# Lambda

Supported languages

* **.Net, Java, Ruby, Python, Node.Js, Go lang**

Important things

* Handler – entry point
* Event – Request message based on event source (S3, API)
* Context – Details about **lambda execution environment**
  + Request Id – error reporting
  + Remaining time
  + Logging

Invocation Models

* Push
  + S3
    - Ex, thumbnail creation
  + API Gateway – ex, webservice backend
  + SNS – CloudWatch Alarm messages
  + CloudFormation – custom resource to execute custom command
  + CloudWatchEvents – Custom/scheduled actions
* Pull
  + DynamoDB
  + Kinesis Streams – Real time analytics

Function Settings

* Code
* Runtime
* Handler
* Environment Variables
* Tags
* IAM Role
* Memory
  + Min 128 MB to Max 3008 MB
* Timeout
  + Default 3 seconds, Max 900 seconds (15 mins), earlier 300 seconds
* VPC
* DLQ
* Concurrency
  + Account level
    - By default, AWS Lambda limits the total concurrent executions across all functions within a given region to **1000.**
  + Function level
    - Unreserved concurrency based on account level

### Reserved vs. Unreserved Concurrency Limits

* + - 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.i f your total account limit is 1000, you are limited to allocating 900 to individual functions. 100 is kept for unreserved concurrency pool as minimum.
* Tracing

## Rules for Naming Environment Variables

There is no limit to the number of environment variables you can create as long as the total size of the set does not exceed 4 KB.Other requirements include:

* Must start with letters *[a-zA-Z]*.
* Can only contain alphanumeric characters and underscores *([a-zA-Z0-9\_]*.

**Event Source Mapping**

* AWS Services (Push model)
  + S3 notification
* AWS Poll-Based Services
  + Kinesis, DynamoDB, SQS
* Custom Applications

Lambda Limits

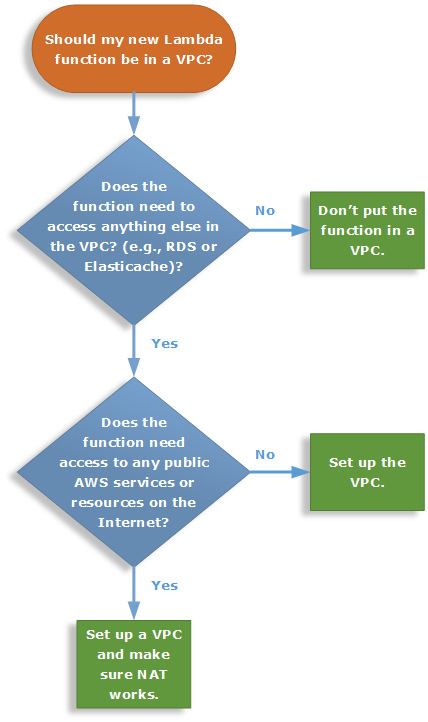
* **1000 concurrency execution**
* Fn Memory – **128 Mb to 3 Gb**
* Fn env. Variables **– 4 KB**
* Fn policy – 20 Kb
* Invocation pay load
  + Req – **6 Mb**
  + Res – 256 Kb
* Deployment package
  + **50 Mb zipped**
  + **250 Mb unzipped**
  + 3 Mb console editors
* Threads – 1024
* **/tmp dir – 512 Mb**

Accessing private resources in Lambda

* By default, resources within a VPC are not accessible from within a Lambda function.
* AWS Lambda runs your function code securely within a 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** **VPC subnet IDs and security group IDs. Using this ENI (Elastic Network Interface) lambda will connect to secured resources within VPC.**

