**AWS Lambda Concepts**

AWS Lambda lets you run functions in a serverless environment to process events in the language of your choice. Each instance of your function runs in an isolated execution context and processes one event at a time. When it finishes processing the event, it returns a response and Lambda sends it another event. Lambda automatically scales up the number of instances of your function to handle high numbers of events.

* **Function** – A script or program that runs in AWS Lambda. Lambda passes invocation events to your function. The function processes an event and returns a response.
* **Runtimes** – Lambda runtimes allow functions in different languages to run in the same base execution environment. You configure your function to use a runtime that matches your programming language. The runtime sits in-between the Lambda service and your function code, relaying invocation events, context information, and responses between the two. You can use runtimes provided by Lambda, or build your own.
* **Layers** – Lambda layers are a distribution mechanism for libraries, custom runtimes, and other function dependencies. Layers let you manage your in-development function code independently from the unchanging code and resources that it uses. You can configure your function to use layers that you create, layers provided by AWS, or layers from other AWS customers.
* **Event source** – An AWS service, such as Amazon SNS, or a custom service, that triggers your function and executes its logic. For more information, see [AWS Lambda Event Source Mapping](https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html).
* **Downstream resources** – An AWS service, such as DynamoDB tables or Amazon S3 buckets, that your Lambda function calls once it is triggered.
* **Log streams** – While Lambda automatically monitors your function invocations and reports metrics to CloudWatch, you can annotate your function code with custom logging statements that allow you to analyze the execution flow and performance of your Lambda function to ensure it's working properly.
* **AWS SAM** – A model to define [serverless applications](https://aws.amazon.com/serverless). AWS SAM is natively supported by AWS CloudFormation and defines simplified syntax for expressing serverless resources..

The following managed policies provide permissions that are required to use Lambda features:

* **AWSLambdaBasicExecutionRole**– Permission to upload logs to CloudWatch.
* **AWSLambdaKinesisExecutionRole**– Permission to read events from an Amazon Kinesis data stream or consumer.
* **AWSLambdaDynamoDBExecutionRole**– Permission to read records from an Amazon DynamoDB stream.
* **AWSLambdaSQSQueueExecutionRole** – Permission to read a message from an Amazon Simple Queue Service (Amazon SQS) queue.
* **AWSLambdaVPCAccessExecutionRole** – Permission to manage elastic network interfaces to connect your function to a VPC.
* **AWSXrayWriteOnlyAccess**– Permission to upload trace data to X-Ray.

**Configuring Lambda Functions**

1 - AWS Lambda does not support connecting to resources within Dedicated Tenancy VPCs.

2 - To keep secrets out of your function code, store them in the function's configuration and read them from the execution environment during initialization. [Environment variables](https://docs.aws.amazon.com/lambda/latest/dg/env_variables.html) are always encrypted at rest, and can be encrypted in transit as well. Use environment variables to make your function code portable by removing connection strings, passwords, and endpoints for external resources.

3- To use your Lambda function with AWS resources in an Amazon VPC, configure it with security groups and subnets to [create a VPC connection](https://docs.aws.amazon.com/lambda/latest/dg/vpc.html). Lambda uses [elastic network interfaces](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_ElasticNetworkInterfaces.html) (ENIs) to create the connection, so you need to ensure that your account has enough ENI capacity (IP addresses) to handle the number of connections made as your function scales up under load.

4 – Lambda function settings –

**Function Settings**

* **Code**
* **Runtime** – The [Lambda runtime](https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html) that executes your function.
* **Handler** – The method that the runtime executes when your function is invoked.
* **Environment variables** – Key-value pairs that Lambda sets in the execution environment.[Use environment variables](https://docs.aws.amazon.com/lambda/latest/dg/env_variables.html) to extend your function's configuration outside of code.
* **Tags** – Key-value pairs that Lambda attaches to your function resource.
* **Execution role** – The [IAM role](https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html) that AWS Lambda assumes when it executes your function.
* **Description** – A description of the function.
* **Memory** – The amount of memory available to the function during execution. Choose an amount [between 128 MB and 3,008 MB](https://docs.aws.amazon.com/lambda/latest/dg/limits.html) in 64 MB increments.Lambda allocates CPU power linearly in proportion to the amount of memory configured. At 1,792 MB, a function has the equivalent of 1 full vCPU (one vCPU-second of credits per second).
* **Timeout** – The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds.
* **Virtual private cloud (VPC)** – If your function needs network access to resources that are not available over the internet, [configure it to connect to a VPC](https://docs.aws.amazon.com/lambda/latest/dg/vpc.html).
* **Dead letter queue (DLQ)** – If your function is invoked asynchronously, [choose a queue or topic](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq) to receive failed invocations.
* **Enable active tracing** – Sample incoming requests and [trace sampled requests with AWS X-Ray](https://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html).
* **Concurrency** – [Reserve concurrency for a function](https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html) to set the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level.Reserved concurrency applies to the entire function, including all versions and aliases.

