

Migrating AWS Lambda functions to Arm-based AWS Graviton2 processors

by Julian Wood | on 24 JAN 2022 | in [AWS Lambda](#), [Best Practices](#), [Graviton](#), [Serverless](#), [Technical How-To](#) | [Permalink](#) | [Comments](#) | [Share](#)

[AWS Lambda](#) now allows you to configure new and existing functions to run on Arm-based AWS Graviton2 processors in addition to x86-based functions. Using this processor architecture option allows you to get up to 34% better price performance. This blog post highlights some considerations when moving from x86 to arm64 as the migration process is code and workload dependent.

Functions using the Arm architecture benefit from the performance and security built into the Graviton2 processor, which is designed to deliver up to 19% better performance for compute-intensive workloads. Workloads using multithreading and multiprocessing, or performing many I/O operations, can experience lower invocation time, which reduces costs.

Duration charges, billed with [millisecond granularity](#), are 20 percent lower when compared to current x86 pricing. This also applies to duration charges when using [Provisioned Concurrency](#). Compute [Savings Plans](#) supports Lambda functions powered by Graviton2.

The architecture change does not affect the way your functions are invoked or how they communicate their responses back. Integrations with APIs, services, applications, or tools are not affected by the new architecture and continue to work as before.

The following runtimes, which use [Amazon Linux 2](#), are supported on Arm:

- Node.js 12 and 14
- Python 3.8 and 3.9
- Java 8 (java8.al2) and 11
- .NET Core 3.1
- Ruby 2.7
- [Custom runtime](#) (provided.al2)

[Lambda@Edge](#) does not support Arm as an architecture option.

You can create and manage Lambda functions powered by Graviton2 processor using the AWS Management Console, [AWS Command Line Interface \(AWS CLI\)](#), [AWS CloudFormation](#), [AWS Serverless Application Model \(AWS SAM\)](#), and [AWS Cloud Development Kit \(AWS CDK\)](#). Support is also available through many [AWS Lambda Partners](#).

Understanding Graviton2 processors

AWS Graviton processors are custom built by AWS. Generally, you don't need to know about the specific Graviton processor architecture, unless your applications can benefit from specific features.

The Graviton2 processor uses the Neoverse-N1 core and supports Arm V8.2 (include CRC and crypto extensions) plus several other architectural extensions. In particular, Graviton2 supports the Large System Extensions (LSE), which improve locking and synchronization performance across large systems.

Migrating x86 Lambda functions to arm64

Many Lambda functions may only need a configuration change to take advantage of the price/performance of Graviton2. Other functions may require repackaging the Lambda function using Arm-specific dependencies, or rebuilding the function binary or container image.

You may not require an Arm processor on your development machine to create Arm-based functions. You can build, test, package, compile, and deploy Arm Lambda functions on x86 machines using AWS SAM and [Docker Desktop](#). If you have an Arm-based system, such as an Apple M1 Mac, you can natively compile binaries.

Functions without architecture-specific dependencies or binaries

If your functions don't use architecture-specific dependencies or binaries, you can switch from one architecture to the other with a single configuration change. Many functions using interpreted languages such as Node.js and Python, or functions compiled to Java bytecode, can switch without any changes. Ensure you check binaries in dependencies, Lambda layers, and Lambda extensions.

To switch functions from x86 to arm64, you can change the Architecture within the function runtime settings using the Lambda console.

Edit runtime settings

Runtime settings [Info](#)

Runtime
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Python 3.9 ▼

Handler [Info](#)

app.lambda_handler

You can change either the function's runtime or the instruction set architecture in one update. To update both, you must repeat the update process.

Architecture [Info](#)
Choose the instruction set architecture you want for your function code.

☐ x86_64

☒ arm64

Cancel

Save

Edit AWS Lambda function Architecture

If you want to display or log the processor architecture from within a Lambda function, you can use OS specific calls. For example, Node.js `process.arch` or Python `platform.machine()`.

