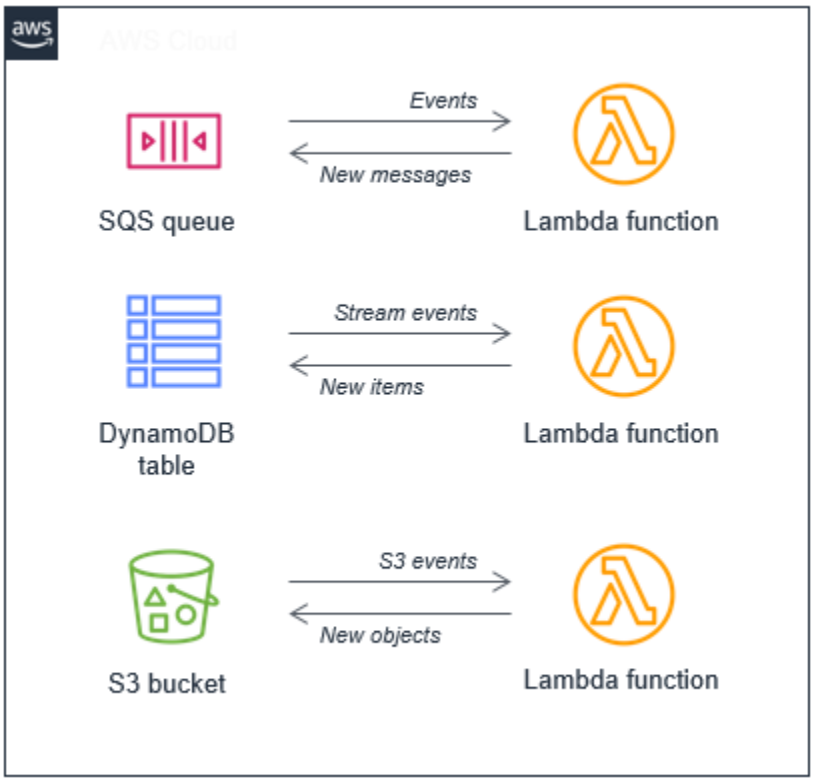


Lambda: Detecting infinite loops

Issue: *Infinite loop patterns related to Lambda functions*

There are two types of infinite loops in Lambda functions. The first is within the function itself, caused by a loop that never exits. The invocation ends only when the function times out. You can identify these by monitoring timeouts, and then fixing the looping behavior.

The second type of loop is between Lambda functions and other AWS resources. These occur when an event from a resource like an S3 bucket invokes a Lambda function, which then interacts with the same source resource to trigger another event. This invokes the function again, which creates another interaction with the same S3 bucket, and so on. These types of loops can be caused by a number of different AWS event sources, including Amazon SQS queues and DynamoDB tables. You can use recursive loop detection to identify these patterns.



You can avoid these loops by ensuring that Lambda functions write to resources that are not the same as the consuming resource. If you must publish data back to the consuming resource, ensure that the new data doesn't trigger the same event. Alternatively, use event filtering. For example, here are two proposed solutions to infinite loops with S3 and DynamoDB resources:

- If you write back to the same S3 bucket, use a different prefix or suffix from the event trigger.
- If you write items to the same DynamoDB table, include an attribute that a consuming Lambda function can filter on. If Lambda finds the attribute, it will not result in another invocation.

General: Downstream service unavailability

Issue: *Downstream services that your Lambda function relies on are unavailable*

For Lambda functions that call out to third-party endpoints or other downstream resources, ensure that they can handle service errors and timeouts. These downstream resources can have variable response times, or become unavailable due to service disruptions. Depending upon the implementation, these downstream errors may appear as Lambda timeouts or exceptions if the service's error response is not handled within the function code.

Anytime a function depends on a downstream service, such as an API call, implement appropriate error handling and retry logic. For critical services, the Lambda function should publish metrics or logs to CloudWatch. For example, if a third-party payment API becomes unavailable, your Lambda function can log this information. You can then set up CloudWatch alarms to send notifications related to these errors.