# Best Practices

* Code
  + Separate lambda handler and core function
  + Use of execution context
    - First time it retrieves dependency and stores it for subsequent invocations
    - Take advantage of Execution Context reuse to improve the performance of your function. **Make sure any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution**
    - **Limit the re-initialization of variables/objects on every invocation**. **Instead use static initialization/constructor, global/static variables and singletons**. Keep alive and reuse connections (HTTP, database, etc.) that were established during a previous invocation. (By declaring out of handler)
  + Use env. Variables to pass data to function
  + Control the dependencies in your function’s deployment package
    - Package all dependencies with the deployment package
  + **Minimize your deployment package size. Selectively include libraries from SDK**
  + **Avoid recursive code**
  + Minimize the complexity of your dependencies
    - Use IOC container, Dagger or Guice. Don’t use complex one like spring
  + **Reduce the time it takes lambda to unpack deployment package**
    - **Put dependency .jar files into \lib directory**
* Configuration
  + Performance testing your lambda function
  + Do load test
  + Use most restrictive permissions with IAM policies
  + Be familiar with Lambda limits
  + Delete unused lambda functions
* Async invokes
  + **Use DLQ to address and replay function errors**
* Lambda VPC



DLQ

* Any Lambda function invoked **asynchronously** is retried twice before the event is discarded.
* Unprocessed lambda events will be pushed to either SQS or SNS
* Message attributes

Name Type Value

RequestID String Unique request identifier

ErrorCode Number 3-digit HTTP error code

ErrorMessage String Error message (truncated to 1 KB)

# Monitoring

* AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics **through Amazon CloudWatch**. To help you monitor your code as it executes, Lambda automatically tracks the number of requests, the execution duration per request, and the number of requests resulting in an error and publishes the associated CloudWatch metrics. You can leverage these metrics to set CloudWatch custom alarms.
* You can **view request rates and error rates for each of your Lambda functions** by using the AWS Lambda console, the CloudWatch console, and other Amazon Web Services (AWS) resources

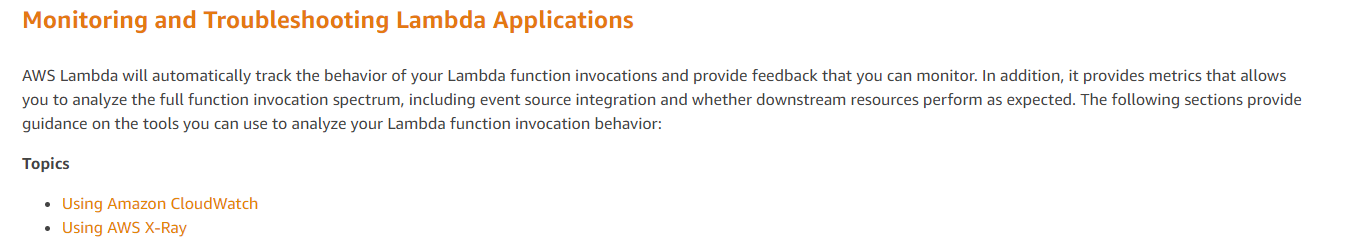
# Debugging

* You can **insert logging statements into your code to help you validate that your code is working as expected**. Lambda automatically integrates with Amazon CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function **(/aws/lambda/*<function name>*).**
* Logging issue
  + 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.

# Resolving external library dependency

* Minimize your deployment package size to its runtime necessities. This will reduce the amount **of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation**. For functions authored **in Java or .NET Core, avoid uploading the entire AWS SDK** library as part of your deployment package. Instead, **selectively depend on the modules which pick up components of the SDK you need**

# Troubleshooting



# Traffic Shifting Using Aliases

By default, an alias points to a single Lambda function version. When the alias is updated to point to a different function version, incoming request traffic in turn instantly points to the updated version. This exposes that alias to any potential instabilities introduced by the new version. To minimize this impact, 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.

