Performance tuning for serverless web applications

James Beswick, AWS Serverless
May 28, 2020
What I'll cover today:

• Cold starts
• Memory and profiling
• Architecture and best practices – applied to Fresh Tracks
How does Lambda work?
Anatomy of an AWS Lambda function

- Your function
- Language runtime
- Execution environment
- Lambda service
- Compute substrate
Where you can impact performance...

Your function
Language runtime
Execution environment
Lambda service
Compute substrate
## Anatomy of an AWS Lambda function

### Handler () function
Function to be executed upon invocation

```python
import json
import mylib

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello World!')
    }
```

### Event object
Data sent during Lambda function Invocation

```python
import json
import mylib

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello World!')
    }
```

### Context object
Methods available to interact with runtime information (request ID, log group, more)

```python
const MyLib = require('my-package')
const myLib = new MyLib()

exports.handler = async (event, context) => {
    # TODO implement
    return {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!')
    }
```

© 2020, Amazon Web Services, Inc. or its Affiliates.
Function lifecycle – worker host

Download your code

Start new Execution environment

Execute INIT code

Execute handler code

Full cold start

Partial cold start

Warm start

AWS optimization

Your optimization
Measuring with AWS X-Ray

Profile and troubleshoot serverless applications:

- Lambda instruments incoming requests and can capture calls made in code
- API Gateway inserts tracing header into HTTP calls and reports data back to X-Ray

```javascript
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))
AWSXRay.captureFunc('annotations', subsegment => {
  subsegment.addAnnotation('Name', name)
  subsegment.addAnnotation('UserID', event.userid)
})
```
## X-Ray Trace Example

### Trace Details
- **Method**: Application
- **Response**: 202
- **Duration**: 2.0 sec
- **Age**: 9.2 sec (2017-04-12 22:23:44 UTC)
- **ID**: 1-58eea8f0-259ac1ee9b0fdec33cf913d9

### Trace Breakdown

<table>
<thead>
<tr>
<th>Name</th>
<th>Res.</th>
<th>Duration</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamoDBexampleFunc</td>
<td>202</td>
<td>161 ms</td>
<td>✔️</td>
</tr>
<tr>
<td>Dwell Time</td>
<td>-</td>
<td>186 ms</td>
<td>✔️</td>
</tr>
<tr>
<td>Attestment #1</td>
<td>200</td>
<td>1.8 sec</td>
<td>✔️</td>
</tr>
<tr>
<td>DynamoDBexampleFunc</td>
<td>-</td>
<td>839 ms</td>
<td>✔️</td>
</tr>
<tr>
<td>Initialization</td>
<td>-</td>
<td>320 ms</td>
<td>✔️</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>200</td>
<td>679 ms</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Data Source**:
- AWS Lambda
- DynamoDB

**Details**:
- ListTables
Three areas of performance

Latency

Throughput

Cost
Cold starts
Function lifecycle – a warm start

1. Request made to Lambda API
2. Service identifies if warm execution environment is available
3. Yes
4. Complete invocation
5. Invoke handler
Function lifecycle – a full cold start

1. Request made to Lambda’s API
2. Service identifies if warm execution environments is available
3. Find available compute resource
   - No
4. Download customer code
5. Start execution environment
6. Execute INIT
7. Invoke handler
8. Complete invocation
Cold starts - Execution environment

The facts:
• <1% of production workloads
• Varies from <100ms to >1s

Be aware…
• You cannot target warm environments
• Pinging functions to keep them warm is limited
Cold starts - Execution environment

The facts:
• <1% of production workloads
• Varies from <100ms to >1s

Be aware...
• You cannot target warm environments
• Pinging functions to keep them warm is limited

Cold starts occur when...
• Environment is reaped
• Failure in underlying resources
• Rebalancing across Azs
• Updating code/config flushes
• Scaling up
Cold starts - Execution environment

Influenced by:

- Memory allocation
- Size of function package
- How often a function is called
- Internal algorithms

AWS optimizes for this part of a cold start.
Lambda + VPC – Major performance improvement

Announcing improved VPC networking for AWS Lambda functions

© 2020, Amazon Web Services, Inc. or its Affiliates.