Since Lambda can scale quickly, non-serverless downstream services may struggle to handle spikes in traffic. There are three common approaches to handling this:

- **Caching** – Consider caching the result of values returned by third-party services if they don't change frequently. You can store these values in global variable in your function, or another service. For example, the results for a product list query from an Amazon RDS instance could be saved for a period of time within the function to prevent redundant queries.
- **Queuing** – When saving or updating data, add an Amazon SQS queue between the Lambda function and the resource. The queue durably persists data while the downstream service processes messages.
- **Proxies** – Where long-lived connections are typically used, such as for Amazon RDS instances, use a proxy layer to pool and reuse those connections. For relational databases, Amazon RDS Proxy is a service designed to help improve scalability and resiliency in Lambda-based applications.

AWS SDK: Versions and updates

Issue: *The AWS SDK included on the runtime is not the latest version*

Issue: *The AWS SDK included on the runtime updates automatically*

Runtimes for interpreted languages include a version of the AWS SDK. Lambda periodically updates these runtimes to use the latest SDK version. To find the version of the SDK that's included in your runtime, see the following sections:

- Runtime included SDK versions (Node.js)
- Runtime included SDK versions (Python)
- Runtime included SDK versions (Ruby)

To use a newer version of the AWS SDK, or to lock your functions to a specific version, you can bundle the library with your function code, or create a Lambda layer. For details on creating a deployment package with dependencies, see the following topics:

Node.js

- Deploy Node.js Lambda functions with .zip file archives

Python

- Working with .zip file archives for Python Lambda functions

Ruby

- Deploy Ruby Lambda functions with .zip file archives

Java

- Deploy Java Lambda functions with .zip or JAR file archives

Go

- Deploy Go Lambda functions with .zip file archives

C#

- Build and deploy C# Lambda functions with .zip file archives

PowerShell

- Deploy PowerShell Lambda functions with .zip file archives

Python: Libraries load incorrectly

Issue: (Python) *Some libraries don't load correctly from the deployment package*

Libraries with extension modules written in C or C++ must be compiled in an environment with the same processor architecture as Lambda (Amazon Linux). For more information, see Working with .zip file archives for Python Lambda functions.

Java: Your function takes longer to process events after updating to Java 17 from Java 11

Issue: (Java) *Your function takes longer to process events after updating to Java 17 from Java 11*

Tune your compiler using the `JAVA_TOOL_OPTIONS` parameter. Lambda runtimes for Java 17 and later Java versions change the default compiler options. The change improves cold start times

for short-lived functions, but the previous behavior is better suited to computationally intensive, longer-running functions. Set `JAVA_TOOL_OPTIONS` to `-XX:-TieredCompilation` to revert to the Java 11 behavior. For more information about the `JAVA_TOOL_OPTIONS` parameter, see the section called "Understanding the `JAVA_TOOL_OPTIONS` environment variable".

Kafka: Error handling and retry configuration issues

Issue: *Kafka event source mapping fails to configure retry settings or on-failure destinations*

Kafka retry configurations and on-failure destinations are only available for event source mappings with provisioned mode enabled. Ensure that you have configured `MinimumPollers` in your `ProvisionedPollerConfig` before attempting to set retry configurations.

Common configuration errors:

- **Infinite retries with bisect batch** – You cannot enable `BisectBatchOnFunctionError` when `MaximumRetryAttempts` is set to -1 (infinite). Set a finite retry limit or disable bisect batch.
- **Same topic recursion** – The Kafka on-failure destination topic cannot be the same as any of your source topics. Choose a different topic name for your dead letter topic.
- **Invalid Kafka destination format** – Use the `kafka://<topic-name>` format when specifying a Kafka topic as an on-failure destination.
- **kafka:WriteData permission issues** – Ensure your execution role has `kafka-cluster:WriteData` permissions for the destination topic. Topic doesn't exist timeout exceptions or write API throttling issues may require increasing the account limits.

Troubleshoot event source mapping issues in Lambda

Issues in Lambda that relate to an event source mapping can be more complex because they involve debugging across multiple services. Moreover, event source behavior can differ based on the exact event source used. This section lists common issues that involve event source mappings, and provides guidance on how to identify and troubleshoot them.