When pointing an alias to more than one version, **the alias cannot point to $LATEST.**

**CLI**

2% to version 2

aws lambda create-alias --name *alias name* --function-name *function-name* \ --function-version 1

--routing-config AdditionalVersionWeights={**"2"=0.02**}

5% to version 2

aws lambda update-alias --name *alias name* --function-name *function-name* \

--routing-config AdditionalVersionWeights={"2"=0.05}

To route entire traffic

aws lambda update-alias --name *alias name* --function-name *function-name* \

--function-version 2 --routing-config AdditionalVersionWeights={}

# Determine which version is invoked

1. **CloudWatch Logs**

Lambda automatically emits a **START log entry that contains the invoked version ID to CloudWatch Logs for every function invocation**. An example follows.

19:44:37 START **RequestId**: *request id* Version: $*version*

1. Response Payload

Responses to synchronous function invocations include an **x-amz-executed-version**header to indicate which function version has been invoked.

# Errors in Lambda

Lambda returns **TooManyRequestsException** if executing the function would cause you to exceed a concurrency limit at either the account level (**ConcurrentInvocationLimitExceeded**) or function level (**ReservedFunctionConcurrentInvocationLimitExceeded**).

APIs

* CreateFunction
* UpdateFunctionCode

Following limit can be increased by contacting AWS

**CodeStorageExceededException**  - Due to function code storage size

|  |  |
| --- | --- |
| **Resource** | **Default Limit** |
| [Concurrent executions](https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html) | 1,000 |
| Function and layer storage | 75 GB |

**ServiceException** - Internal error

**TooManyRequestsException** - throughput limit exceeded

# Using AWS Lambda with Amazon S3

**Amazon S3 can publish events (for example, when an object is created in a bucket) to AWS Lambda and invoke your Lambda function by passing the event data as a parameter.** This integration enables you to write Lambda functions that process Amazon S3 events. **In Amazon S3, you add bucket notification configuration that identifies the type of event that you want Amazon S3 to publish and the Lambda function that you want to invoke**.

**Important**

If your Lambda function uses the same bucket that triggers it, it could cause the function to execute in a loop. For example, if the bucket triggers a function each time an object is uploaded, and the function uploads an object to the bucket, then the function indirectly triggers itself. To avoid this, use two buckets, or configure the trigger to only apply to a prefix used for incoming objects.

# Using Environment Variables to Communicate with AWS X-Ray

AWS Lambda uses environment variables to facilitate communication with **the X-Ray daemon and configure the X-Ray SDK.**

The sampling decision and trace ID are added to HTTP requests in **tracing headers named X-Amzn-Trace-Id.**

* **\_X\_AMZN\_TRACE\_ID:** Contains the tracing header, which includes the sampling decision, trace ID, and parent segment ID. (To learn more about these properties, see [Tracing Header](https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-tracingheader).) If Lambda receives a tracing header when your function is invoked, that header will be used to populate the \_X\_AMZN\_TRACE\_ID environment variable. If a tracing **header was not received, Lambda will generate one for you**.
* **AWS\_XRAY\_CONTEXT\_MISSING:** The X-Ray SDK uses this variable to determine its behavior in the event that your function tries to record X-Ray data, but a tracing header is not available. Lambda sets this value **to LOG\_ERROR** by default.
* **AWS\_XRAY\_DAEMON\_ADDRESS:** This environment variable exposes the X-Ray daemon's address in the following format: *IP\_ADDRESS***:***PORT*. You can use the X-Ray daemon's address to send trace data to the X-Ray daemon directly, without using the X-Ray SDK.

# Lambda@Edge

Lambda@Edge is an extension of AWS Lambda, a compute service that lets you execute functions that customize the content that CloudFront delivers.

Processing requests at AWS locations closer to the viewer instead of on origin servers significantly reduces latency and improves the user experience.

When you associate a CloudFront distribution with a Lambda@Edge function, **CloudFront intercepts requests and responses at CloudFront edge locations**. You can execute Lambda functions when the following CloudFront events occur:

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

Request -> lambda() -> forward requests to Origin

Responst <- lambda() <- receive response from Origin

Uses

* Cookies inspection
* Deliver content based on user’s devices (User-agent, Referrer header)
* Inspect authorization tokens.