Before: 14.8 sec duration

After: 933ms duration

Cold starts - Static initialization

The facts:
- Code run before handler
- Used to initialize objects, establish connections, etc.
- Biggest impact on cold-starts

Also occurs when...
- A new execution environment is run for the first time
- Scaling up
Cold starts - Static initialization

Influenced by:
• Size of function package
• Amount of code
• Amount of initialization work

The developer is responsible for this part of a cold start.

What can help…
• Code optimization
  • Trim SDKs
  • Reuse connections
  • Don’t load if not used
  • Lazily load variables

• Provisioned Concurrency
Provisioned Concurrency on AWS Lambda

Pre-creates execution environments, running INIT code.

- Mostly for latency-sensitive, interactive workloads
- Improved consistency across the long tail of performance
- Minimal changes to code or Lambda usage
- Integrated with AWS Auto Scaling
- Adds a cost factor for per concurrency provisioned but a lower duration cost for execution
  - This could save you money when heavily utilized

![Chart showing percentage of requests served within a certain time (ms) with and without provisioned concurrency]
Function lifecycle – a Provisioned Concurrency start

Function configured with Provisioned Concurrency

Find available compute resource

Download customer code

Execute INIT

Start execution environment
Function lifecycle – a Provisioned Concurrency invocation

1. Request made to Lambda’s API
2. Service identifies if warm execution environment is available
   - Yes
3. Complete invocation
   - Invoke handler

This becomes the default for all provisioned concurrency execution environments.
Provisioned Concurrency – things to know

- Reduces the start time to <100ms
- Can’t configure for $LATEST
  - Use versions/aliases
- Provisioning rampup of 500 per minute
- No changes to function handler code performance
- Requests above provisioned concurrency follow on-demand Lambda limits and behaviors for cold-starts, bursting, pricing
- Still overall account concurrency per limit region
- Wide support (CloudFormation, Terraform, Serverless Framework, Datadog, Epsagon, Lumigo, Thundra, etc.)
Provisioned Concurrency

Things to know:

• We provision more than you request.
• We still reap these environments.
• There is less CPU burst than On-Demand during init
Provisioned Concurrency - configuration

Configure provisioned concurrency

Provisioned concurrency

Version: 1
Aliases: -

Provisioned concurrency
To maintain capacity for your function to serve a large number of concurrent requests, without waiting for it to scale up, use provisioned concurrency. Provisioned concurrency runs continually and is billed in addition to standard invocation costs. Learn more Estimate your cost

5000
19724 available

Provisioned concurrency

<table>
<thead>
<tr>
<th>Provisioned concurrency</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>In progress (3001/5000)</td>
</tr>
</tbody>
</table>
## Provisioned Concurrency - configuration

<table>
<thead>
<tr>
<th>Provisioned concurrency</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>Ready</td>
</tr>
</tbody>
</table>
Provisioned Concurrency – Application Auto Scaling

MyScalableTarget:
  Type: AWS::ApplicationAutoScaling::ScalableTarget
  Properties:
    MaxCapacity: 100
    MinCapacity: 1
    ResourceId: !Sub function:${MyFunction}:live # You need to specify an alias or version here
    RoleARN: !Sub arn:aws:iam::${AWS::AccountId}:role/aws-service-role/lambda.application-autoscaling.amazonaws.com/AWS
    ScalableDimension: lambda:function:ProvisionedConcurrency
    ServiceNamespace: lambda
    DependsOn: MyFunctionAliaslive # This is your function logical ID + "Alias" + what you use for AutoPublishAlias

MyTargetTrackingScalingPolicy:
  Type: AWS::ApplicationAutoScaling::ScalingPolicy
  Properties:
    PolicyName: utilization
    PolicyType: TargetTrackingScaling
    ScalingTargetId: !Ref MyScalableTarget
    TargetTrackingScalingPolicyConfiguration:
      TargetValue: 0.70 # Any value between 0 and 1 can be used here
      PredefinedMetricSpecification:
        PredefinedMetricType: LambdaProvisionedConcurrencyUtilization
Memory and profiling
Resources allocation