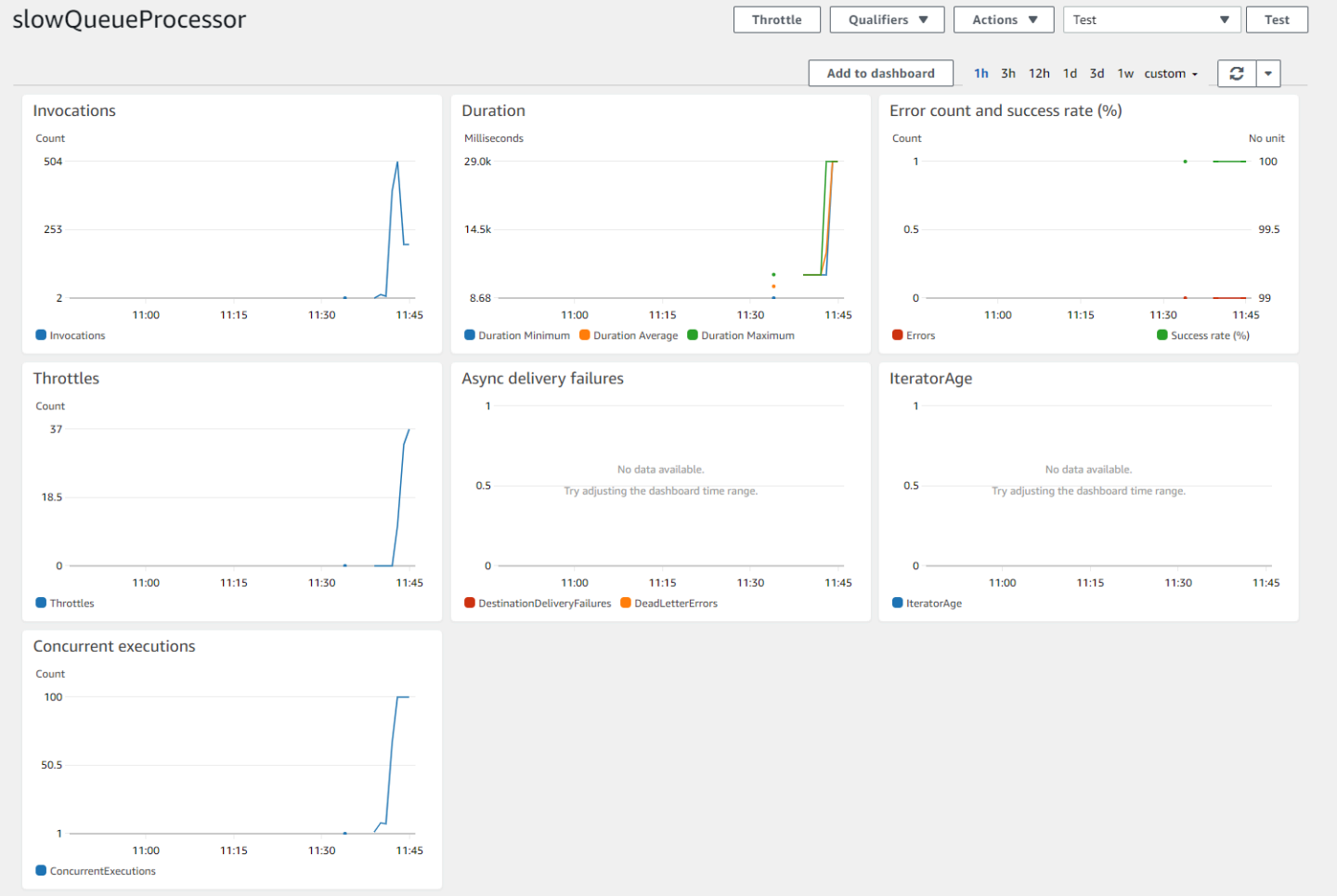
> **Note**
>
> This section uses an Amazon SQS event source for illustration, but the principles apply to other event source mappings that queue messages for Lambda functions.

Identifying and managing throttling

In Lambda, throttling occurs when you reach your function's or account's concurrency limit. Consider the following example, where there is a Lambda function that reads messages from an Amazon SQS queue. This Lambda function simulates 30 second invocations, and has a batch size of 1. This means that the function processes only 1 message every 30 seconds:

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)

