



Adopting Java for the Serverless world

from the perspective of the **AWS** developer



Vadym Kazulkin, ip.labs, Jug Saxony Day , 23 September 2022



Contact



Vadym Kazulkin

ip.labs GmbH Bonn, Germany



Co-Organizer of the Java User Group Bonn



v.kazulkin@gmail.com



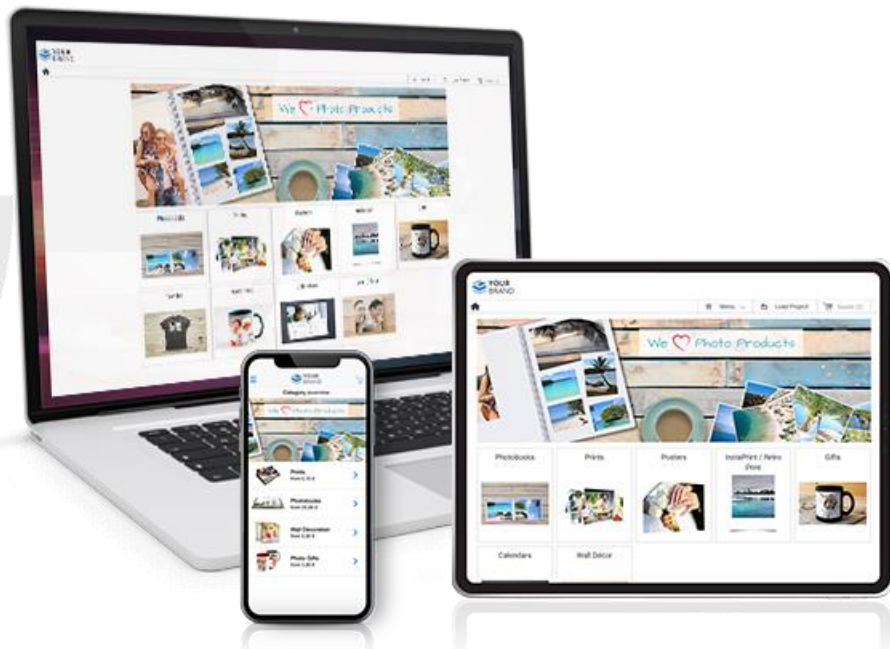
<https://www.linkedin.com/in/vadymkazulkin>



