AWS Compute Blog

Building well-architected serverless applications: Optimizing application performance – part 1

by Julian Wood | on 17 AUG 2021 | in Amazon CloudWatch, Amazon DynamoDB, Amazon RDS, AWS Lambda, AWS X-Ray, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the introduction post for a table of contents and explanation of the example application.

PERF 1. Optimizing your serverless application's performance

> Evaluate and optimize your serverless application's performance based on access patterns, scaling mechanisms, and native integrations. This allows you to continuously gain more value per transaction. You can improve your overall experience and make more efficient use of the platform in terms of both value and resources.

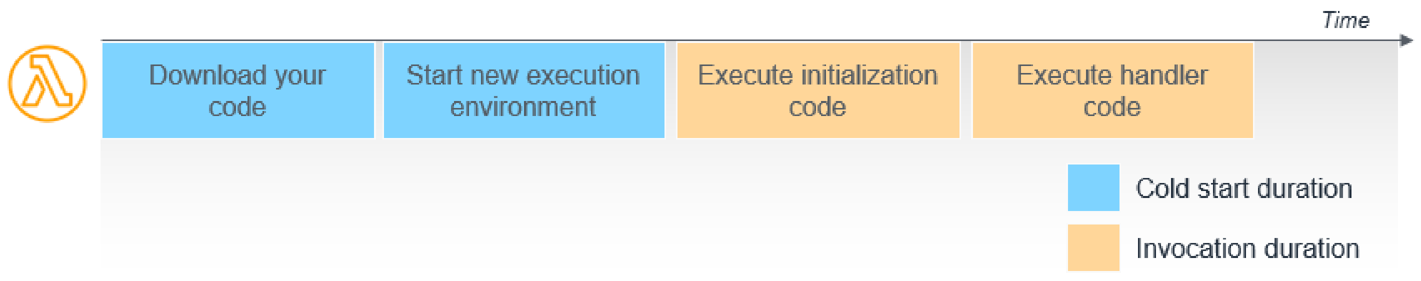
Good practice: Measure and optimize function startup time

Evaluate your AWS Lambda function startup time for both performance and cost.

Take advantage of execution environment reuse to improve the performance of your function.

Lambda invokes your function in a secure and isolated runtime environment, and manages the resources required to run your function. When a function is first invoked, the Lambda service creates an instance of the function to process the event. This is called a cold start. After completion, the function remains available for a period of time to process subsequent events. These are called warm starts.

Lambda functions must contain a handler method in your code that processes events. During a cold start, Lambda runs the function initialization code, which is the code outside the handler, and then runs the handler code. During a warm start, Lambda runs the handler code.



Lambda function cold and warm starts

Initialize SDK clients, objects, and database connections outside of the function handler so that they are started during the cold start process. These connections then remain during subsequent warm starts, which improves function performance and cost.

Lambda provides a writable local file system available at /tmp. This is local to each function but shared between subsequent invocations within the same execution environment. You can download and cache assets locally in the /tmp folder during the cold start. This data is then available locally by all subsequent warm start invocations, improving performance.

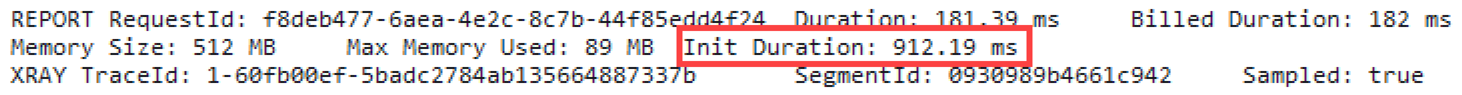
In the serverless airline example used in this series, the confirm booking Lambda function initializes a number of components during the cold start. These include the Lambda Powertools utilities and creating a session to the Amazon DynamoDB table `BOOKING_TABLE_NAME`.