}
```

With such a long invocation time, messages begin arriving in the queue more rapidly than they are processed. If your account's unreserved concurrency is 100, Lambda scales up to 100 concurrent executions, and then throttling occurs. You can see this pattern in the CloudWatch metrics for the function:



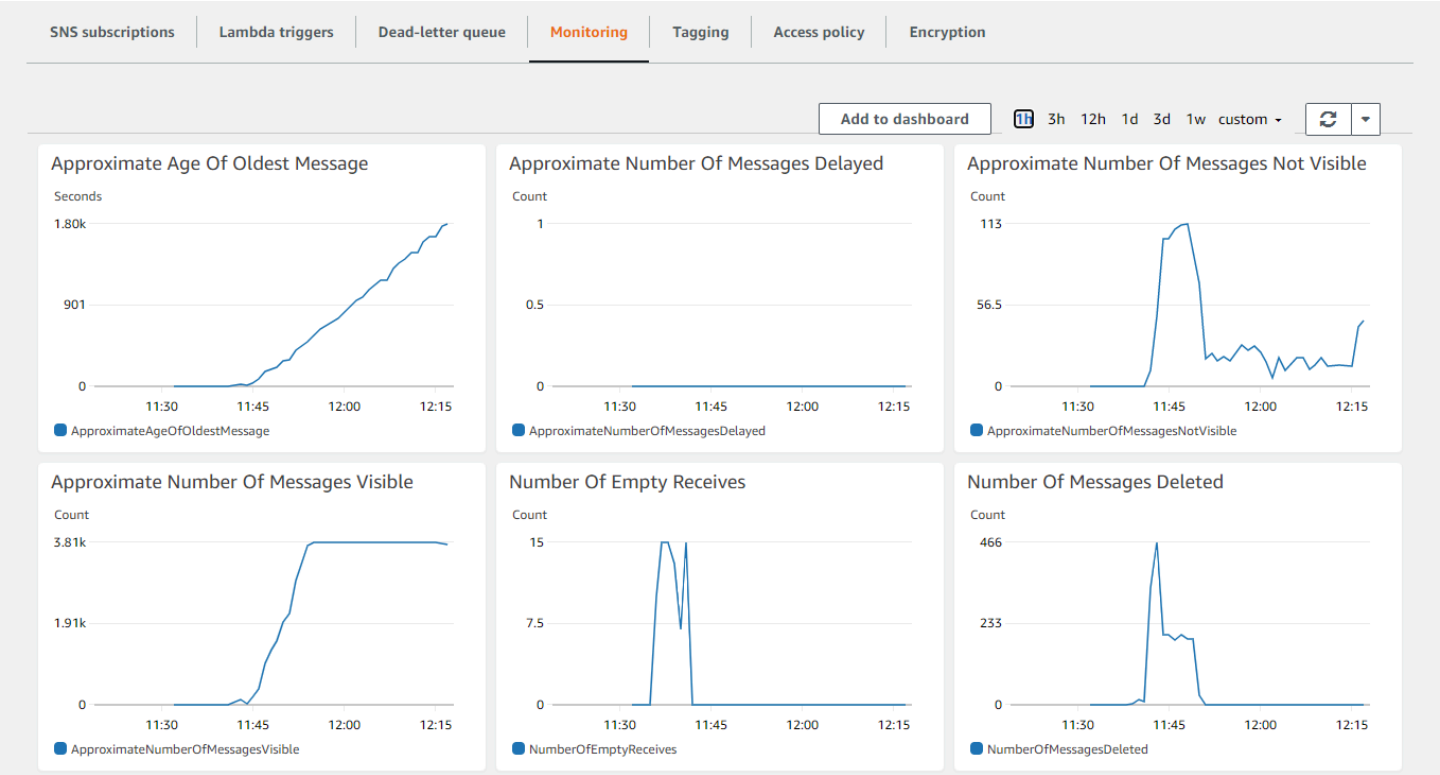
CloudWatch metrics for the function show no errors, but the **Concurrent executions** chart shows that the maximum concurrency of 100 is reached. As a result, the **Throttles** chart shows the throttling in place.

You can detect throttling with CloudWatch alarms, and setting an alarm anytime the throttling metric for a function is greater than 0. After you've identified the throttling issue, you have a few options for resolution:

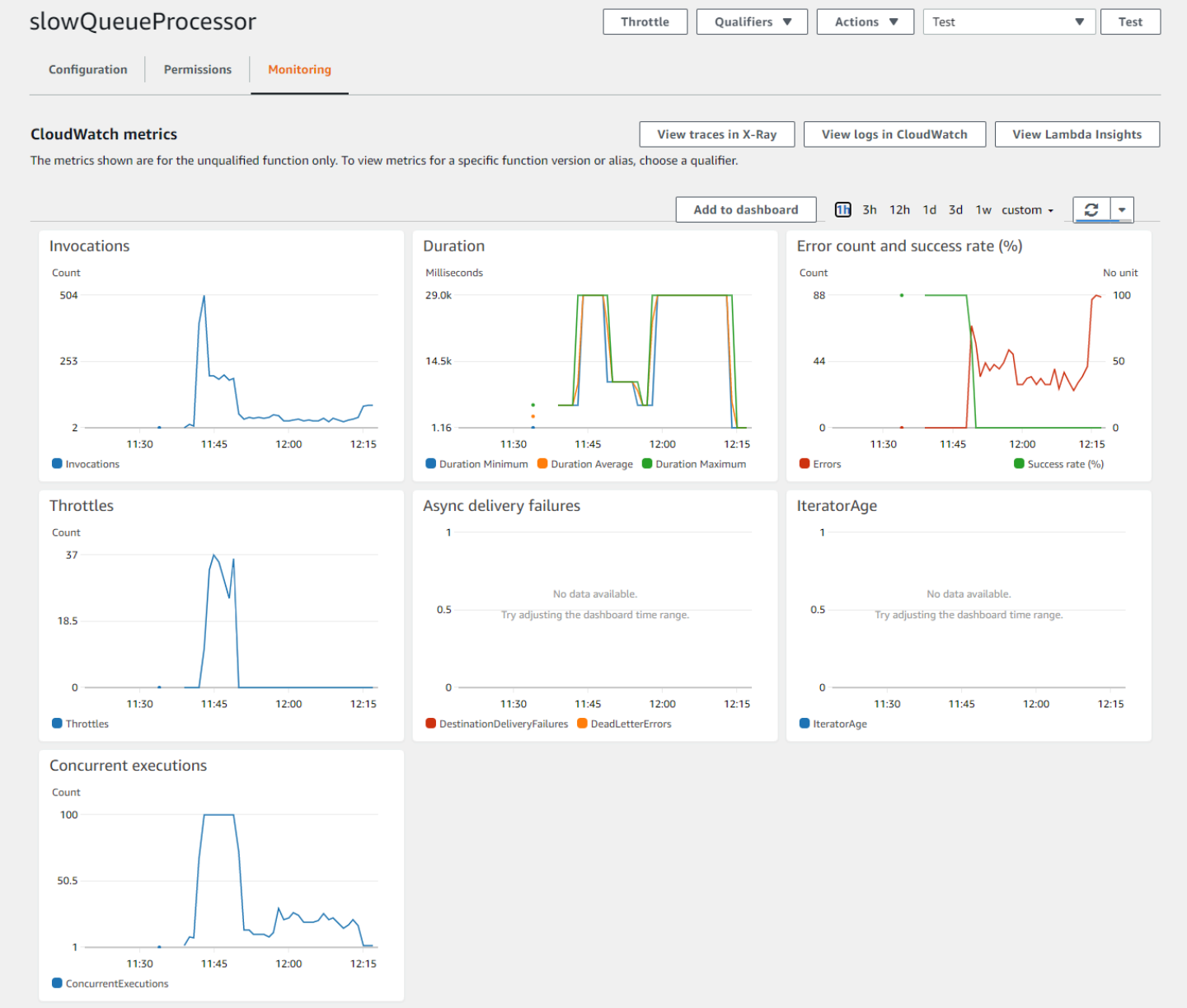
- Request a concurrency increase from AWS Support in this Region.
- Identify performance issues in the function to improve the speed of processing and therefore improve throughput.
- Increase the batch size of the function, so more messages are processed by each invocation.

Errors in the processing function

If the processing function throws errors, Lambda returns the messages to the SQS queue. Lambda prevents your function from scaling to prevent errors at scale. The following SQS metrics in CloudWatch indicate an issue with queue processing:

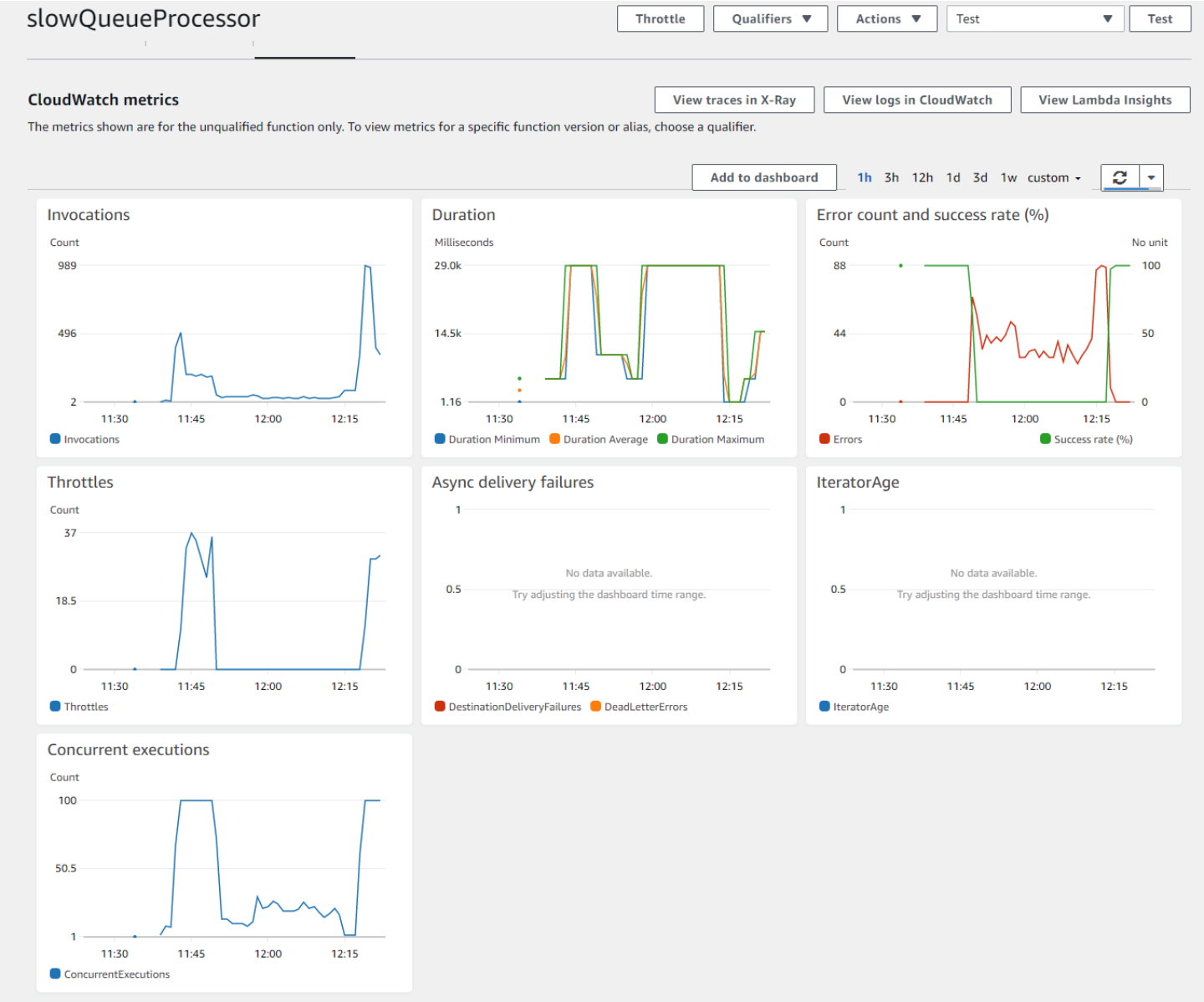


In particular, both the age of the oldest message and the number of messages visible are increasing, while no messages are deleted. The queue continues to grow but messages are not being processed. The CloudWatch metrics for the processing Lambda function also indicate that there is a problem:



The **Error count** metric is non-zero and growing, while **Concurrent executions** have reduced and throttling has stopped. This shows that Lambda has stopped scaling up your function due to errors. The CloudWatch logs for the function provide details of the type of error.

