
Lambda Warmer

license MIT license MIT coverage 99% dependencies 0

A module to optimize AWS Lambda function cold starts

At a recent AWS Startup Day event in Boston, MA, Chris Munns, the Senior Developer Advocate for Serverless at AWS, discussed **Cold Starts** and how to mitigate them. According to Chris (although he acknowledged that it is a “hack”) using the **CloudWatch Events “ping”** method is really the only way to do it right now. He gave a number of good tips on how to do this “correctly”:

- Don’t ping more often than every 5 minutes
- Invoke the function directly (i.e. don’t use API Gateway to invoke it)
- Pass in a test payload that can be identified as such
- Create handler logic that replies accordingly without running the whole function

He also mentioned that if you want to keep several **concurrent** functions warm, that you need to invoke the same function multiple times with delayed executions. This prevents the system from reusing the same container.

You can read the key takeaways from his talk [here](#).

Following these “best practices”, I created **Lambda Warmer**. It is a lightweight module (with no dependencies) that can be added to your Lambda functions to manage “warming” events as well as handling automatic fan-out for initializing *concurrent functions*. Just instrument your code and schedule a “ping”.

NOTE: Lambda Warmer will invoke the function multiple times using the AWS-SDK in order to scale concurrency (if you want to). Your functions **MUST** have `lambda:InvokeFunction` permissions so that they can invoke themselves. Following the Principle of Least Privilege, you should limit the `Resource` to the function itself, e.g.:

```
1 - Effect: "Allow"
2   Action:
3     - "lambda:InvokeFunction"
4   Resource: "arn:aws:lambda:us-east-1:{AWS-ACCOUNT-ID}:function:my-test-function"
```

If you’d like to know more about how **Lambda Warmer** works, and why you might (or might not) want to use it, read this *Lambda Warmer: Optimize AWS Lambda Function Cold Starts* post.

Using AWS SDK v2?

lambda-warmer@v2 is using AWS SDK v3. If you are using AWS SDK v2, please use lambda-warmer@v1.

Installation

Install Lambda Warmer from NPM as a project dependency.

```
1 npm i lambda-warmer
```

Instrumenting your Lambda functions

Adding Lambda Warmer to your functions is simple. Require `lambda-warmer` outside of your main handler and then pass the `event` as the first argument into your declared variable. Lambda Warmer will return a resolved promise with either `true`, meaning this *is* a warming invocation, or `false`, this isn't a warming invocation.

If you're using `async/await`, you can `await` the result of Lambda Warmer and `return` if true. This will short circuit your function and prevent it from executing the rest of the main handler.

```
1 const warmer = require('lambda-warmer')
2
3 exports.handler = async (event) => {
4   // if a warming event
5   if (await warmer(event)) return 'warmed'
6   // else proceed with handler logic
7   return 'Hello from Lambda'
8 }
```

If you are using `callbacks`, use Lambda Warmer to start a promise chain and then make your handler logic conditional depending on its result.

```
1 const warmer = require('lambda-warmer')
2
3 exports.handler = (event, context, callback) => {
4   // Start a promise chain
5   warmer(event).then(isWarmer => {
6     // if a warming event
7     if (isWarmer) {
8       callback(null, 'warmed')
9     } else proceed with handler logic
10    } else {
11      callback(null, 'Hello from Lambda')
12    }
13  })
14 }
```

```
13     })  
14 }
```

Configuration Options

You can send in a configuration object as the second parameter to change Lambda Warmer's default behavior. All of the settings are optional. Here is a sample configuration object.

```
1 {  
2   flag: 'warmer',  
3   concurrency: 'concurrency',  
4   test: 'test',  
5   log: true,  
6   correlationId: 'XXXXXXXXXXXX',  
7   delay: 75  
8 }
```

flag (string)

Name of the `event` field used to notify Lambda Warmer that this is a “warming” invocation. Defaults to `warmer`.

concurrency (string)

Name of the `event` field used to specify the number of concurrent functions you'd like to warm. Defaults to `concurrency`.

test (string)

Name of the `event` field used to flag a warming invocation as a test. Defaults to `test`.

log (boolean)

Flag to control whether or not CloudWatch logs are automatically generated. Defaults to `true`.

correlationId (string)

Identifier that gets passed to all concurrent Lambda invocations. This can be used to group invocations within your logs. If no `correlationId` is passed, it will default to the id generated for the invoked function. Passing the `context.awsRequestId` is good practice.

delay (number)

Minimum amount of time (in milliseconds) for concurrent functions to run. Concurrent functions are invoked asynchronously. Setting a delay enforces Lambda to create multiple invocations. Defaults to 75 to attempt sub 100ms invocation times.

target (string)

Name of the target function to be warmed. Defaults to `funcName` (the name of the function itself).