@VKazulkin



ip.labs



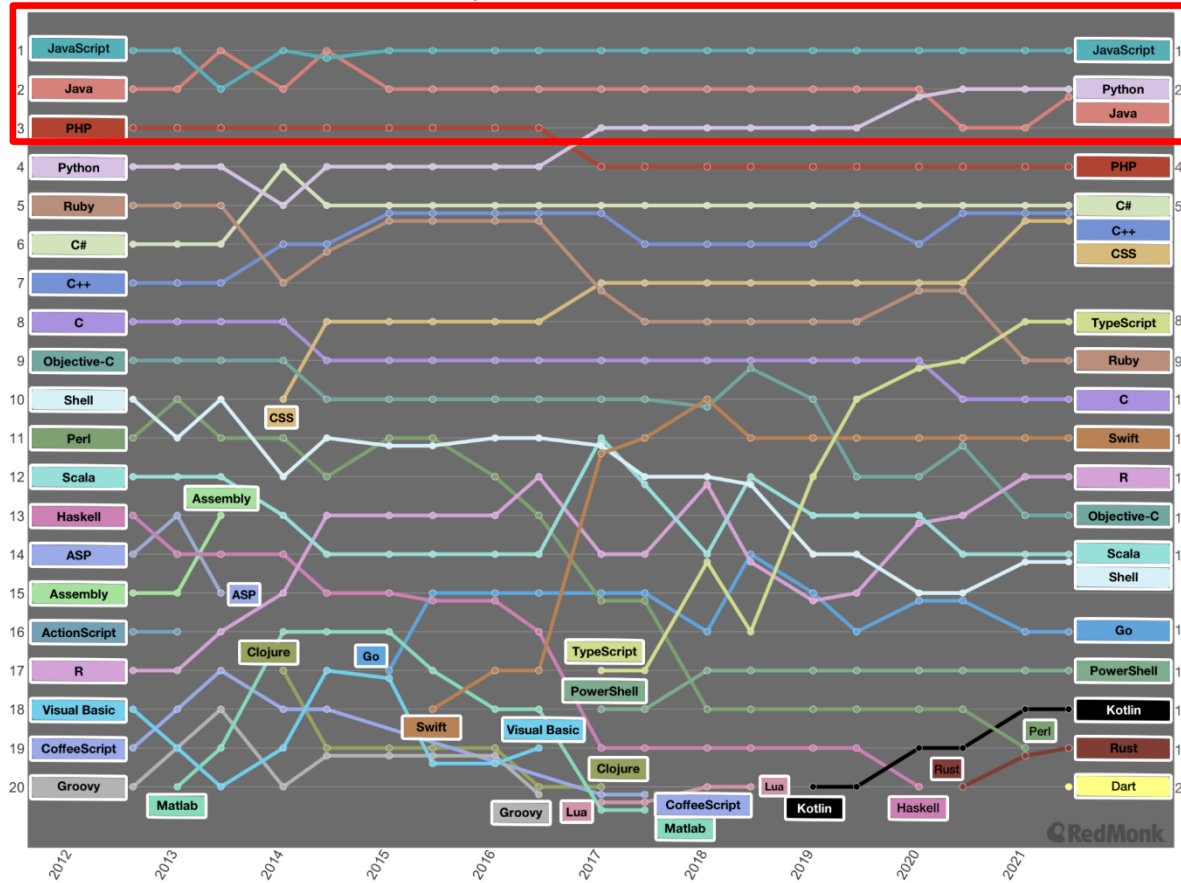


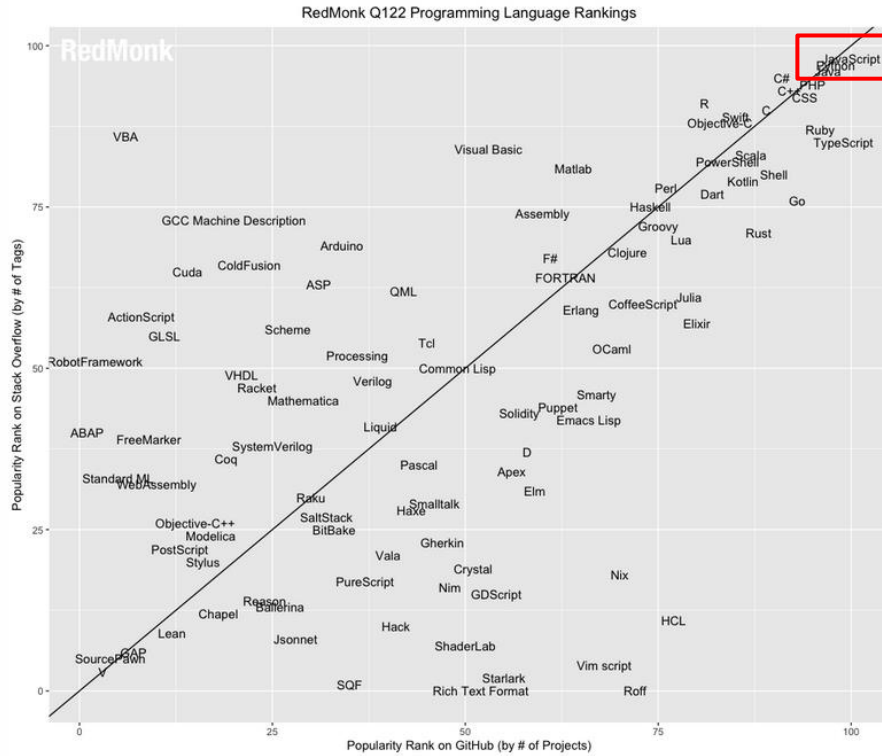
Java popularity



RedMonk Language Rankings

September 2012 - June 2021





- 1 JavaScript
- 2 Python
- 3 Java
- 4 PHP
- 5 CSS