When using the AWS CLI to create a Lambda function, specify the `--architectures` option. If you do not specify the architecture, the default value is `x86-64`. For example, to create an arm64 function, specify

```
--architectures arm64 .
```

Bash

```
aws lambda create-function \
  --function-name MyArmFunction \
  --runtime nodejs14.x \
  --architectures arm64 \
  --memory-size 512 \
  --zip-file fileb://MyArmFunction.zip \
  --handler lambda.handler \
  --role arn:aws:iam::123456789012:role/service-role/MyArmFunction-role
```

When using AWS SAM or CloudFormation, add or amend the `Architectures` property within the function configuration.

YAML

```
MyArmFunction:
  Type: AWS::Lambda::Function
  Properties:
    Runtime: nodejs14.x
    Code: src/
    Architectures:
      - arm64
    Handler: lambda.handler
    MemorySize: 512
```

When initiating an AWS SAM application, you can specify:

Bash

```
sam init --architecture arm64
```

When building Lambda layers, you can specify `CompatibleArchitectures`.

YAML

```
MyArmLayer:
  Type: AWS::Lambda::LayerVersion
  Properties:
    ContentUri: layersrc/
    CompatibleArchitectures:
      - arm64
```

Building function code for Graviton2

If you have dependencies or binaries in your function packages, you must rebuild the function code for the architecture you want to use. Many packages and dependencies have arm64 equivalent versions. Test your own workloads against arm64 packages to see if your workloads are good migration candidates. Not all workloads show improved performance due to the different processor architecture features.

For compiled languages like Rust and Go, you can use the `provided.al2` custom runtime, which supports Arm. You provide a binary that communicates with the [Lambda Runtime API](#).

When compiling for Go, set `GOOS` to `arm`.

```
GOOS=linux GOARCH=arm64 go build
```

When compiling for Rust, set the target.

```
cargo build --release -- target-cpu=neoverse-n1
```

The default installation of Python `pip` on some Linux distributions is out of date (<19.3). To install binary wheel packages released for Graviton, upgrade the pip installation using:

```
Bash
sudo python3 -m pip install --upgrade pip
```

The Arm software ecosystem is continually improving. As a general rule, use later versions of compilers and language runtimes whenever possible. The [AWS Graviton Getting Started GitHub repository](#) includes known recent changes to popular packages that improve performance, including ffmpeg, PHP, .Net, PyTorch, and zlib.

You can use <https://pkgs.org/> as a package repository search tool.

Sometimes code includes architecture specific optimizations. These can include code optimized in assembly using specific instructions for `CRC`, or enabling a `feature` that works well on particular architectures. One way to see if any optimizations are missing for arm64 is to search the code for `__x86_64__` `ifdef` s and see if there is corresponding arm64 code included. If not, consider alternative solutions.

For additional language-specific considerations, see the links within the [GitHub repository](#).

- [C/C++](#)
- [Go](#)
- [Java](#)
- [.NET](#)
- [Python](#)
- [Rust](#)

The [Graviton performance runbook](#) is a performance profiling reference by the Graviton to benchmark, debug, and optimize application code.

Building functions packages as container images

Functions packaged as container images must be built for the architecture (x86 or arm64) they are going to use. There are arm64 architecture versions of the [AWS provided base images for Lambda](#). To specify a container image for arm64, use the arm64 specific image tag, for example, for Node.js 14:

- public.ecr.aws/lambda/nodejs:14-arm64
- public.ecr.aws/lambda/nodejs:latest-arm64
- public.ecr.aws/lambda/nodejs:14.2021.10.01.16-arm64

Arm64 Images are also available from [Docker Hub](#).

You can also use arbitrary Linux base images in addition to the AWS provided Amazon Linux 2 images. Images that support arm64 include [Alpine](#) Linux 3.12.7 or later, [Debian](#) 10 and 11, Ubuntu 18.04 and 20.04. For more information and details of other supported Linux versions, see [Operating systems available for Graviton based instances](#).