Example passing a configuration:

```
1 exports.handler = async (event, context) => {
2   // if a warming event
3   if (await warmer(event, { correlationId: context.awsRequestId, delay:
4     50 }))) return 'warmed'
5   // else proceed with handler logic
6   return 'Hello from Lambda'
7 }
```

Warming your Lambda functions

Lambda Warmer facilitates the warming of your functions by analyzing invocation events and appropriately managing handler processing. It **DOES NOT** manage the periodic invocation of your functions. In order to keep your functions warm, you must create a CloudWatch Rule that invokes your functions on a predetermined schedule.

A rule that invokes your function should contain a `Constant` (JSON text) under the “Configure input” setting. The following is a sample event (using the default configuration and a concurrency of 3):

```
1 { "warmer":true,"concurrency":3 }
```

The names of `warmer` and `concurrency` can be changed using the configuration option when instrumenting your code.

NOTE: Non-VPC functions are kept warm for approximately 5 minutes whereas VPC-based functions are kept warm for 15 minutes. Set your schedule for invocations accordingly. There is no need to ping your functions more often than the minimum warm time.

TIP: Preparing for traffic spikes

If your application experiences periodic traffic spikes throughout the day, you can set up multiple CloudWatch Rules that change the concurrency based on the time of day or day itself.

Using a SAM Template

To add a schedule event to your Lambda functions, you can add a `Type: Schedule` to the `Events` section of your function in a SAM template:

```
1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: 'AWS::Serverless-2016-10-31'
3 Resources:
4   MyFunction:
5     Type: 'AWS::Serverless::Function'
6     Properties:
7       Handler: index.handler
8       Runtime: nodejs16.x
9       CodeUri: 's3://my-bucket/function.zip'
10    Events:
11      WarmingSchedule:
12        Type: Schedule
13        Properties:
14          Schedule: rate(5 minutes)
15          Input: '{ "warmer":true,"concurrency":3 }'
```

Using the Serverless Framework

If you are using the Serverless Framework, you can include a `schedule` event for your functions using the following format:

```
1 myFunction:
2   name: myFunction
3   handler: myFunction.handler
4   events:
5     - schedule:
6       name: warmer-schedule-name
7       rate: rate(5 minutes)
8       enabled: true
9       input:
```

```
10     warmer: true
11     concurrency: 1
```

Setting multiple targets

In addition to passing a single-target input (either the function itself or the configured target), Lambda Warmer also accepts an array of events, each allowing a separate config (concurrency, target, etc.). This allows the re-use of a single CloudWatch rule for multiple targets, beyond the limit of CloudWatch itself, which is 5. It also simplifies sharing the rule in Serverless.

```
1 myFunction:
2   name: myFunction
3   handler: myFunction.handler
4   events:
5     - schedule:
6       name: warmer-schedule-name
7       rate: rate(5 minutes)
8       enabled: true
9       input:
10        - warmer: true
11          concurrency: 1
12          target: myOtherFunction
13        - warmer: true
14          concurrency: 2
15          target: myOtherFunction2
16        - warmer: true
17          concurrency: 2
18          target: myOtherFunction3
```

Logs

Logs are automatically generated unless the `log` configuration option is set to `false`. Logs contain useful information beyond just invocation data. The `warm` field indicates whether or not the Lambda function was already warm when invoked. The `lastAccessed` field is the timestamp (in milliseconds) of the last time the function was accessed by a non-warming event. Similarly, the `lastAccessedSeconds` gives you a counter (in seconds) of how long it's been since it has been accessed. These can be used to determine if your concurrency can be lowered.

Sample log:

```
1 {
2   action: 'warmer', // identifier
3   function: 'my-test-function', // function name
4   id: '1531413096993-0568', // unique function instance id
```

```
5  correlationId: '1531413096993-0568', // correlation id
6  count: 1, // function number of total concurrent e.g. 3 of 10
7  concurrency: 2, // number of concurrent functions being invoked
8  warm: true, // was this function already warm
9  lastAccessed: 1531413096995, // timestamp (in ms) of last non-warming
    access
10 lastAccessedSeconds: '25.6' // time since last non-warming access
11 }
```

Contributing

I've created a number of custom scripts to do similar cold start mitigation, but I figured I'd share this more complete version to save everyone some time (including my future self). If you would like to contribute, please submit a PR or add issues for bug reports and feature ideas.