5 - Function settings can only be changed on the unpublished version of a function. When you publish a version, code and most settings are locked to ensure a consistent experience for users of that version.

6 - **Account Level Concurrent Execution Limit** - By default, AWS Lambda limits the total concurrent executions across all functions within a given region to 1000. This limit can be increased by requesting AWS.

7 – **Function Level Concurrent Execution Limit -** the concurrent execution limit is enforced against the sum of the concurrent executions of all functions. The shared concurrent execution pool is referred to as the unreserved concurrency allocation. If you haven’t set up any function-level concurrency limit, then the unreserved concurrency limit is the same as the account level concurrency limit. Any increases to the account level limit have a corresponding increase in the unreserved concurrency limit.

You can optionally set the concurrent execution limit for a function. You may choose to do this for a few reasons:

* The default behavior means a surge of concurrent executions in one function prevents the function you have isolated with an execution limit from getting throttled. By setting a concurrent execution limit on a function, you are reserving the specified concurrent execution value for that function.
* Functions scale automatically based on incoming request rate, but not all resources in your architecture may be able to do so. For example, relational databases have limits on how many concurrent connections they can handle. You can set the concurrent execution limit for a function to align with the values of its downstream resources support.
* If your function connects to VPC based resources, you must make sure your subnets have adequate address capacity to support the ENI scaling requirements of your function. You can estimate the approximate ENI capacity with the following formula:

Concurrent executions \* (Memory in GB / 3 GB)

Where:

* + **Concurrent execution** – This is the projected concurrency of your workload. Use the information in [Understanding Scaling Behavior](https://docs.aws.amazon.com/lambda/latest/dg/scaling.html) to determine this value.
  + **Memory in GB** – The amount of memory you configured for your Lambda function.

You can set the concurrent execution limit for a function to match the subnet size limits you have.

**Note** If you need a function to stop processing any invocations, you can choose to set the concurrency to 0 and throttle all incoming executions.

**Note** Concurrency limits can only be set at the function level, not for individual versions. All invocations to all versions and aliases of a given function will accrue towards the function limit.

**8 - Reserved vs. Unreserved Concurrency Limits -** If you set the concurrent execution limit for a function, the value is deducted from the unreserved concurrency pool. For example, if your account's concurrent execution limit is 1000 and you have 10 functions, you can specify a limit on one function at 200 and another function at 100. The remaining 700 will be shared among the other 8 functions.

**Note -** AWS Lambda will keep the unreserved concurrency pool at a minimum of 100 concurrent executions, so that functions that do not have specific limits set can still process requests. So, in practice, if your total account limit is 1000, you are limited to allocating 900 to individual functions.

**9 - Throttling Behavior -** On reaching the concurrency limit associated with a function, any further invocation requests to that function are throttled, i.e. the invocation doesn't execute your function. Each throttled invocation increases the Amazon CloudWatch Throttles metric for the function. AWS Lambda handles throttled invocation requests differently, depending on their source:

* **Event sources that aren't stream-based:** Some of these event sources invoke a Lambda function synchronously, and others invoke it asynchronously. Handling is different for each:
  + **Synchronous invocation:** If the function is invoked synchronously and is throttled, Lambda returns a 429 error and the invoking service is responsible for retries. TheThrottledReason error code explains whether you ran into a function level throttle (if specified) or an account level throttle (see note below). Each service may have its own retry policy.
  + **Asynchronous invocation:** If your Lambda function is invoked asynchronously and is throttled, AWS Lambda automatically retries the throttled event for up to six hours, with delays between retries.
* **Poll-based event sources that are also stream-based:** such as [Amazon Kinesis](https://docs.aws.amazon.com/kinesis/latest/dev/) or [Amazon DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/), AWS Lambda polls your stream and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until the time the data expires. This time period can be up to seven days for Amazon Kinesis. The throttled request is treated as blocking per shard, and Lambda doesn't read any new records from the shard until the throttled batch of records either expires or succeeds. If there is more than one shard in the stream, Lambda continues invoking on the non-throttled shards until one gets through.
* **Poll-based event sources that are not stream-based:** such as [Amazon Simple Queue Service](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-getting-started.html), AWS Lambda polls your queue and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until it is successfully invoked (in which case the message is automatically deleted from the queue) or until the MessageRetentionPeriod set for the queue expires.