Python
```python
import boto3
from aws_lambda_powertools import Logger, Metrics, Tracer
from aws_lambda_powertools.metrics import MetricUnit
from botocore.exceptions import ClientError

logger = Logger()
tracer = Tracer()
metrics = Metrics()

session = boto3.Session()
dynamodb = session.resource("dynamodb")
table_name = os.getenv("BOOKING_TABLE_NAME", "undefined")
table = dynamodb.Table(table_name)
```

Analyze and improve startup time

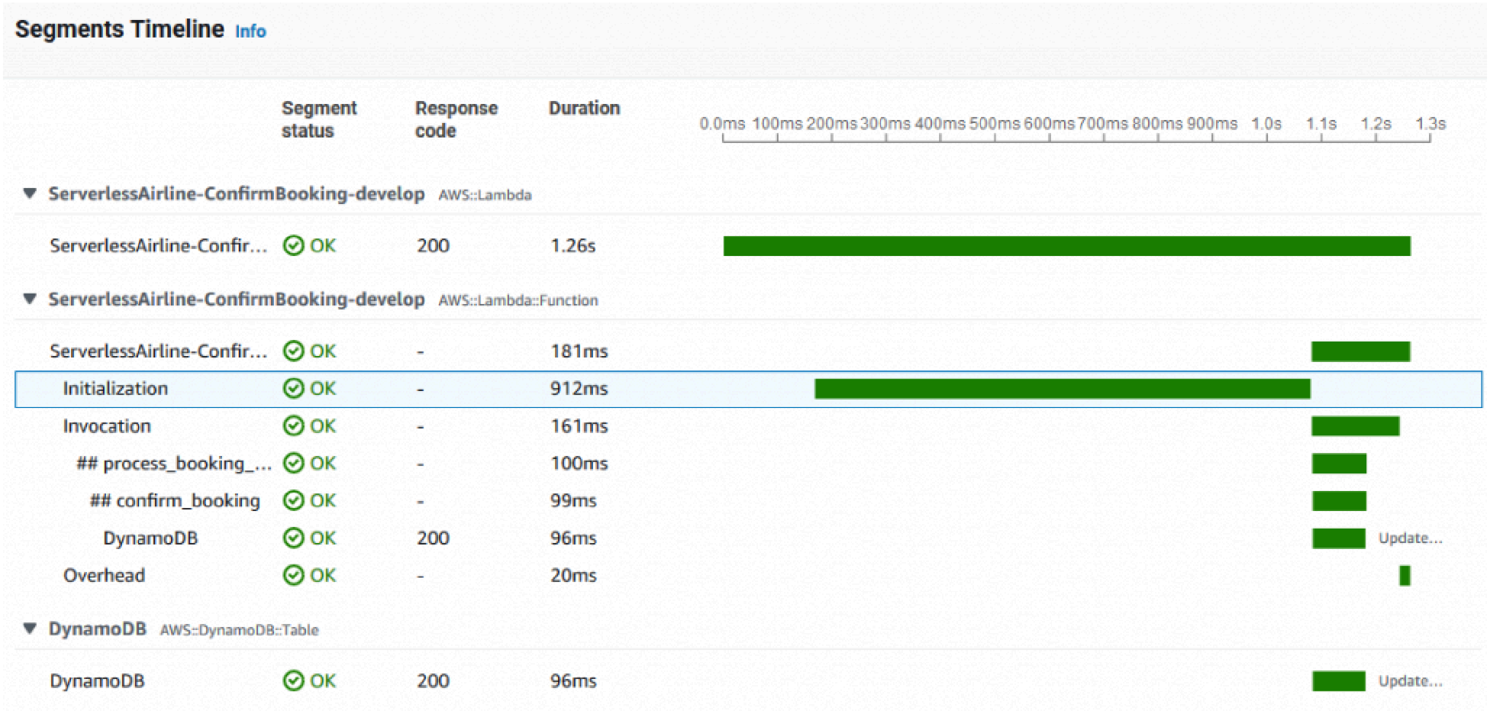
There are a number of steps you can take to measure and optimize Lambda function initialization time.

You can view the function cold start initialization time using Amazon CloudWatch Logs and AWS X-Ray. A log `REPORT` line for a cold start includes the `Init Duration` value. This is the time the initialization code takes to run before the handler.


```
REPORT RequestId: f8deb477-6aea-4e2c-8c7b-44f85edd4f24  Duration: 181.39 ms     Billed Duration: 182 ms
Memory Size: 512 MB     Max Memory Used: 89 MB  Init Duration: 912.19 ms
XRAY TraceId: 1-60fb00ef-5badc2784ab135664887337b       SegmentId: 0930989b4661c942     Sampled: true
```

CloudWatch Logs cold start report line

When X-Ray tracing is enabled for a function, the trace includes the *Initialization* segment.


Segments Timeline Info
0.0ms 100ms 200ms 300ms 400ms 500ms 600ms 700ms 800ms 900ms 1.0s 1.1s 1.2s 1.3s
Update...
Update...