You can resolve this issue by identifying the function causing the error, then finding and resolving the error. After you fix the error and deploy the new function code, the CloudWatch metrics should show the processing recover:



Here, the **Error count** metric drops to zero and the **Success rate** metric returns to 100%. Lambda starts scaling up the function again, as shown in the **Concurrent executions** graph.

Identifying and handling backpressure

If an event producer consistently generates messages for an SQS queue faster than a Lambda function can process them, backpressure occurs. In this case, SQS monitoring should show the age of the oldest message growing linearly, along with the approximate number of messages visible. You can detect backpressure in queues using CloudWatch alarms.

The steps to resolve backpressure depend on your workload. If the primary goal is to increase processing capability and throughput by the Lambda function, you have a few options:

- Request a concurrency increase in the specific Region from AWS Support.
- Increase the batch size of the function, so more messages are processed by each invocation.

Troubleshoot networking issues in Lambda

By default, Lambda runs your functions in an internal virtual private cloud (VPC) with connectivity to AWS services and the internet. To access local network resources, you can configure your function to connect to a VPC in your account. When you use this feature, you manage the function's internet access and network connectivity with Amazon Virtual Private Cloud (Amazon VPC) resources.

Network connectivity errors can result from issues with your VPC's routing configuration, security group rules, AWS Identity and Access Management (IAM) role permissions, or network address translation (NAT), or from the availability of resources such as IP addresses or network interfaces. Depending on the issue, you might see a specific error or timeout if a request can't reach its destination.

Topics

- VPC: Function loses internet access or times out
- VPC: TCP or UDP connection intermittently fails