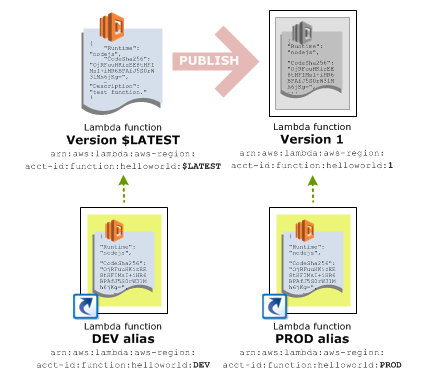
**10 - Monitoring Your Concurrency Usage -** To understand your concurrent execution usage, AWS Lambda provides the following metrics:

* **ConcurrentExecutions**: This shows you the concurrent executions at an account level, and for any function with a custom concurrency limit.
* **UnreservedConcurrentExecutions**: This shows you the total concurrent executions for functions assigned to the default unreserved concurrency pool.

11 - The first time you create or update Lambda functions that use environment variables in a region, a default service key is created for you automatically within AWS KMS. This key is used to encrypt environment variables. However, you can use your own AWS KMS key and choose it instead of the default key.

12 - **Storing Sensitive Information** - when you deploy your Lambda function, all the environment variables you've specified are encrypted by default after, but not during, the deployment process. They are then decrypted automatically by AWS Lambda when the function is invoked. If you need to store sensitive information in an environment variable, we strongly suggest you encrypt that information before deploying your Lambda function.

13 - **Alias** - Conceptually, an AWS Lambda alias is a pointer to a specific Lambda function version. It's also a resource similar to a Lambda function, and each alias has a unique ARN. Each alias maintains an ARN for the function version to which it points. An alias can only point to a function version, not to another alias. Unlike versions, aliases can be modified. You can update aliases to point to different versions of functions



AWS Lambda aliases enable the following use cases:

**Easier support for promotion of new versions of Lambda functions and rollback when needed –** After initially creating a Lambda function (the $LATEST version), you can publish a version 1 of it. By creating an alias named PROD that points to version 1, you can now use the PROD alias to invoke version 1 of the Lambda function.

Now, you can update the code (the $LATEST version) with all of your improvements, and then publish another stable and improved version (version 2). You can promote version 2 to production by remapping the PROD alias so that it points to version 2. If you find something wrong, you can easily roll back the production version to version 1 by remapping the PROD alias so that it points to version 1.

**Simplify management of event source mappings –** Instead of using Amazon Resource Names (ARNs) for Lambda function in event source mappings, you can use an alias ARN. This approach means that you don't need to update your event source mappings when you promote a new version or roll back to a previous version.

You can access the function using either the function ARN or the alias ARN.

arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST

When using any of the alias ARNs, you are using a qualified ARN. Each alias ARN has an alias name suffix.

arn:aws:lambda:aws-region:acct-id:function:helloworld:PROD

arn:aws:lambda:aws-region:acct-id:function:helloworld:BETA

arn:aws:lambda:aws-region:acct-id:function:helloworld:DEV

**14 – Versioning -** When you create a Lambda function, there is only one version the $LATEST version. You can refer to this function using its Amazon Resource Name (ARN). There are two ARNs associated with this initial version:

* **Qualified ARN** – The function ARN with the version suffix.

**arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST**

* **Unqualified ARN** – The function ARN without the version suffix. The unqualified ARN has its own resource policies.

**arn:aws:lambda:aws-region:acct-id:function:helloworld**

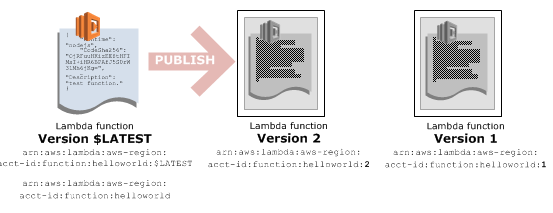
Note - Unless you choose to publish versions, the $LATEST function version is the only Lambda function version that you have. You can use either the qualified or unqualified ARN in your event source mapping to invoke the $LATEST version.