AWS and Serverless



Figure 1. Magic Quadrant for Cloud Infrastructure and Platform Services



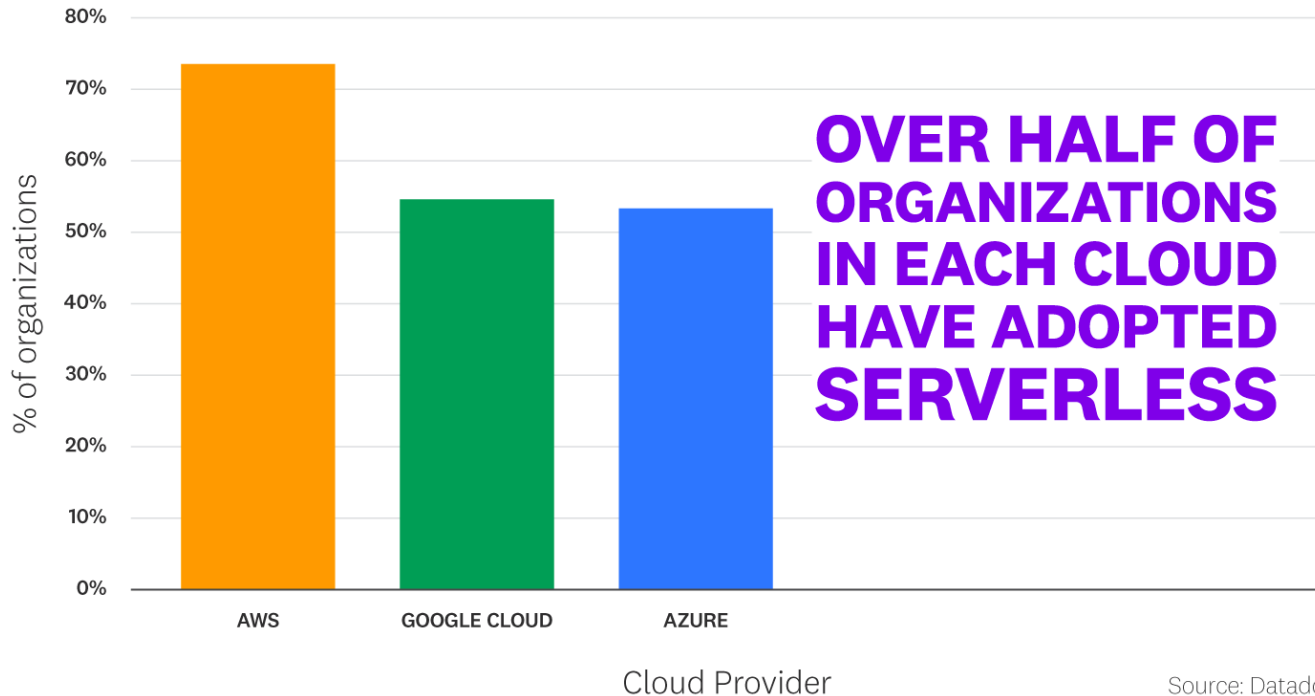
Figure 1: Magic Quadrant for Cloud Infrastructure and Platform Services



Source: Gartner (July 2021)