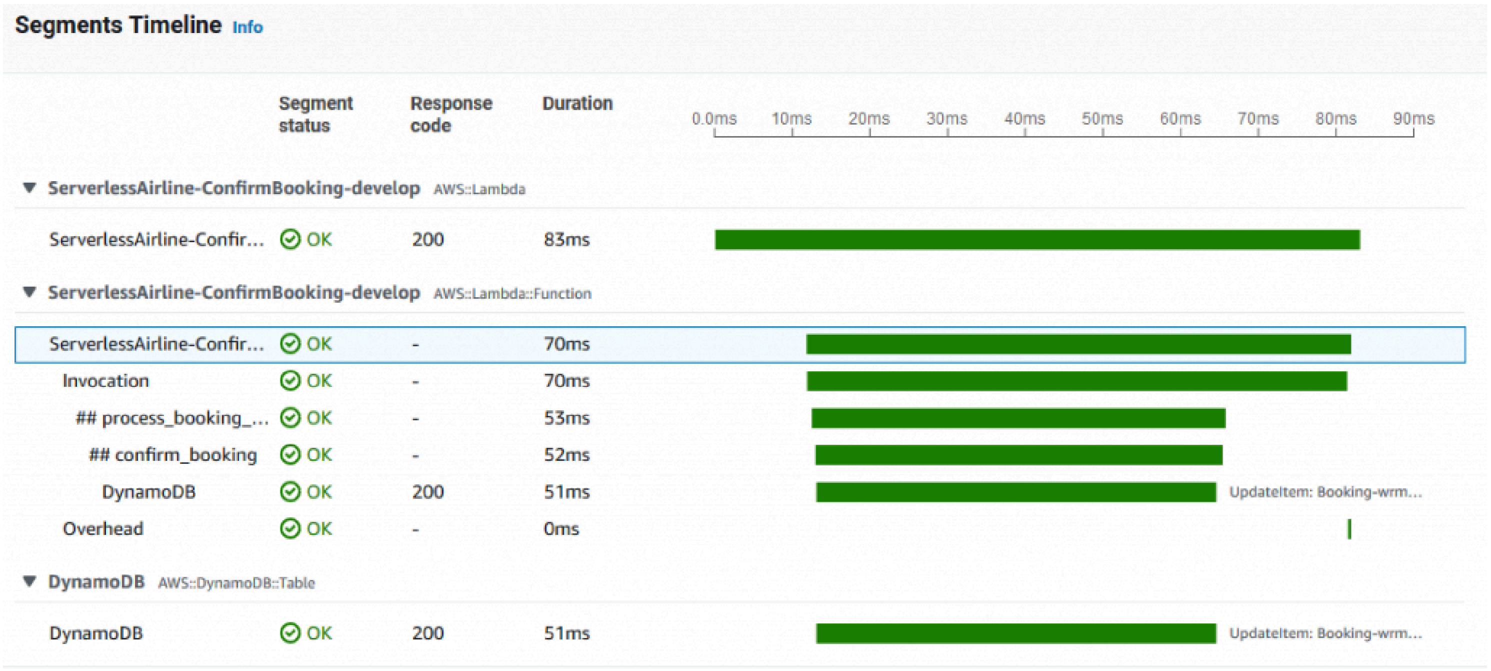
	Segment status	Response code	Duration
▼ ServerlessAirline-ConfirmBooking-develop AWS::Lambda			
ServerlessAirline-Confir...	OK	200	1.26s
▼ ServerlessAirline-ConfirmBooking-develop AWS::Lambda::Function			
ServerlessAirline-Confir...	OK	-	181ms
Initialization	OK	-	912ms
Invocation	OK	-	161ms
## process_booking_...	OK	-	100ms
## confirm_booking	OK	-	99ms
DynamoDB	OK	200	96ms
Overhead	OK	-	20ms
▼ DynamoDB AWS::DynamoDB::Table			
DynamoDB	OK	200	96ms

X-Ray trace cold start showing initialization segment

A subsequent warm start REPORT line does not include the Init Duration value, and is not present in the X-Ray trace:

```
REPORT RequestId: d907ca3d-ab0e-4ca4-9428-31e92d5b38cd  Duration: 70.28 ms      Billed Duration: 71 ms
Memory Size: 512 MB     Max Memory Used: 90 MB
XRAY TraceId: 1-60fb0178-4641c32751cc69fa635a41a5       SegmentId: 46c75f840287c33d     Sampled: true
```

CloudWatch Logs warm start report line


Segments Timeline Info
0.0ms 10ms 20ms 30ms 40ms 50ms 60ms 70ms 80ms 90ms
UpdateItem: Booking-wrm...
UpdateItem: Booking-wrm...