**15 - Publishing an AWS Lambda Function Version -** When you publish a version, AWS Lambda makes a snapshot copy of the Lambda function code (and configuration) in the $LATEST version. A published version is immutable. That is, you can't change the code or configuration information. The new version has a unique ARN that includes a version number suffix as shown following.

You can publish a version by using any of the following methods:

* **Publish a version explicitly**
* **Publish a version at the time you create or update a Lambda function (recommended).**

You can publish multiple versions of a Lambda function. Each time you publish a version, AWS Lambda copies $LATEST version (code and configuration information) to create a new version. When you publish additional versions, AWS Lambda assigns a monotonically increasing sequence number for versioning, even if the function was deleted and recreated. Version numbers are never reused, even for a function that has been deleted and recreated. This approach means that the consumer of a function version can depend on the executable of that version to never change (except if it's deleted).



**16 - Deleting a Lambda Function and a Specific Version -** With versioning, you have the following choices:

* **Delete a specific version** – If there are aliases that depend on this version, the request fails
* **Delete the entire Lambda function (all of its versions and aliases)**

**17 -** When you use a [resource-based policy](https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html) to give a service, resource, or account access to your function, the scope of that permission depends on if you applied it to a version, to an alias, or to a function. e.g If you use a qualified function name (such as helloworld:1), the permission is valid for invoking the helloworld function version 1 only using its qualified ARN (using any other ARNs results in a permission error).

**18 - Traffic Shifting Using Aliases -** you can implement the routing-config parameter of the Lambda alias that allows you to point to two different versions of the Lambda function and dictate what percentage of incoming traffic is sent to each version.

You can point an alias to a maximum of two Lambda function versions. In addition:

* Both versions must have the same IAM execution role.
* Both versions must have the same [AWS Lambda Function Dead Letter Queues](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq) configuration, or no DLQ configuration.
* When pointing an alias to more than one version, the alias cannot point to $LATEST.

**Determining Which Version Has Been Invoked -** there are two ways to determine which Lambda function version has been invoked:

* CloudWatch Logs
* Response payload (synchronous invocations - x-amz-executed-version header)

**19 -** A function can use up to 5 layers at a time. The total unzipped size of the function and all layers can't exceed the unzipped deployment package size limit of 250 MB. For more information, see [AWS Lambda Limits](https://docs.aws.amazon.com/lambda/latest/dg/limits.html).

**20 - Lambda Function to Access Resources in an Amazon VPC -** AWS Lambda runs your function code securely within its own isolated VPC by default. However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration information that includes private subnet IDs and security group IDs. AWS Lambda uses this information to set up elastic network interfaces [(ENIs)](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_ElasticNetworkInterfaces.html) (IP addresses) that enable your function to connect securely to other resources within your private VPC.

Lambda functions cannot connect directly to a VPC with [dedicated instance tenancy](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-instance.html). To connect to resources in a dedicated VPC, [peer it to a second VPC with default tenancy](https://aws.amazon.com/premiumsupport/knowledge-center/lambda-dedicated-vpc/).

**Internet Access for Lambda Functions -** AWS Lambda uses the VPC information you provide to set up [ENIs](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_ElasticNetworkInterfaces.html) that allow your Lambda function to access VPC resources. Each ENI is assigned a private IP address from the IP address range within the subnets you specify. Functions that are connected to a VPC do not have public IP addresses or internet access by default.

Internet access from a private subnet requires network address translation (NAT). You can give your function access to the internet by adding a NAT gateway or NAT instance to your VPC.

**Note** - Several services offer [VPC Endpoints](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-endpoints.html). You can use VPC endpoints to connect to AWS services from within a VPC without internet access.

**21 - Guidelines for Setting Up VPC-Enabled Lambda Functions**

Your Lambda function automatically scales based on the number of events it processes. The following are general guidelines for setting up VPC-enabled Lambda functions to support the scaling behavior.

* If your Lambda function accesses a VPC, you must make sure that your VPC has sufficient ENI capacity to support the scale requirements of your Lambda function. You can use the following formula to approximately determine the ENI requirements.