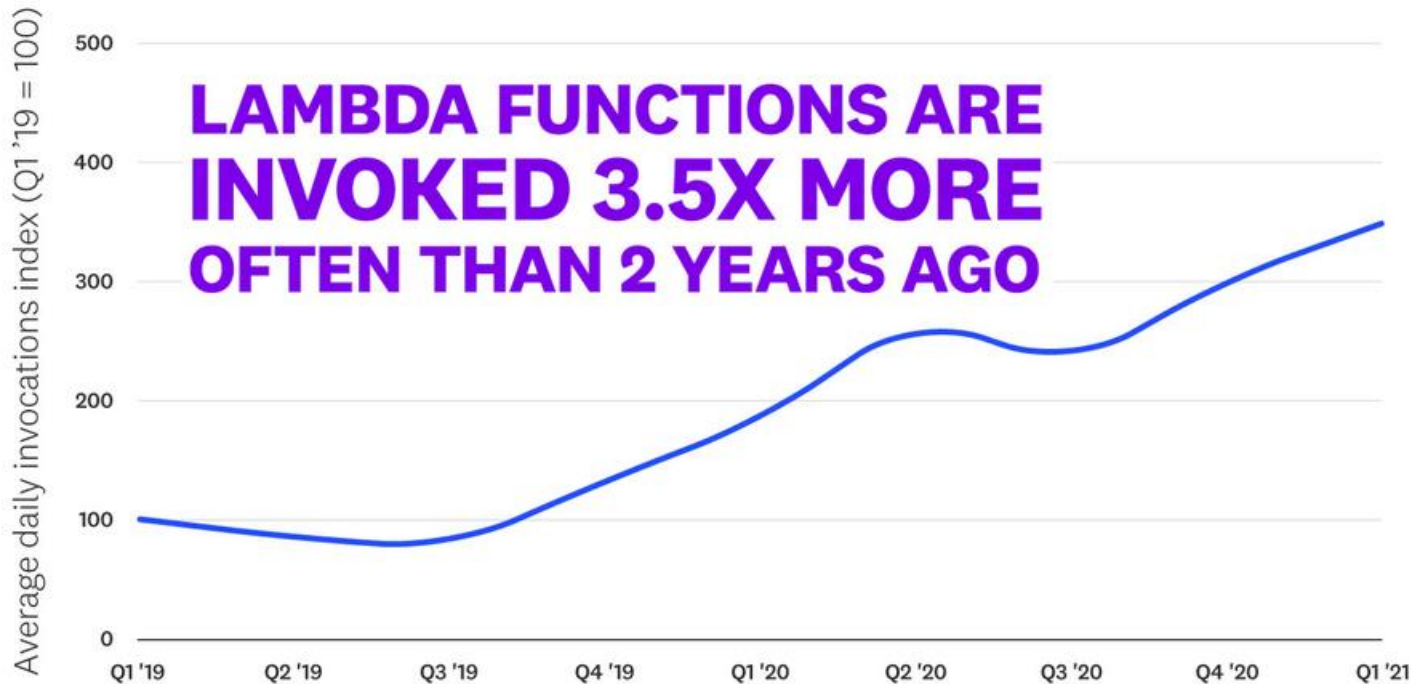
Serverless adoption by cloud provider



Source: Datadog



Average Daily Invocations per Lambda Function Index



Source: Datadog



Life of the Java (Serverless) developer on AWS



AWS Java Versions Support

- Java 8
 - With extended long-term support
- Java 11 (since 2019)
- Only Long Term Support (LTS) by AWS
- Current LTS Java version is Java 17
 - Amazon Corretto Support for 17 is released, but not currently available for AWS Lambda

Amazon Corretto

No-cost, multiplatform, production-ready distribution of OpenJDK

Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). Corretto comes with long-term support that will include performance enhancements and security fixes. Amazon runs Corretto internally on thousands of production services and Corretto is certified as compatible with the Java SE standard. With Corretto, you can develop and run Java applications on popular operating systems, including Linux, Windows, and macOS.



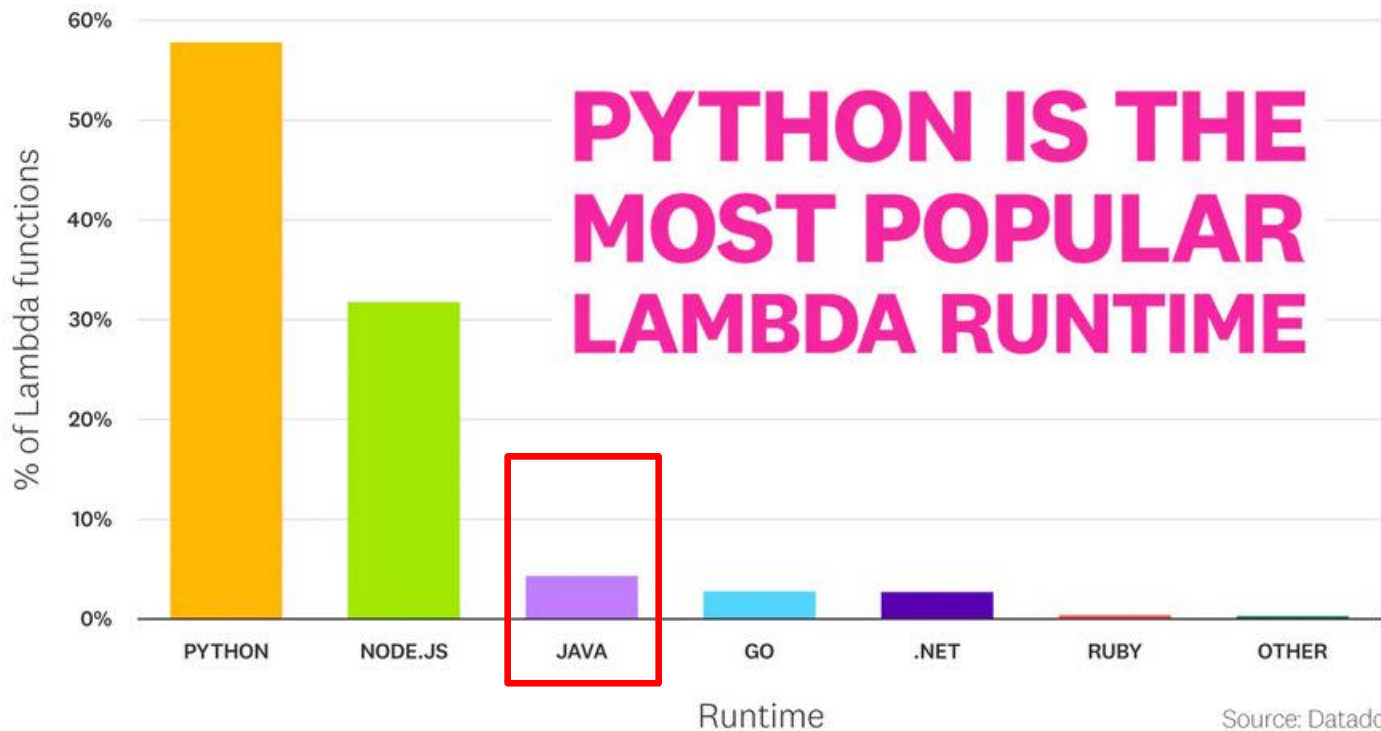
Java ist very fast
and mature
programming
language...