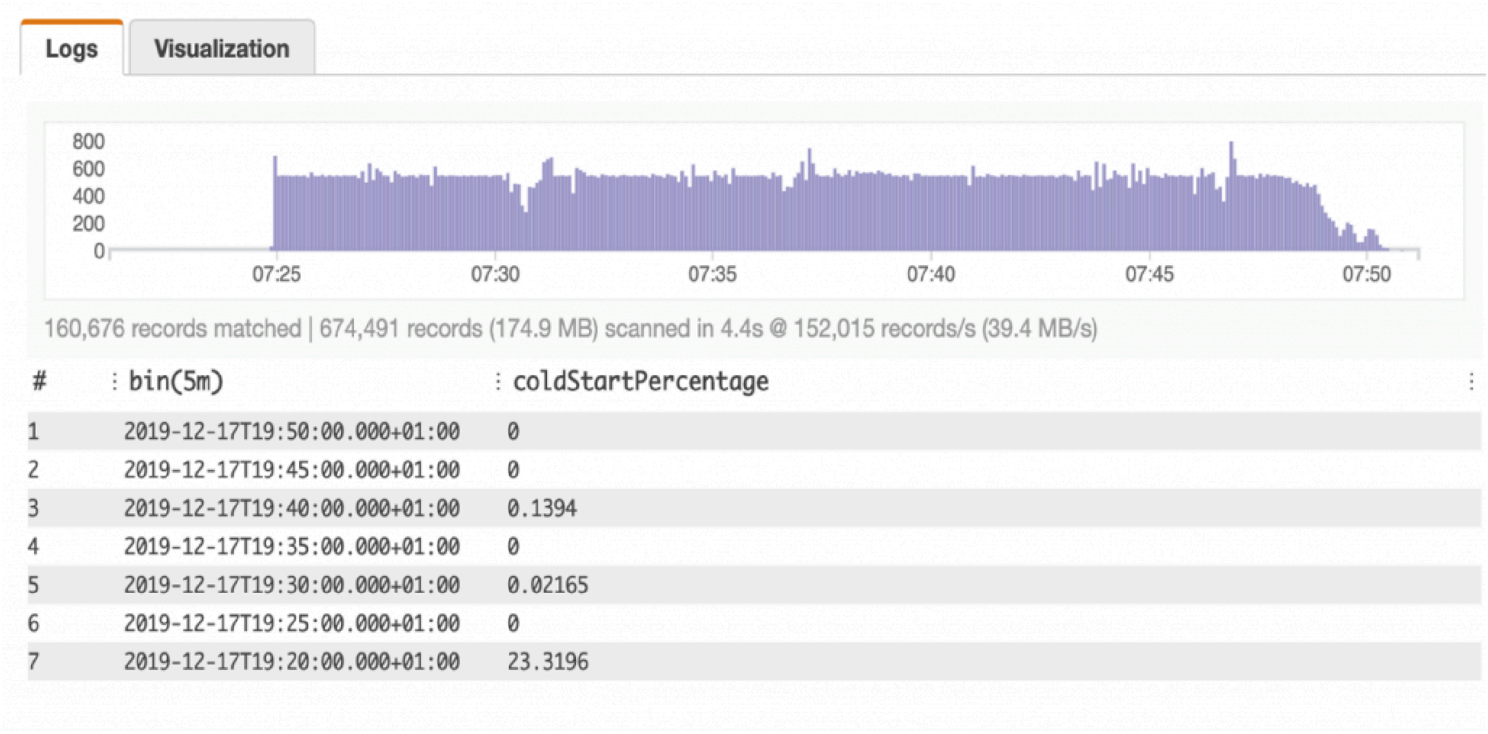
	Segment status	Response code	Duration
▼ ServerlessAirline-ConfirmBooking-develop AWS::Lambda			
ServerlessAirline-Confir...	OK	200	83ms
▼ ServerlessAirline-ConfirmBooking-develop AWS::Lambda::Function			
ServerlessAirline-Confir...	OK	-	70ms
Invocation	OK	-	70ms
## process_booking_...	OK	-	53ms
## confirm_booking	OK	-	52ms
DynamoDB	OK	200	51ms
Overhead	OK	-	0ms
▼ DynamoDB AWS::DynamoDB::Table			
DynamoDB	OK	200	51ms

X-Ray trace warm start without showing initialization segment

CloudWatch Logs Insights allows you to search and analyze CloudWatch Logs data over multiple log groups. There are some useful searches to understand cold starts.

