A COMPREHENSIVE ANALYSIS OF AWS LAMBDA FUNCTION: OPTIMIZE SPIKES AND PREVENT COLD STARTS

AWS Lambda | DevOps | Serverless

beSharp | 20 March 2020

When it comes to Serverless, many are the aspects that we have to keep in mind to avoid latency and produce better, more reliable and robust applications. In this article, we will discuss many aspects we need to keep in mind when developing through AWS Lambda, how we can avoid common problems and how to exploit some of the recently introduced features to create more performant, efficient and less costly Serverless applications.

Cold Starts

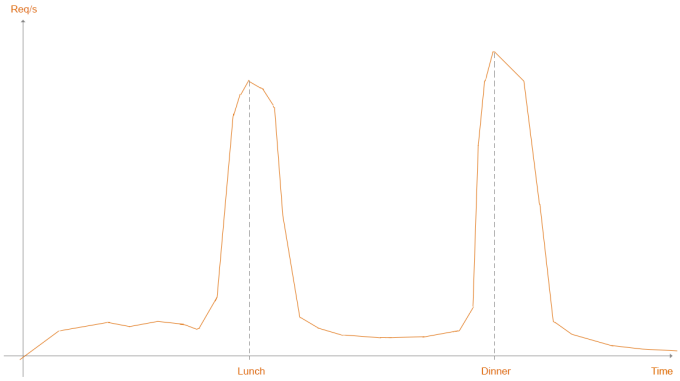
For years the topic of cold starts has been one of the hottest and most frequently debated topics in the Serverless community.

Suppose you've just deployed a brand new Lambda Function. Regardless of the way the function is invoked, a new Micro VM needs to be instantiated, since there are no existing instances already available to respond to the event. The time needed to set up the Lambda Function runtime, together with your code, all of its dependencies and connections, is commonly called Cold Start.

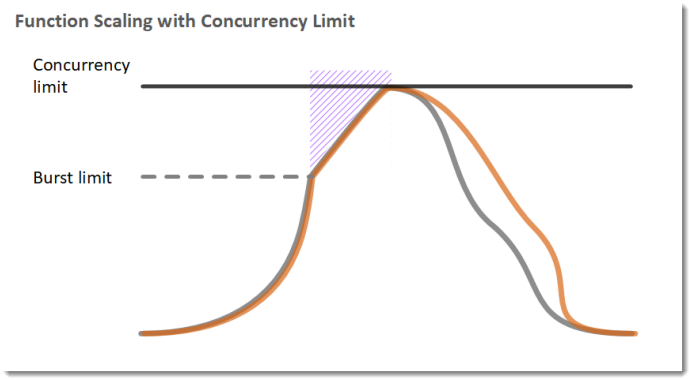
Depending on the runtime you choose, this setup process could take at least 50 – 200 ms before any execution actually started. Java and .Net Lambdas often experience cold starts that last for several seconds!

Depending on your use case, cold starts may be a stumbling block, preventing you from adopting the Serverless paradigm. Cold Starts should be avoided in scenarios where low-latency is a driver factor, e.g. customer-facing applications. Luckily, for many developers, this situation is an avoidable issue because their workload is predictable and stable or is mainly based on internal calculations, e.g. data-processing.

AWS documentation provides an example to better understand cold starts issues correlated to scaling needs. Imagine some companies, such as JustEat or Deliveroo, which experience very spiky traffic around lunches and dinners.



These spikes cause the application to run into limits such as how quickly AWS Lambda is able to scale out after the initial burst capacity. After the initial burst, it can scale up linearly to 500 instances per minute to serve your concurrent requests. Before it can handle the incoming requests, each new instance should face a cold start. Concurrency limit and high latencies due to cold starts could make your function scaling not able to deal with incoming traffic, causing new requests to be throttled.

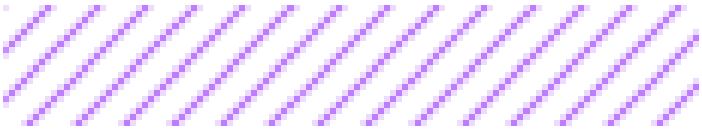


courtesy of AWS

Legend

Function instances

Open requests



Throttling possible

Reserved Concurrency

Concurrency has a regional limit that is shared among the functions in a Region, so this is also to take into account when some Lambdas are subject to very frequent calls. See the table below:

Burst Concurrency Limits

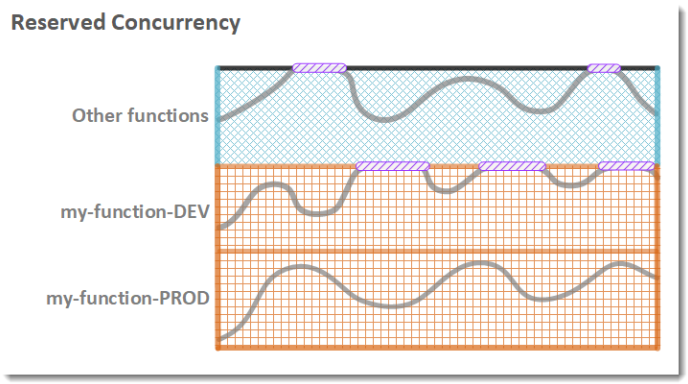
- **3000** – US West (Oregon), US East (N. Virginia), Europe (Ireland)
- **1000** – Asia Pacific (Tokyo), Europe (Frankfurt)
- **500** – Other Regions

To ensure that a specific function can always reach a specific level of concurrency and to restrict the number of instances of a Lambda Function that has access to downstream resources (like a database), you can configure **reserved concurrency.** When a function has reserved concurrency enabled, it always has the possibility to raise its number of instances to the one specified in the reserved concurrency configuration, regardless of other Lambda Functions' utilization. Reserved concurrency applies to the function as a whole, including versions and aliases.

To reserve concurrency for a function follow these simple steps:

1. Open the Lambda console Functions page and choose a function.
2. Under Concurrency, choose Reserve concurrency.
3. Enter the amount of concurrency to reserve for the function and save.

The following example shows how the reserved concurrency can help manage throttling.

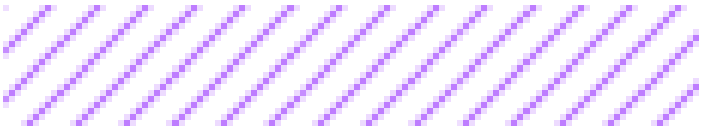


courtesy of AWS

Legend

Function instances

Open requests



Throttling possible

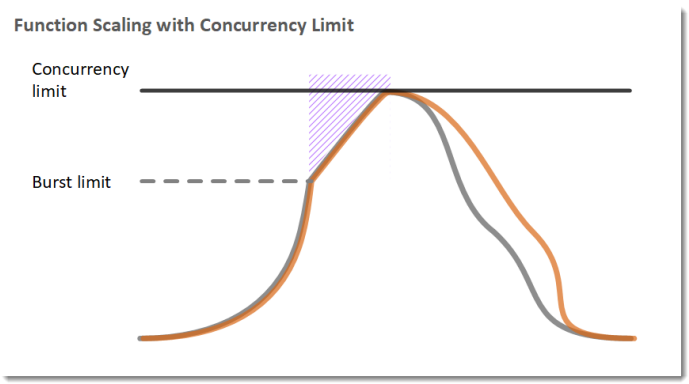
Reserved Concurrency can be applied when we have a clear understanding of the maximum possible concurrency rate, but cannot avoid problems related to cold start as every new instance created to sustain the concurrency rate will incur in that problem.

Before AWS released the Provisioned Concurrency feature, trying to avoid or even reduce cold-start to make Lambdas more responsive, was a very difficult task: you'll have to rely on custom user's logic to verify if a Lambda was ready or warming up. Later on, some libraries like lambda-warmer and serverless-plugin-warmup took the spot, but still may not be considered an ideal and clean solution.

Provisioned Concurrency

To enable your function to scale without fluctuations in latency, use **provisioned concurrency**. It allows you to configure warm instances right from the start and doesn't require code changes.

The following example shows a function with provisioned concurrency processing a single spike in traffic.

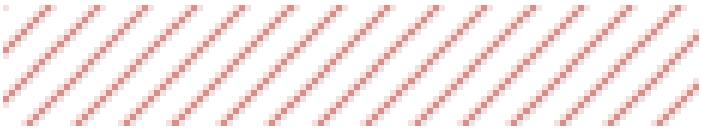


courtesy of AWS

Legend

Function instances

Open requests



Provisioned concurrency

Standard concurrency

When **provisioned concurrency** is allocated, the function scales with the same burst behavior as standard concurrency.

After it's allocated, **Provisioned Concurrency** serves incoming requests with very low latency. When all provisioned concurrency is in use, the function scales up normally to handle any additional requests. Those additional requests will meet cold starts but they should be few if you have properly configured Provisioned Concurrency.