Migrating a function

Here is an example of how to migrate a Lambda function from x86 to arm64 and take advantage of newer software versions to improve price and performance. You can follow a similar approach to test your own code.

I have an existing Lambda function as part of an AWS SAM template configured without an `Architectures` property, which defaults to `x86_64`.

YAML

```
Imagex86Function:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: src/
    Handler: app.lambda_handler
    Runtime: python3.9
```

The Lambda function code performs some compute intensive image manipulation. The code uses a dependency configured with the following version:

JSON

```
{
  "dependencies": {
    "imagechange": "^1.1.1"
  }
}
```

I duplicate the Lambda function within the AWS SAM template using the same source code and specify `arm64` as the `Architectures`.

YAML

```
ImageArm64Function:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: src/
    Handler: app.lambda_handler
    Runtime: python3.9
```

Architectures:

- arm64

I use AWS SAM to build both Lambda functions. I specify the `--use-container` flag to build each function within its architecture-specific build container.

Bash

```
sam build --use-container
```

I can use `sam local invoke` to test the arm64 function locally even on an x86 system.

```
$ sam local invoke TestArmFunction

Invoking app.lambda_handler (python3.9)
Image was not found.
Removing rapid images for repo public.ecr.aws/sam/emulation-python3.9
Building image.....
.....
Skip pulling image and use local one: public.ecr.aws/sam/emulation-python3.9:rapid-1.36.0-arm64.

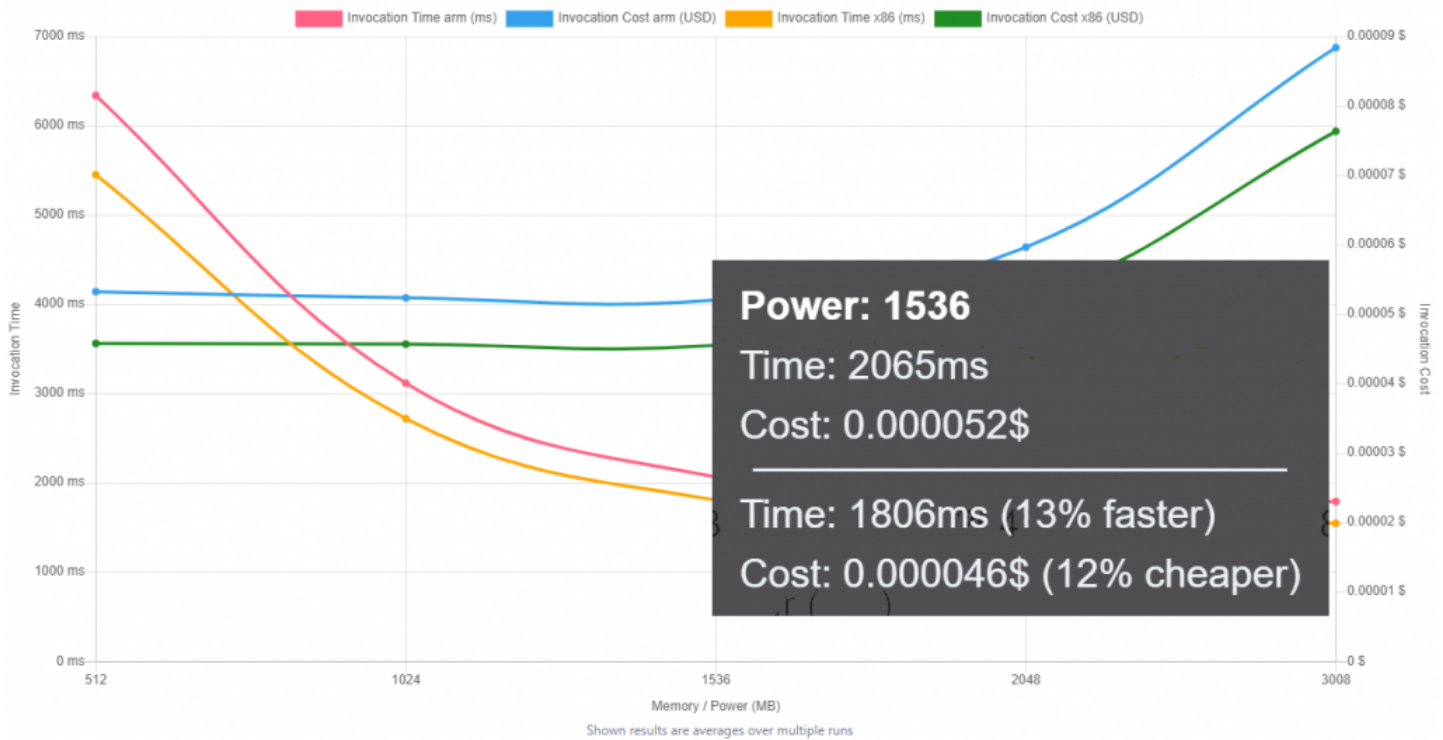
Mounting /mnt/c/GitProjects/Graviton/graviton-test/.aws-sam/build/TestArmFunction as /var/task:ro,delegated inside runtime container
START RequestId: 2a755d84-3b95-40fd-9e24-3bf539da70c4 Version: $LATEST
END RequestId: 2a755d84-3b95-40fd-9e24-3bf539da70c4
REPORT RequestId: 2a755d84-3b95-40fd-9e24-3bf539da70c4  Init Duration: 3.09 ms  Duration: 1146.40 ms  Billed Duration: 1147 ms  Memory Size: 128 MB  Max Memory Used: 128 MB
{"Platform architecture:": "aarch64"}%
```