Understand cold start percentage over time:

```
filter @type = "REPORT"
| stats
  sum(strcontains(
    @message,
    "Init Duration"))
  / count(*)
  * 100
  as coldStartPercentage,
  avg(@duration)
  by bin(5m)
```



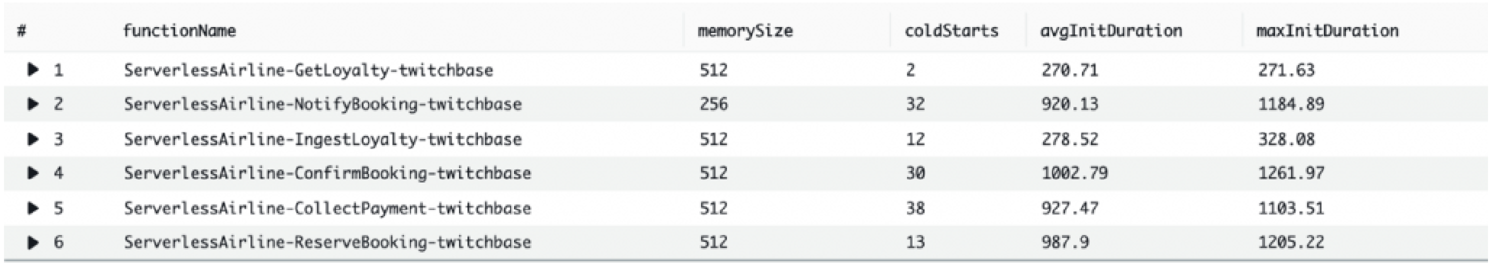
160,676 records matched | 674,491 records (174.9 MB) scanned in 4.4s @ 152,015 records/s (39.4 MB/s)

#	bin(5m)	coldStartPercentage
1	2019-12-17T19:50:00.000+01:00	0
2	2019-12-17T19:45:00.000+01:00	0
3	2019-12-17T19:40:00.000+01:00	0.1394
4	2019-12-17T19:35:00.000+01:00	0
5	2019-12-17T19:30:00.000+01:00	0.02165
6	2019-12-17T19:25:00.000+01:00	0
7	2019-12-17T19:20:00.000+01:00	23.3196

Cold start percentage over time

Cold start count and `InitDuration`:

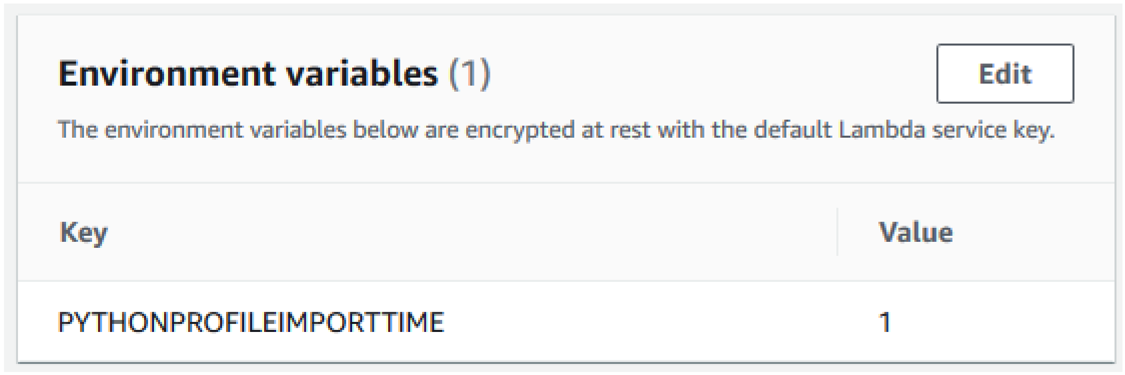
```
filter @type="REPORT"
| fields @memorySize / 1000000 as memorySize
| filter @message like /(?i)(Init Duration)/
| parse @message /^REPORT.*Init Duration: (?<initDuration>.*) ms.*/
| parse @log /^.*\/aws\/lambda\/(?<functionName>.*)/
| stats count() as coldStarts, median(initDuration) as avgInitDuration, max(initDurati
```