Memory ➡ Power
## CPU-bound example

“Compute **1,000 times** all prime numbers <= 1M”

<table>
<thead>
<tr>
<th>Memory (MB)</th>
<th>Time (sec)</th>
<th>Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>11.722</td>
<td>$0.024628</td>
</tr>
<tr>
<td>256</td>
<td>6.678</td>
<td>$0.028035</td>
</tr>
<tr>
<td>512</td>
<td>3.194</td>
<td>$0.026830</td>
</tr>
<tr>
<td>1024</td>
<td>1.465</td>
<td>$0.024638</td>
</tr>
</tbody>
</table>
Meet AWS Lambda Power Tuning

Features:

- Data-driven cost and performance optimization for AWS Lambda
- Available as an AWS Serverless Application Repository app
- Easy to integrate with CI/CD

https://github.com/alexcasalboni/aws-lambda-power-tuning

© 2020, Amazon Web Services, Inc. or its Affiliates.
Meet AWS Lambda Power Tuning

AWS Lambda Power Tuning is an open-source tool that can help you visualize and fine-tune the memory/power configuration of Lambda functions. It runs in your AWS account - powered by AWS Step Functions - and it supports multiple optimization strategies.

AWS Lambda Power Tuning

AWS Lambda Power Tuning is an AWS Step Functions state machine that helps you optimize your Lambda functions in a data-driven way.

The state machine is designed to be quick and language agnostic. You can provide any Lambda function as input, and the state machine will run it with multiple power configurations (from 128MB to 3GB), analyze execution logs and suggest you the best configuration to minimize cost or maximize performance.

The input function will be executed in your AWS account - performing real HTTP calls, SDK calls, cold starts, etc. The state machine also supports cross-region invocations and you can enable parallel execution to generate results in just a few seconds. Optionally, you can configure the state machine to automatically optimize the function and the end of its execution.

Last but not least, the state machine will generate a dynamic visualization of average cost and speed for each power configuration (more details below).

How to execute the state machine

Once the state machine and all the Lambda functions have been deployed, you can execute the state machine and provide an input object.

You will find the new state machine in the Step Functions Console or in your app's Resources section.

The state machine name will be prefixed with powerTuningStateMachine. Find it and click "Start execution". Here you can provide the execution input and an execution id (see section...)

https://github.com/alexcasalboni/aws-lambda-power-tuning
AWS Lambda Power Tuning (input)

```json
{
  "lambdaARN": "your-lambda-function-arn",
  "powerValues": [128, 256, 512, 1024, 2048, 3008],
  "num": 100,
  "payload": {"data": "abc"}
}
```
AWS Lambda Power Tuning (input)

```json
{
    "lambdaARN": "your-lambda-function-arn",
    "powerValues": 'ALL',
    "num": 100,
    "payload": {"data": "abc"}
}
```
AWS Lambda Power Tuning (input)

```json
{
    "lambdaARN": "your-lambda-function-arn",
    "powerValues": [128, 256, 512, 1024, 2048, 3008],
    "num": 100,
    "payload": {"data": "abc"},
    "parallelInvocation": true
}
```
AWS Lambda Power Tuning (input)

```json
{
    "lambdaARN": "your-lambda-function-arn",
    "powerValues": [128, 256, 512, 1024, 2048, 3008],
    "num": 100,
    "payload": {"data": "abc"},
    "strategy": "speed|cost"
}
```
AWS Lambda Power Tuning (input)

```json
{
    "lambdaARN": "your-lambda-function-arn",
    "powerValues": [128, 256, 512, 1024, 2048, 3008],
    "num": 100,
    "payload": {
        "data": "abc",
        "strategy": "balanced",
        "balancedWeight": 0.5
    }
}
```
AWS Lambda Power Tuning (input)

```json
{
    "lambdaARN": "your-lambda-function-arn",
    "powerValues": [128, 256, 512, 1024, 2048, 3008],
    "num": 100,
    "payload": {
        "data": "abc"
    },
    "autoOptimize": true,
    "autoOptimizeAlias": "prod"
}
```
AWS Lambda Power Tuning (output)

```json
{
    "results": {
        "power": "128",
        "cost": 2.08e-7,
        "duration": 2.906,
        "stateMachine": {
            "executionCost": 0.00045,
            "lambdaCost": 0.0005252,
            "visualization": "https://lambda-power-tuning.show/ ... "
        }
    }
}
```
AWS Lambda Power Tuning (visualization)