Thanks to this new feature it is now possible to migrate to serverless workloads that were previously difficult to migrate, such as:

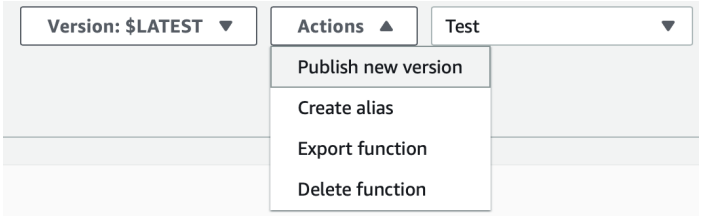
- APIs that run on Java/.Net runtimes
- Microservices where there are many API-to-API communications.
- APIs that have a very spiky traffic pattern.

You can configure Provisioned Concurrency for an Alias or for a Version. If you configure provisioned execution for the Alias, the associated Versions will inherit the Alias' Provisioned Concurrency configuration. Otherwise, each Version should have its own configuration. This way, it is possible to associate different Lambda Function Versions with different concurrencies to different traffic loads, not only, it is possible to do AB testing by exploiting this functionality.

Remember that you cannot configure Provisioned Concurrency against the $LATEST Alias or any Alias that points to it.

To publish a Lambda Version, move to the specific Lambda Function's details page from the AWS Lambda Console and publish a new version, clicking "Publish new version" from the "Actions" drop-

down menu.

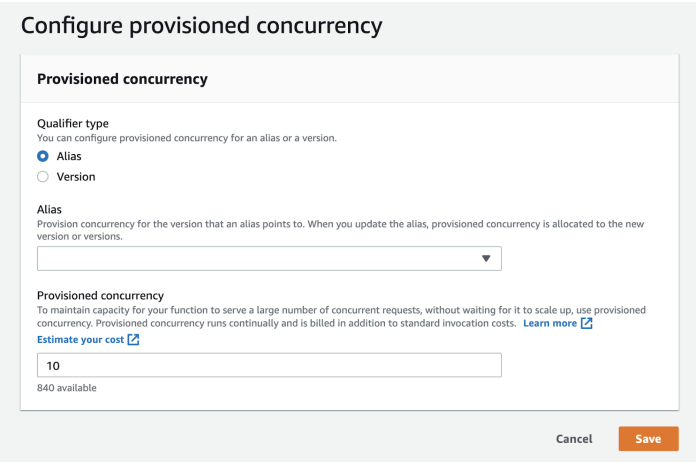


After publishing it, the Lambda version is set:

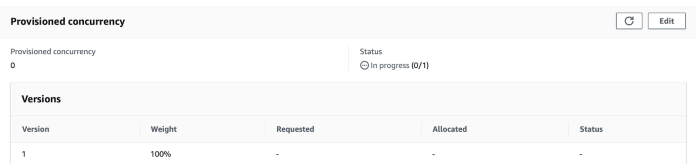
At this point, you could configure Provisioned Concurrency for the newly created Version 1 but, for the sake of this article, we will explore how to configure Provisioned Concurrency for a new Alias that points to Version 1.

Therefore, under the **Aliases** section, click on "+ **Create alias**". This command opens a modal in which you can create a new Alias which needs to be pointed to the previously created Version 1.

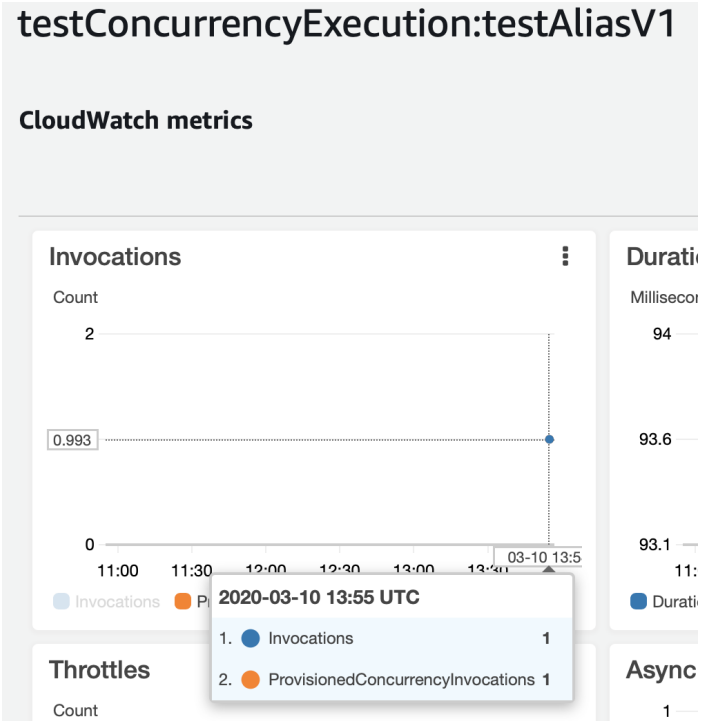
Once the Alias has been created, you can configure Provisioned Concurrency for it.