... but Serverless
adoption of Java
looks like this



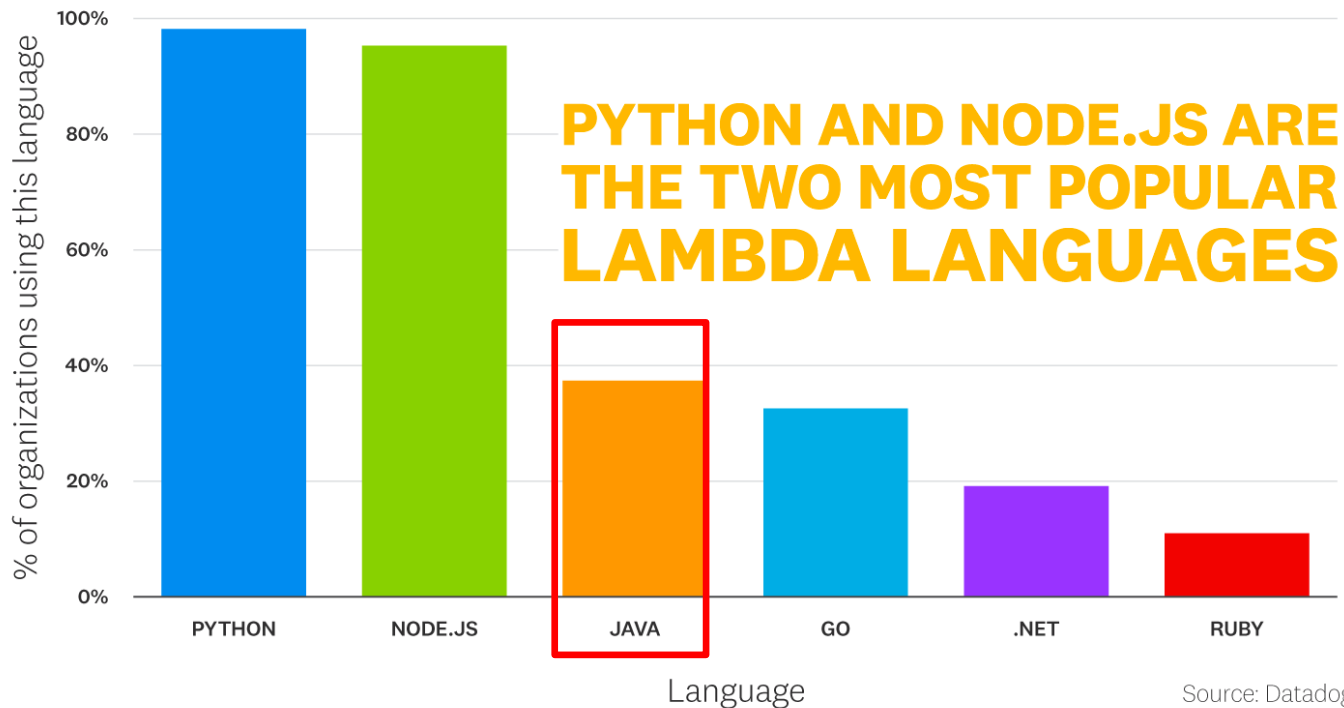


Most Popular Runtimes by Distinct Functions





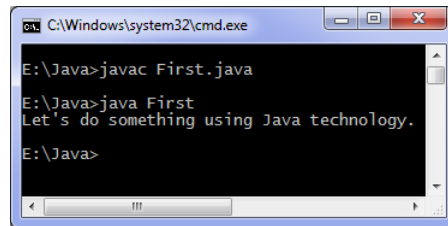
Top languages used in Lambda functions





Developers love Java and will be happy
to use it for Serverless

But what are the challenges ?