AWS SAM local invoke

I then use `sam deploy` to deploy the functions to the AWS Cloud.

The [AWS Lambda Power Tuning](#) open-source project runs your functions using different settings to suggest a configuration to minimize costs and maximize performance. The tool [allows you to compare two results on the same chart](#) and incorporate arm64-based pricing. This is useful to compare two versions of the same function, one using x86 and the other arm64.

I compare the performance of the X86 and arm64 Lambda functions and see that the arm64 Lambda function is 12% cheaper to run:



Compare x86 and arm64 with dependency version 1.1.1

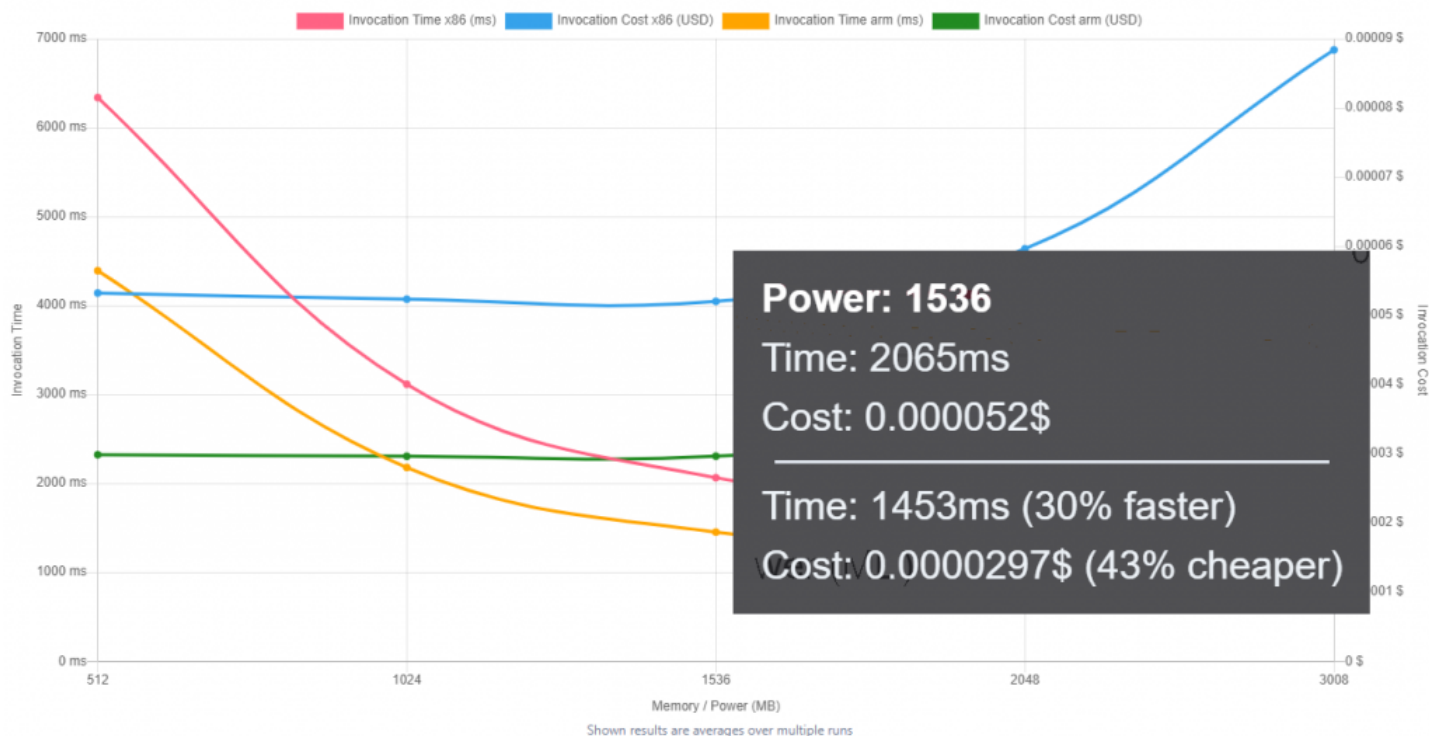
I then upgrade the package dependency to use version 1.2.1, which has been optimized for arm64 processors.

JSON

```
{
  "dependencies": {
    "imagechange": "^1.2.1"
  }
}
```

I use `sam build` and `sam deploy` to redeploy the updated Lambda functions with the updated dependencies.

I compare the original x86 function with the updated arm64 function. Using arm64 with a newer dependency code version increases the performance by 30% and reduces the cost by 43%.



Compare x86 and arm64 with dependency version 1.2.1

You can use [Amazon CloudWatch](#), to view performance metrics such as duration, using [statistics](#). You can then compare average and [p99](#) duration between the two architectures. Due to the Graviton2 architecture, functions may be able to use less memory. This could allow you to right-size function memory configuration, which also reduces costs.