- VPC: Function needs access to AWS services without using the internet
- VPC: Elastic network interface limit reached
- EC2: Elastic network interface with type of "lambda"
- DNS: Fail to connect to hosts with UNKNOWNHOSTEXCEPTION

VPC: Function loses internet access or times out

Issue: *Your Lambda function loses internet access after connecting to a VPC.*

Error: *Error: connect ETIMEDOUT 176.32.98.189:443*

Error: *Error: Task timed out after 10.00 seconds*

Error: *ReadTimeoutError: Read timed out. (read timeout=15)*

When you connect a function to a VPC, all outbound requests go through the VPC. To connect to the internet, configure your VPC to send outbound traffic from the function's subnet to a NAT

gateway in a public subnet. For more information and sample VPC configurations, see the section called “Internet access for VPC functions”.

If some of your TCP connections are timing out, see the section called “VPC: TCP or UDP connection intermittently fails” if your subnet is using a network access control list (NACL). Otherwise, this is likely due to packet fragmentation. Lambda functions cannot handle incoming fragmented TCP requests, since Lambda does not support IP fragmentation for TCP or ICMP.

VPC: TCP or UDP connection intermittently fails

> **Note**
>
> This issue applies only if your subnet uses a network access control list (ACL). Network ACLs aren't required for Lambda to connect to your subnets.

Issue: *Lambda intermittently loses connection to your VPC subnets, which you have configured a network access control list (ACL) for.*

For VPC-enabled Lambda functions, AWS creates hyperplane ENIs in the customer's account, and uses ephemeral ports 1024 to 65535 to connect Lambda to the customer's VPC. If you use network ACLs in the target subnet, you must allow the port range 1024 to 65535 for both TCP and UDP. Not allowing this full port range can cause intermittent connection failures.

VPC: Function needs access to AWS services without using the internet

Issue: *Your Lambda function needs access to AWS services without using the internet.*