**Projected peak concurrent executions \* (Memory in GB / 3GB)**

Where: Memory – The amount of memory you configured for your Lambda function.

* The subnets you specify should have sufficient available IP addresses to match the number of ENIs.

We also recommend that you specify at least one subnet in each Availability Zone in your Lambda function configuration. By specifying subnets in each of the Availability Zones, your Lambda function can run in another Availability Zone if one goes down or runs out of IP addresses.

If your VPC does not have sufficient ENIs or subnet IPs, your Lambda function will not scale as requests increase, and you will see an increase in invocation errors with EC2 error types like EC2ThrottledException.

**Invoking Lambda Functions**

**22 -** When you invoke a function, you can choose to invoke it **synchronously or asynchronously.** With [**synchronous invocation**](https://docs.aws.amazon.com/lambda/latest/dg/invocation-sync.html)**,** you wait for the function to process the event and return a response. With [**asynchronous**](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html)**invocation**, Lambda queues the event for processing and returns a response immediately. For asynchronous invocation, Lambda handles retries and can send failed events to a [dead-letter queue](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq).

**23 -** Lambda manages the function's asynchronous invocation queue and attempts to retry failed events automatically. If the function returns an error, Lambda attempts to run it two more times, with a one minute wait between the first two attempts, and two minutes between the second and third. Function errors include errors returned by the function's code, and errors returned by the function's runtime, such as timeouts. If all 3 attempts fail, Lambda sends the event to a [**dead-letter queue**](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq)**,** if configured.

**24 - AWS Lambda Function Dead Letter Queues -** When all three attempts to process an asynchronous invocation fail, Lambda can send the event to an Amazon SQS queue or an Amazon SNS topic (they are called DLQ). Configure your function with a dead-letter queue to save these events for further processing.

* [**Amazon SQS queue**](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-create-queue.html) – A queue holds failed events until they are retrieved. You can retrieve events manually, or [configure Lambda to read from the queue](https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html) and invoke a function.
* [**Amazon SNS topic**](https://docs.aws.amazon.com/sns/latest/gsg/CreateTopic.html) – A topic relays failed events to one or more destinations. You can configure a topic to send events to an email address, a Lambda function, or an HTTP endpoint.

**25** - If Lambda can't send a message to the dead-letter queue, it deletes the event and emits the [DeadLetterErrors](https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-metrics.html) metric. This can happen due to lack of permissions, or if the total size of the message exceeds the limit (allowed in the DLQ.) for the target queue or topic.

**26 - Services That Lambda Reads Events From**

* [Amazon Kinesis](https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html) (stream based)
* [Amazon DynamoDB](https://docs.aws.amazon.com/lambda/latest/dg/with-ddb.html) (stream based)
* [Amazon Simple Queue Service](https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html)

**27 -** Estimate the capacity (Concurrent executions) used by your function

For example, suppose that the Lambda function takes on average three seconds and Amazon S3 publishes 10 events per second. Then, you will have 30 concurrent executions of your Lambda function.

**Formula:** Estimate the capacity used by your function = invocations per second \* average execution duration in seconds

**28 –** **Request Rate** - Request rate refers to the rate at which your Lambda function is invoked.

**Forumal**: request rate = number of concurrent executions / function duration

For example, if there are five active shards on a stream (that is, you have five Lambda functions running in parallel) and your Lambda function takes about two seconds, the request rate is 2.5 requests/second.

**29 - Automatic Scaling -** Under sustained load, your function's concurrency bursts to an initial level between 500 and 3000 concurrent executions that varies per region. After the initial burst, the function's capacity increases by an additional 500 concurrent executions each minute until either the load is accommodated, or the total concurrency of all functions in the region hits the limit.

**Note -** If your function is connected to a VPC, the [Amazon VPC network interface limit](https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_vpc) can prevent it from scaling.

**30 - AWS Lambda Invocation Retries -** Function invocation can result in an error for several reasons. Your code might raise an exception, time out, or run out of memory. The runtime executing your code might encounter an error and stop. You might run out concurrency and be throttled.

Lambda handles retries in the following manner, depending on the source of the invocation.