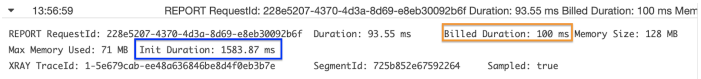
Move to the "Provisioned concurrency" section. Here you can configure **Provisioned Concurrency** on the newly created Alias, defining a value that represents how many concurrent executions you can provide out of your Reserved pool. It's simple as that; just be aware that this consists of an additional cost as specified by AWS.



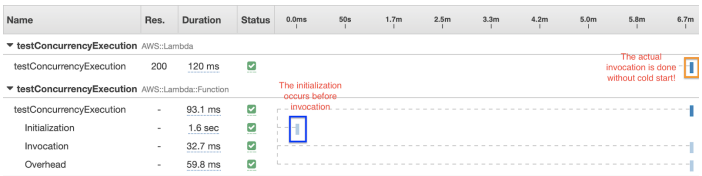
Once fully provisioned, the "Status" column, under the "Provisioned concurrency" section, will change to **Ready**. Invocations will then be handled by the Provisioned Concurrency ahead of standard on-demand concurrency.



In the graph, we can see that the Lambda invocation is now managed by provisioned instances and if we check Cloudwatch Logs we can easily see that the first invocation reports the **Init Duration**, which corresponds to the time needed to provide the requested number of concurrent executions by Lambda, together with the standard **Billed Duration**.



You can see evidence of this also in the X-Ray trace for the first invocation.



To make a further enhancement we can provide **autoscaling for provisioned concurrency**. When using Application Auto Scaling, you can create a **target tracking scaling policy** that modifies the number of concurrent executions based on the utilization metric emitted by Lambda.

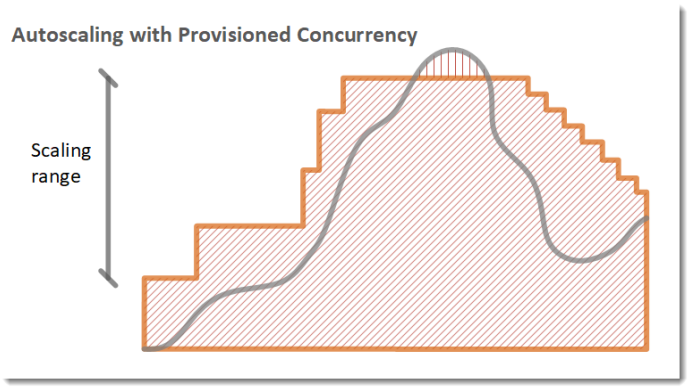
On a side note, since there is no clear way to delete the Provisioned Concurrency configuration from the AWS console, you can use the following aws-cli command:

```
aws lambda delete-provisioned-concurrency-config --function-name
<function-name> --qualifier <version-number/alias-name>
```

Auto Scaling API

Use the Application **Auto Scaling API** to register an alias as a scalable target and create a scaling policy.

In the following example, a function scales between a minimum and maximum amount of provisioned concurrency based on utilization; when the number of open requests increases, Application Auto Scaling increases provisioned concurrency in large steps until it reaches the configured maximum.

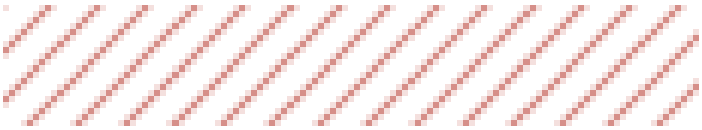


courtesy of AWS

Legend

Function instances

Open requests



Provisioned concurrency

Standard concurrency

The function continues to scale on standard concurrency until utilization starts to drop. When utilization is consistently low, Application Auto Scaling decreases provisioned concurrency in smaller periodic steps (right-half of the picture).

Beside utilization-based scaling, AWS Auto Scaling allows you to schedule scaling actions. In both scaling strategies, you have to first register the alias as a scaling target for AWS Auto Scaling.

Lambda Metrics

New or improved Aws Lambda metrics are now available to help define the traffic peaks as well as how your lambdas respond in terms of concurrency and simultaneous invocation in general.