#	functionName	memorySize	coldStarts	avgInitDuration	maxInitDuration
▶ 1	ServerlessAirline-GetLoyalty-twitchbase	512	2	270.71	271.63
▶ 2	ServerlessAirline-NotifyBooking-twitchbase	256	32	920.13	1184.89
▶ 3	ServerlessAirline-IngestLoyalty-twitchbase	512	12	278.52	328.08
▶ 4	ServerlessAirline-ConfirmBooking-twitchbase	512	30	1002.79	1261.97
▶ 5	ServerlessAirline-CollectPayment-twitchbase	512	38	927.47	1103.51
▶ 6	ServerlessAirline-ReserveBooking-twitchbase	512	13	987.9	1205.22

Cold start count and `InitDuration`

Once you have measured cold start performance, there are a number of ways to optimize startup time. For Python, you can use the `PYTHONPROFILEIMPORTTIME=1` environment variable.



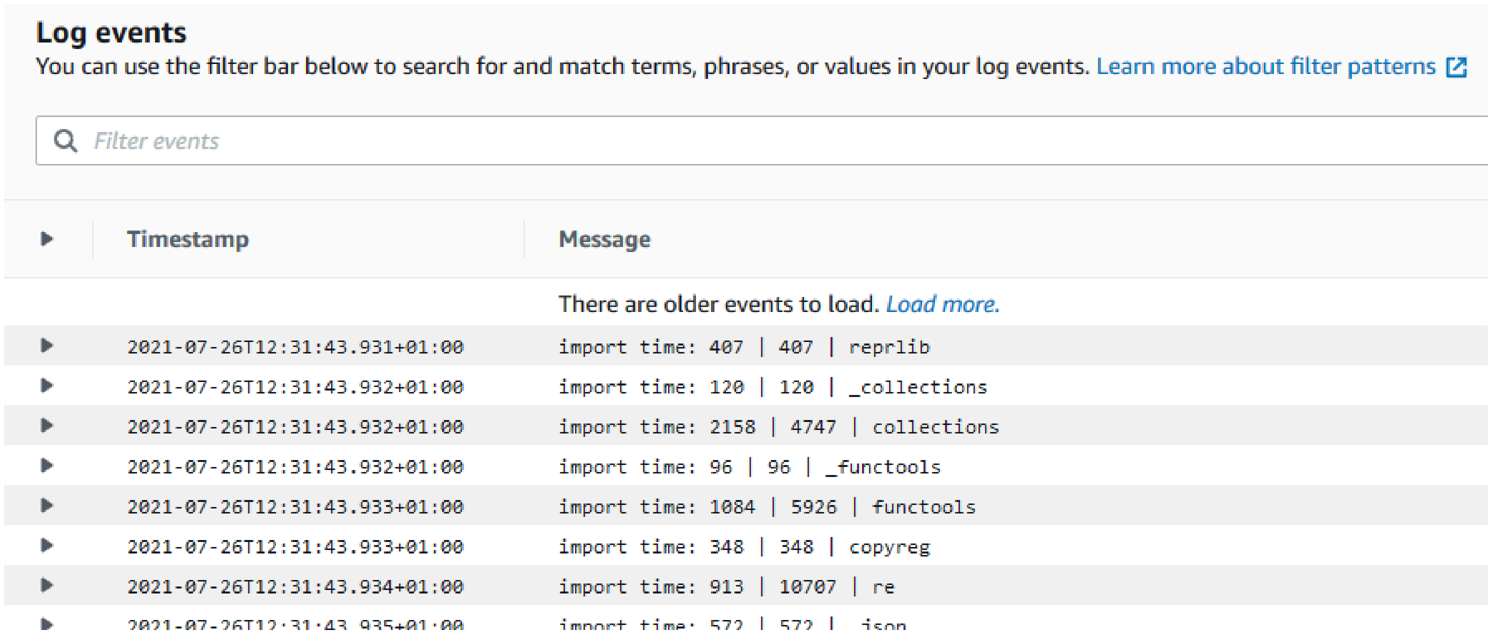
Environment variables (1)

The environment variables below are encrypted at rest with the default Lambda service key.

Key	Value
PYTHONPROFILEIMPORTTIME	1

PYTHONPROFILEIMPORTTIME environment variable

This shows how long each package import takes to help you understand how packages impact startup time.



Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

```
Timestamp                        Message
                                 There are older events to load. Load more.
2021-07-26T12:31:43.931+01:00    import time: 407 | 407 | reprlib
2021-07-26T12:31:43.932+01:00    import time: 120 | 120 | _collections
2021-07-26T12:31:43.932+01:00    import time: 2158 | 4747 | collections
2021-07-26T12:31:43.932+01:00    import time: 96 | 96 | _functools
2021-07-26T12:31:43.933+01:00    import time: 1084 | 5926 | functools
2021-07-26T12:31:43.933+01:00    import time: 348 | 348 | copyreg
2021-07-26T12:31:43.934+01:00    import time: 913 | 10707 | re
2021-07-26T12:31:43.935+01:00    import time: 572 | 572 |   json
```

Python import time

Previously, for the AWS Node.js SDK, you enabled HTTP keep-alive in your code to maintain TCP connections. Enabling keep-alive allows you to avoid setting up a new TCP connection for every request. Since AWS SDK version 2.463.0, you can also set the Lambda function environment variable `AWS_NODEJS_CONNECTION_REUSE_ENABLED=1` to make the SDK reuse connections by default.

You can configure Lambda's provisioned concurrency feature to pre-initialize a requested number of execution environments. This runs the cold start initialization code so that they are prepared to respond immediately to your function's invocations.

Use Amazon RDS Proxy to pool and share database connections to improve function performance. For additional options for using RDS with Lambda, see the AWS Serverless Hero blog post "How To: Manage RDS Connections from AWS Lambda Serverless Functions".

Choose frameworks that load quickly on function initialization startup. For example, prefer simpler Java dependency injection frameworks like Dagger or Guice over more complex framework such as Spring. When using the AWS SDK for Java, there are some cold start performance optimization suggestions in the documentation. For further Java performance optimization tips, see the AWS re:Invent session, "Best practices for AWS Lambda and Java".

To minimize deployment packages, choose lightweight web frameworks optimized for Lambda. For example, use MiddyJS, Lambda API JS, and Python Chalice over Node.js Express, Python Django or Flask.

If your function has many objects and connections, consider splitting the function into multiple, specialized functions. These are individually smaller and have less initialization code. I cover designing smaller, single purpose functions from a security perspective in "Managing application security boundaries – part 2".

Minimize your deployment package size to only its runtime necessities

Smaller functions also allow you to separate functionality. Only import the libraries and dependencies that are necessary for your application processing. Use code bundling when you can to reduce the impact of file system lookup calls. This also includes deployment package size.

For example, if you only use Amazon DynamoDB in the AWS SDK, instead of importing the entire SDK, you can import an individual service. Compare the following three examples as shown in the Lambda Operator Guide:

JavaScript
```javascript
// Instead of const AWS = require('aws-sdk'), use: +
const DynamoDB = require('aws-sdk/clients/dynamodb')

// Instead of const AWSXRay = require('aws-xray-sdk'), use: +
const AWSXRay = require('aws-xray-sdk-core')

// Instead of const AWS = AWSXRay.captureAWS(require('aws-sdk')), use: +
const dynamodb = new DynamoDB.DocumentClient() +
AWSXRay.captureAWSClient(dynamodb.service)
```

In testing, importing the DynamoDB library instead of the entire AWS SDK was 125 ms faster. Importing the X-Ray core library was 5 ms faster than the X-Ray SDK. Similarly, when wrapping a service initialization, preparing a `DocumentClient` before wrapping showed a 140-ms gain. Version 3 of the AWS SDK for JavaScript supports modular imports, which can further help reduce unused dependencies.

For additional options when for optimizing AWS Node.js SDK imports, see the AWS Serverless Hero blog post.

Conclusion

Evaluate and optimize your serverless application's performance based on access patterns, scaling mechanisms, and native integrations. You can improve your overall experience and make more efficient use of the platform in terms of both value and resources.

In this post, I cover measuring and optimizing function startup time. I explain cold and warm starts and how to reuse the Lambda execution environment to improve performance. I show a number of ways to analyze and optimize the initialization startup time. I explain how only importing necessary libraries and dependencies increases application performance.

This well-architected question continues in part 2 where I look at designing your function to take advantage of concurrency via asynchronous and stream-based invocations. I cover measuring, evaluating, and selecting optimal capacity units.

For more serverless learning resources, visit Serverless Land.
TAGS: serverless, well-architected