* **Event sources that aren't stream-based –** Some of these event sources are set up to invoke a Lambda function synchronously and others invoke it asynchronously. Accordingly, exceptions are handled as follows:

1. **Synchronous invocation –**Lambda only returns error status codes if there is an issue with the request, function, or permissions that prevents the handler from processing the event. [AWS service triggers](https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html) can retry depending on the service. If you invoke the Lambda function directly from your application, you can choose whether to retry or not.
2. **Asynchronous invocation –** Asynchronous events are queued before being used to invoke the Lambda function. If AWS Lambda is unable to fully process the event, it will automatically retry the invocation twice, with delays between retries. Configure a [dead letter queue](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq) for your function to capture requests that fail all three attempts.

* **Poll-based event sources that are stream-based –** These consist of Kinesis Data Streams or DynamoDB. When a Lambda function invocation fails, AWS Lambda attempts to process the erring batch of records until the time the data expires, which can be up to seven days. The exception is treated as blocking, and AWS Lambda will not read any new records from the shard until the failed batch of records either expires or is processed successfully. This ensures that AWS Lambda processes the stream events in order.
* **Poll-based event sources that are not stream-based –** If you configure an Amazon SQS queue as an event source, AWS Lambda will poll a batch of records in the queue and invoke your Lambda function. If the invocation fails or times out, every message in the batch will be returned to the queue, and each will be available for processing once the [Visibility Timeout](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html) period expires. (Visibility timeouts are a period of time during which Amazon Simple Queue Service prevents other consumers from receiving and processing the message). Once an invocation successfully processes a batch, each message in that batch will be removed from the queue. When a message is not successfully processed, it is either discarded or if you have configured an [Amazon SQS Dead Letter Queue](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-configure-dead-letter-queue.html), the failure information will be directed there for you to analyze.

If you don't require ordered processing of events, the advantage of using Amazon SQS queues is that AWS Lambda will continue to process new messages, regardless of a failed invocation of a previous message. In other words, processing of new messages will not be blocked.

**Lambda Runtimes**

**31 - AWS Lambda Execution Context -** When you create a Lambda function, you specify configuration information, such as the amount of memory and maximum execution time that you want to allow for your Lambda function. When a Lambda function is invoked, AWS Lambda launches an execution context based on the configuration settings you provide. The execution context is a temporary runtime environment that initializes any external dependencies of your Lambda function code, such as database connections or HTTP endpoints. This affords subsequent invocations better performance because there is no need to "cold-start" or initialize those external dependencies, as explained below.

This execution context reuse approach has the following implications:

* Any declarations in your Lambda function code (outside the handler code, see [Programming Model](https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html)) remains initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations.
* Each execution context provides 512 MB of additional disk space in the /tmp directory. The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored.
* Background processes or callbacks initiated by your Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the execution context. You should make sure any background processes or callbacks in your code are complete before the code exits.

**Lambda Applications**

**32 - Common Lambda Application Types and Use Cases -** When building applications on AWS Lambda the core components are Lambda functions and triggers. To illustrate, consider the following scenarios:

* **File processing –** a photo sharing application.
* **Data and analytics –** DynamoDB streams
* **Websites –**host the backend logic on Lambda.
* **Mobile applications –** configure a Lambda function to process the clicks within your custom mobile application.

**33 - Best Practices for Working with AWS Lambda Functions**

**Function Code**

* Separate the Lambda handler (entry point) from your core logic
* Take advantage of Execution Context reuse to improve the performance of your function.
* Use [AWS Lambda Environment Variables](https://docs.aws.amazon.com/lambda/latest/dg/env_variables.html) to pass operational parameters to your function
* Control the dependencies in your function's deployment package
* Minimize your deployment package size to its runtime necessities
* Reduce the time it takes Lambda to unpack deployment packages
* Minimize the complexity of your dependencies
* Avoid using recursive code

**Function Configuration**

* Performance testing your Lambda function
* Load test your Lambda function
* Use most-restrictive permissions when setting IAM policies
* Be familiar with [AWS Lambda Limits](https://docs.aws.amazon.com/lambda/latest/dg/limits.html)
* Delete Lambda functions that you are no longer using
* If you are using Amazon Simple Queue Service as an event source, make sure the value of the function's expected execution time does not exceed the [Visibility Timeout](https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html) value on the queue.

**Alarming and Metrics**

* Use [AWS Lambda Metrics](https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-metrics.html) and [CloudWatch Alarms](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html)
* Leverage your logging library and [AWS Lambda Metrics and Dimensions](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/lam-metricscollected.html) to catch app errors (e.g. ERR, ERROR, WARNING, etc.)