To connect a function to AWS services from a private subnet with no internet access, use VPC endpoints.

VPC: Elastic network interface limit reached

Error: *ENILimitReachedException: The elastic network interface limit was reached for the function's VPC.*

When you connect a Lambda function to a VPC, Lambda creates an elastic network interface for each combination of subnet and security group attached to the function. The default service quota is 250 network interfaces per VPC. To request a quota increase, use the Service Quotas console.

EC2: Elastic network interface with type of "lambda"

Error Code: *Client.OperationNotPermitted*

Error message: *The security group can not be modified for this type of interface*

You will receive this error if you attempt to modify an elastic network interface (ENI) that is managed by Lambda. The `ModifyNetworkInterfaceAttribute` is not included in the Lambda API for update operations on elastic network interfaces created by Lambda.

DNS: Fail to connect to hosts with UNKNOWNHOSTEXCEPTION

Error Message: *UNKNOWNHOSTEXCEPTION*

Lambda functions support a maximum of 20 concurrent TCP connections for DNS resolution. Your function may be exhausting that limit. Most common DNS requests are done over UDP. If your function is only making UDP DNS connections, this is unlikely to be your issue. This error is commonly thrown due to misconfiguration or degraded infrastructure, so before examining your DNS traffic in depth, confirm that your DNS infrastructure is properly configured and healthy and that your Lambda function is referring to a host specified in DNS.

If you diagnose your issue as related to the TCP connection maximum, note that you cannot request an increase to this limit. If your Lambda function is falling back to TCP DNS because of large DNS payloads, confirm that your solution is using libraries that support EDNS. For more information about EDNS, see the RFC 6891 standard. If your DNS payloads consistently exceed EDNS max sizes, your solution may still exhaust the TCP DNS limit.