Concurrency Metrics

- `ConcurrentExecutions`

  – The number of function instances that are processing events. If this number reaches your concurrent executions limit for the Region or the reserved concurrency limit that you configured on the function, additional invocation requests are throttled.

- `ProvisionedConcurrentExecutions`

  – The number of function instances that are processing events on provisioned concurrency. For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count.

- `ProvisionedConcurrencyUtilization`

  – For a version or alias, the value of

  `ProvisionedConcurrentExecutions`

  divided by the total amount of provisioned concurrency allocated. For example,

  `.5`

  indicates that 50 percent of allocated provisioned concurrency is in use.

- `UnreservedConcurrentExecutions`

  – For an AWS Region, the number of events that are being processed by functions that don't have reserved concurrency.

Up till now, we have talked about setting up Provisioned Concurrency in response to our needs of fast responding lambdas, but as a DevOps, probably you would like the ability to manipulate these parameters as code. So following are some examples for Serverless and SDK.

Serverless Example

In Serverless you can set the **provisionedConcurrency** and the **reservedConcurrency** properties as per documentation, the following is a simple example with all the parameters you can set:

```
service: myService
provider:
  name: aws
  runtime: nodejs12.x
  memorySize: 512 # optional, in MB, default is 1024
  timeout: 10 # optional, in seconds, default is 6
  versionFunctions: false # optional, default is true
  tracing:
    lambda: true # optional, enables tracing for all functions (can be true (true equa
ls 'Active') 'Active' or 'PassThrough')
functions:
  hello:
    handler: handler.hello # required, handler set in AWS Lambda
    name: ${self:provider.stage}-lambdaName # optional, Deployed Lambda name
    description: Description of what the lambda function does # optional, Description
 to publish to AWS
    runtime: python2.7 # optional overwrite, default is provider runtime
    memorySize: 512 # optional, in MB, default is 1024
    timeout: 10 # optional, in seconds, default is 6
    provisionedConcurrency: 3 # optional, Count of provisioned lambda instances
    reservedConcurrency: 5 # optional, reserved concurrency limit for this function. B
y default, AWS uses account concurrency limit
    tracing: PassThrough # optional, overwrite, can be 'Active' or 'PassThrough'
```

In the example above, the hello Lambda function will always have **3 warm instances** ready to go to handle incoming HTTP requests from API Gateway.

However, it doesn't end there. You can even go so far as to write a simple Lambda that you run on an hourly basis with a pre-determined schedule in mind. If you are like a lot of organisations, you will have busy spikes you know about well in advance. In these situations, you may not want provisionedConcurrency all the time, but you may want it during those known spikes.

Provisioned Concurrency can be also set via the **AWS SDK**:

```
const AWS = require('aws-sdk');
const params = {
FunctionName: 'Hello',
ProvisionedConcurrentExecutions: '3',
Qualifier: 'your-alias-here'
};
const lambda = new AWS
lambda.putProvisionedConcurrencyConfig(params, function(err, data) {
if (err) { console.log(err, err.stack); } // an error occurred
else { console.log(data); } // successful response
});
```

Now you have the means to schedule the provisioned concurrency whenever you choose and so optimise the cost efficiency of it.

Saving Plan AWS Lambda

To further entice users in the adoption of Provisioned Concurrency, Compute Saving Plans, now includes AWS Lambda, alongside Amazon EC2 and AWS Fargate. The service provides extra discounts for making a commitment for a consistent computational usage for a range of 1 up to 3 years.

The benefit of Compute Saving Plans is that the commitment is based solely on computational usage and thus it still allows to freely change the Size, Type or any other characteristics of the aforementioned resources.

By using Savings Plans, Lambda usage for Duration, Provisioned Concurrency, and Duration (Provisioned Concurrency) will be charged at discounted Savings Plans rates of up to 17% as stated by AWS documentation.

Although there is no specific discount for the Requests usage, it will still be covered by Savings Plans, thus helping customers to better exploit their commitment.

Conclusion

Finally one of the most fiercely debated potential limitations of the AWS Lambda platform is becoming a thing of the past, and this is great news for DevOps and in general for anyone willing to adopt a serverless paradigm.

To summarize, in this post we discussed:

- The problems with cold starts and why it’s a blocker for some AWS customers.
- What is Provisioned Concurrency and how does it work?
- How does it work with auto-scaling?
- Some snippet to enable Provisioned concurrency via templating
- How saving plans can help reduce costs

I hope you have enjoyed this post. Still curious about that? Contact us!

See you soon 🙂

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it

Copyright © 2011-2021 by beSharp srl - P.IVA IT02415160189