**Stream Event Invokes**

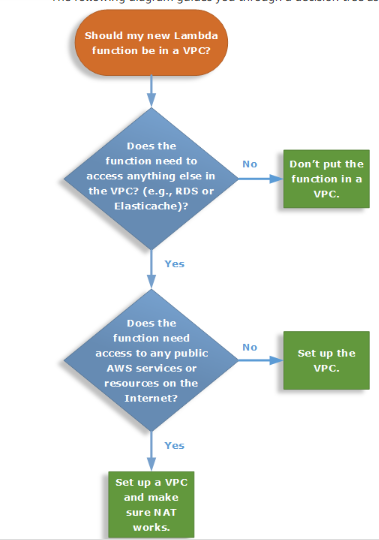
* Test with different batch and record sizes
* Increase Kinesis stream processing throughput by adding shards.
* Use [Amazon CloudWatch](https://docs.aws.amazon.com/streams/latest/dev/monitoring-with-cloudwatch.html) on IteratorAge to determine if your Kinesis stream is being processed.

**Async Invokes**

* Create and use [AWS Lambda Function Dead Letter Queues](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#dlq) to address and replay async function errors**.**

**Lambda VPC**

* The following diagram guides you through a decision tree as to whether you should use a VPC (Virtual Private Cloud):



* **Don't put your Lambda function in a VPC unless you have to.** There is no benefit outside of using this to access resources you cannot expose publicly, like a private [Amazon Relational Database](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/) instance. Services like Amazon Elasticsearch Service can be secured over IAM with access policies, so exposing the endpoint publicly is safe and wouldn't require you to run your function in the VPC to secure it.
* **Lambda creates elastic network interfaces**[**(ENIs)**](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_ElasticNetworkInterfaces.html) in your VPC to access your internal resources. Before requesting a concurrency increase, ensure you have enough ENI capacity (the formula for this can be found here: [Configuring a Lambda Function to Access Resources in an Amazon VPC](https://docs.aws.amazon.com/lambda/latest/dg/vpc.html)) and IP address space. If you do not have enough ENI capacity, you will need to request an increase. If you do not have enough IP address space, you may need to create a larger subnet.
* **Create dedicated private subnets in your VPC:**

1. This will make it easier to apply a custom route table for NAT Gateway traffic without changing your other subnets. Lambda functions can only run in private subnets. For more information, see [Configuring a Lambda Function to Access Resources in an Amazon VPC](https://docs.aws.amazon.com/lambda/latest/dg/vpc.html)
2. This also allows you to dedicate an address space to Lambda without sharing it with other resources.

**Using Lambda with other services**

**34 - Services That Lambda Reads Events From** (Lambda reads data from the other service, creates an event, and invokes your function.)

* [Amazon Kinesis](https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html) streams
* [Amazon DynamoDB](https://docs.aws.amazon.com/lambda/latest/dg/with-ddb.html) streams
* [Amazon Simple Queue Service](https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html)

**35 - Services That Invoke Lambda Functions synchronously** (service waits for the response from your function and might [retry on errors](https://docs.aws.amazon.com/lambda/latest/dg/retries-on-errors.html).)

* [Elastic Load Balancing (Application Load Balancer)](https://docs.aws.amazon.com/lambda/latest/dg/services-alb.html)
* [Amazon Cognito](https://docs.aws.amazon.com/lambda/latest/dg/services-cognito.html)
* [Amazon Lex](https://docs.aws.amazon.com/lambda/latest/dg/services-lex.html) (AWS service for building conversational interfaces into applications using voice and text.)
* [Amazon Alexa](https://docs.aws.amazon.com/lambda/latest/dg/services-alexa.html)
* [Amazon API Gateway](https://docs.aws.amazon.com/lambda/latest/dg/with-on-demand-https.html)
* [Amazon CloudFront (Lambda@Edge)](https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html)
* [Amazon Kinesis Data Firehose](https://docs.aws.amazon.com/lambda/latest/dg/services-kinesisfirehose.html)

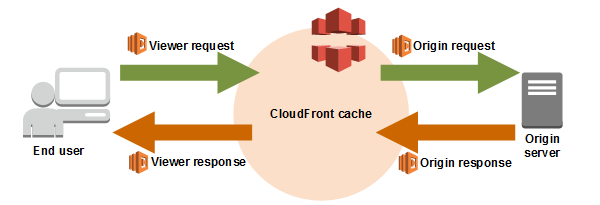
**36 - Services That Invoke Lambda Functions Asynchronously** (Lambda queues the event before passing it to your function. The other service gets a success response as soon as the event is queued and isn't aware of what happens afterwards. If an error occurs, Lambda handles retries, and can send failed events to a dead-letter queue that you configure.)