Deploying arm64 functions in production

Once you have confirmed your Lambda function performs successfully on arm64, you can migrate your workloads. You can use function versions and aliases with weighted aliases to control the rollout. Traffic gradually shifts to the arm64 version or rolls back automatically if any specified CloudWatch alarms trigger.

AWS SAM supports [gradual Lambda deployments](#) with a feature called [Safe Lambda deployments](#) using [AWS CodeDeploy](#). You can compile package binaries for arm64 using a number of CI/CD systems. [AWS CodeBuild](#) supports [building Arm based applications](#) natively. [CircleCI](#) also has [Arm compute resource classes](#) for deployment. [GitHub Actions](#) allows you to use [self-hosted runners](#). You can also use [AWS SAM within GitHub Actions](#) and other CI/CD pipelines to create arm64 artifacts.

Conclusion

Lambda functions using the Arm/Graviton2 architecture provide up to 34 percent price performance improvement. This blog discusses a number of considerations to help you migrate functions to arm64.

Many functions can migrate seamlessly with a configuration change, others need to be rebuilt to use arm64 packages. I show how to migrate a function and how updating software to newer versions may improve your function performance on arm64. You can test your own functions using the Lambda PowerTuning tool.

Start migrating your Lambda functions to Arm/Graviton2 today.

For more serverless learning resources, visit [Serverless Land](#).

TAGS: [serverless](#)