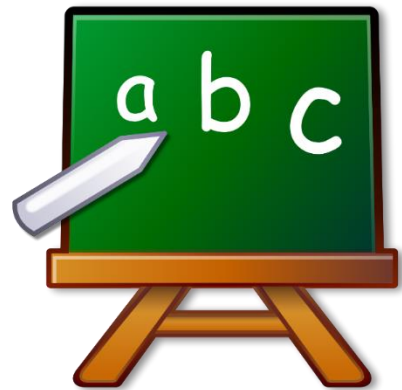
```
C:\Windows\system32\cmd.exe
E:\Java>javac First.java
E:\Java>java First
Let's do something using Java technology.
E:\Java>
```





Serverless with Java challenges

- “cold start” times (latencies)
- memory footprint (high cost in AWS)



AWS Lambda Basics



Creating AWS Lambda with Java 1/3

Basic information

Function name

Enter a name that describes the purpose of your function.

MyFirstJavaFunction

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime [Info](#)

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

Java 11 (Corretto)

Architecture [Info](#)

Choose the instruction set architecture you want for your function code.

x86_64

arm64

Permissions [Info](#)

By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▼ Change default execution role

Execution role

Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions

Use an existing role

Create a new role from AWS policy templates

Latest supported

.NET 6 (C#/PowerShell)

.NET Core 3.1 (C#/PowerShell)

Go 1.x

Java 11 (Corretto)

Node.js 14.x

Python 3.9

Ruby 2.7

Other supported

Java 8 on Amazon Linux 1

Java 8 on Amazon Linux 2

Node.js 12.x

Java 11 (Corretto)

Basic settings

Description

Memory (MB) [Info](#)

Your function is allocated CPU proportional to the memory configured.



Timeout [Info](#)

0 min 3 sec

Full CPU access only
approx. at 1.8 GB
memory allocated



Creating AWS Lambda with Java 2/3

```
import javax.inject.Inject;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class MonthlyInvoiceGeneratorFunction
implements RequestHandler<MonthlyInvoiceRequest, MonthlyInvoiceResponse> {

    private static final Logger LOG = LoggerFactory.getLogger(MonthlyInvoiceGeneratorFunction.class);

    @Inject
    private MonthlyInvoiceGeneratorService monthlyInvoiceGeneratorService;

    @Override
    public MonthlyInvoiceResponse handleRequest(MonthlyInvoiceRequest monthlyInvoiceRequest,
        final Context context) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("request: {}", monthlyInvoiceRequest);
        }
        return this.monthlyInvoiceGeneratorService.generateInvoice(monthlyInvoiceRequest);
    }
}
```

```
import java.util.function.Function;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;

public class MonthlyInvoiceGeneratorFunction
implements Function<MonthlyInvoiceRequest, MonthlyInvoiceResponse> {

    private static final Logger LOG = LoggerFactory.getLogger(MonthlyInvoiceGeneratorFunction.class);

    @Autowired
    private MonthlyInvoiceGeneratorService monthlyInvoiceGeneratorService;

    @Override
    public MonthlyInvoiceResponse apply(MonthlyInvoiceRequest monthlyInvoiceRequest) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("request: {}", monthlyInvoiceRequest);
        }
        return this.monthlyInvoiceGeneratorService.generateInvoice(monthlyInvoiceRequest);
    }
}
```



Creating AWS Lambda with Java 3/3

AWS Lambda context object in Java

[PDF](#) | [Kindle](#) | [RSS](#)

When Lambda runs your function, it passes a context object to the [handler](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- `getFunctionName()` – Returns the name of the Lambda function.
- `getFunctionVersion()` – Returns the [version](#) of the function.
- `getInvokedFunctionArn()` – Returns the Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `getMemoryLimitInMB()` – Returns the amount of memory that's allocated for the function.
- `getAwsRequestId()` – Returns the identifier of the invocation request.
- `getLogGroupName()` – Returns the log group for the function.
- `getLogStreamName()` – Returns the log stream for the function instance.
- `getIdentity()` – (mobile apps) Returns information about the Amazon Cognito identity that authorized the request.
- `getClientContext()` – (mobile apps) Returns the client context that's provided to Lambda by the client application.
- `getLogger()` – Returns the [logger object](#) for the function.