https://github.com/alexcasalboni/aws-lambda-power-tuning

© 2020, Amazon Web Services, Inc. or its Affiliates.
Optimization for web apps
Fresh Tracks architecture

Access
- AWS Cloud
- Auth0
- Amazon API Gateway
- AWS Amplify Console

Compute
- Custom Authorizer
  - [POST] /SignUrl
  - getSignedUrl S3
  - [GET] /Activity
  - getActivity
  - [GET] /Activities
  - getActivitiesForUser

Storage
- FreshTracks S3Bucket
- Amazon DynamoDB

Messaging
- Amazon EventBridge
  - [Message] Workflow Complete
  - Process GPX File
  - Save meta to DB
  - Publish to IOT

Orchestration
- AWS Step Functions Express workflow

© 2020, Amazon Web Services, Inc. or its Affiliates.
Optimization best practices

Avoid monolithic functions
- Reduce deployment package size
- Micro/Nano services

Optimize dependencies (and imports)
- E.g. up to 120ms faster with Node.js SDK

Minify/uglify production code
- Browserify/Webpack

Lazy initialization of shared libs/objs
- Helps if multiple functions per file
Optimized dependency usage (Node.js SDK & X-Ray)

```javascript
// const AWS = require('aws-sdk')
const DynamoDB = require('aws-sdk/clients/dynamodb') // 125ms faster

// const AWSXRay = require('aws-xray-sdk')
const AWSXRay = require('aws-xray-sdk-core') // 5ms faster

// const AWS = AWSXRay.captureAWS(require('aws-sdk'))
const dynamodb = new DynamoDB.DocumentClient()
AWSXRay.captureAWSClient(dynamodb.service) // 140ms faster
```
Reusing connections with HTTP Keep-Alive

- For functions using http(s) requests
- Use in SDK with environment variable
- Or set `keep-alive` property in your function code
- Can reduce typical DynamoDB operation from 30ms to 10ms
- Available in most runtime SDKs

For more details, visit https://bit.ly/reuse-connection
Fresh Tracks architecture

Access
- AWS Cloud
- Auth0
- Amazon API Gateway
- AWS Amplify Console

Compute
- Custom Authorizer
- getSignedUrl S3
- getActivity
- getActivitiesForUser

Storage
- FreshTracks S3Bucket
- Amazon DynamoDB

Messaging
- Amazon EventBridge
- AWS IoT Core

Orchestration
- AWS Step Functions Express workflow
  - Process GPX File
  - Save meta to DB
  - Publish to IOT

© 2020, Amazon Web Services, Inc. or its Affiliates.
Service integration

The facts:
- Adding services increases latency
- Mainly a synchronous concern

This occurs when…
- A service invokes Lambda
- Lambda calls a service

What can help…
- Moving away from synchronous execution
- Use the new HTTP APIs
- Use VTL where appropriate
- Transform don’t transport data
- Avoid Lambda calling Lambda
Comparing sync and async

**Synchronous:**
- The caller is waiting
- Waiting incurs cost
- Downstream slowdown affects entire process
- Process change is complex
- Dependent on custom code

**Asynchronous:**
- The caller receives ack quickly
- Minimizes cost of waiting
- Queueing separates fast and slow processes
- Process change is easy
- Uses managed services
Summary

Cold starts
• Causes of a cold start
• VPC improvements
• Provisioned Concurrency

Memory and profiling
• Memory is power for Lambda
• AWS Lambda Power Tuning
• Trade-offs of cost and speed

Architecture and optimization
• Best practices
• Service integration
• Async and service integration
Thanks!

James Beswick, AWS Serverless
jbeswick@amazon.com
@jbesw