* [Amazon Simple Storage Service](https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html)
* [Amazon Simple Notification Service](https://docs.aws.amazon.com/lambda/latest/dg/with-sns.html) (including events from RDS)
* [Amazon Simple Email Service](https://docs.aws.amazon.com/lambda/latest/dg/services-ses.html)
* [AWS CloudFormation](https://docs.aws.amazon.com/lambda/latest/dg/services-cloudformation.html)
* [Amazon CloudWatch Logs](https://docs.aws.amazon.com/lambda/latest/dg/services-cloudwatchlogs.html)
* [Amazon CloudWatch Events](https://docs.aws.amazon.com/lambda/latest/dg/with-scheduled-events.html)
* [AWS CodeCommit](https://docs.aws.amazon.com/lambda/latest/dg/services-codecommit.html)
* AWS IOT Button
* Cloud Watch Schedule Events

**Using AWS Lambda with CloudFront ( Lambda@Edge )**

Lambda@Edge lets you run Lambda functions to customize content that CloudFront delivers, executing the functions in AWS locations closer to the viewer. The functions run in response to CloudFront events, without provisioning or managing servers. You can use Lambda functions to change CloudFront requests and responses at the following points:

* After CloudFront receives a request from a viewer (viewer request)
* Before CloudFront forwards the request to the origin (origin request)
* After CloudFront receives the response from the origin (origin response)
* Before CloudFront forwards the response to the viewer (viewer response)



**Monitoring and Troubleshooting Lambda Applications**

**37 -** If your Lambda function code is executing, but you don't see any log data being generated after several minutes, this could mean your execution role for the Lambda function did not grant permissions to write log data to CloudWatch Logs

**38 - Lambda Monitoring Graphs / Metrics**

* **Invocations –** The number of times that the function was invoked in each 5-minute period.
* **Duration –** The average, minimum, and maximum execution times.
* **Error count and success rate (%) –** The number of errors and the percentage of executions that completed without error.
* **Throttles –** The number of times that execution failed due to concurrency limits.
* **IteratorAge –** For stream event sources, the age of the last item in the batch when Lambda received it and invoked the function.
* **DeadLetterErrors –** The number of events that Lambda attempted to write to a dead-letter queue, but failed.
* **ConcurrentExecutions**
* **UnreservedConcurrentExecutions**

**39 - Troubleshooting Scenario 1: Lambda Function Not Working as Expected**

* **Check your code and verify that it is working correctly.** An increased error rate would indicate that it is not. Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is /aws/lambda/<function name>. Following are some examples of errors that might show up in the logs:

1. If you see a stack trace in your log, there is probably an error in your code.
2. If you see a permissions denied error in the log,Check the IAM role and verify that it has all of the necessary permissions to access any AWS resources that your code references.
3. If you see a timeout exceeded error in the log, This may be because the timeout is too low, or the code is taking too long to execute.
4. If you see a memory exceeded error in the log, your memory setting is too low. Set it to a higher value.

* **Check your Lambda function and verify that it is receiving requests.** Even if your function code is working as expected and responding correctly to test invokes, the function may not be receiving requests from Amazon S3. If Amazon S3 is able to invoke the function, you should see an increase in your CloudWatch requests metrics. If you do not see an increase in your CloudWatch requests, check the access permissions policy associated with the function.

**40 - Troubleshooting Scenario 2: Increased Duration in Lambda Function Execution**

To determine why there is increased duration in the execution of a Lambda function

* **Test your code with different memory settings.**

If your code is taking too long to execute, it could be that it does not have enough compute resources to execute its logic. Try increasing the memory allocated to your function and testing the code again.

* **Use logs to investigate the source of the execution bottleneck**

Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is named aws/lambda/<function name>. Add logging statements around various parts of your code, such as callouts to other services, to see how much time it takes to execute different parts of your code.

**Lambda Security**

In addition to the AWS global infrastructure, Lambda offers several features to help support your data resiliency and backup needs.

* Versioning
* Scaling
* High availability
* Reserved concurrency
* Retries
* Dead-letter queue