AWS Lambda Price Model



Cost for Lambda



REQUEST



DURATION



Request Tier

\$ 0.20

Per 1 Mio Requests



Duration Tier

\$ 0.00001667 (x86)

\$ 0.00001333 (Arm)

Per GB-Second



GB-Second



ONE SECOND



ONE GB



Example

- 1 Mio requests
- Lambda x86 with 512MiB
- Each lambda takes 200ms

$0.5 \text{ GiB} * 0.2 \text{ sec} * 1 \text{ Mio}$
= 100 000 GB-Seconds



Requests:
\$0.20



GB-Seconds:
\$1.67

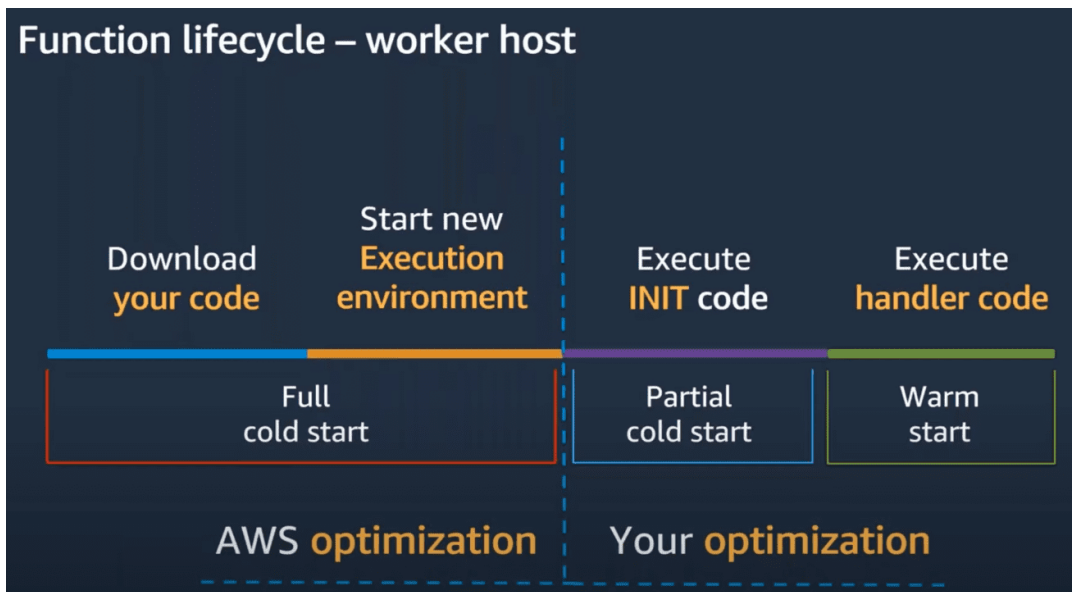


Challenge Number 1 with Java is a
big **cold-start**



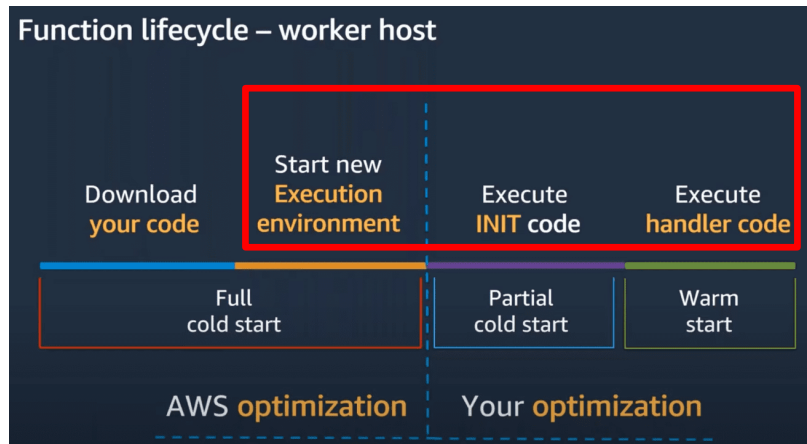


Cold Start





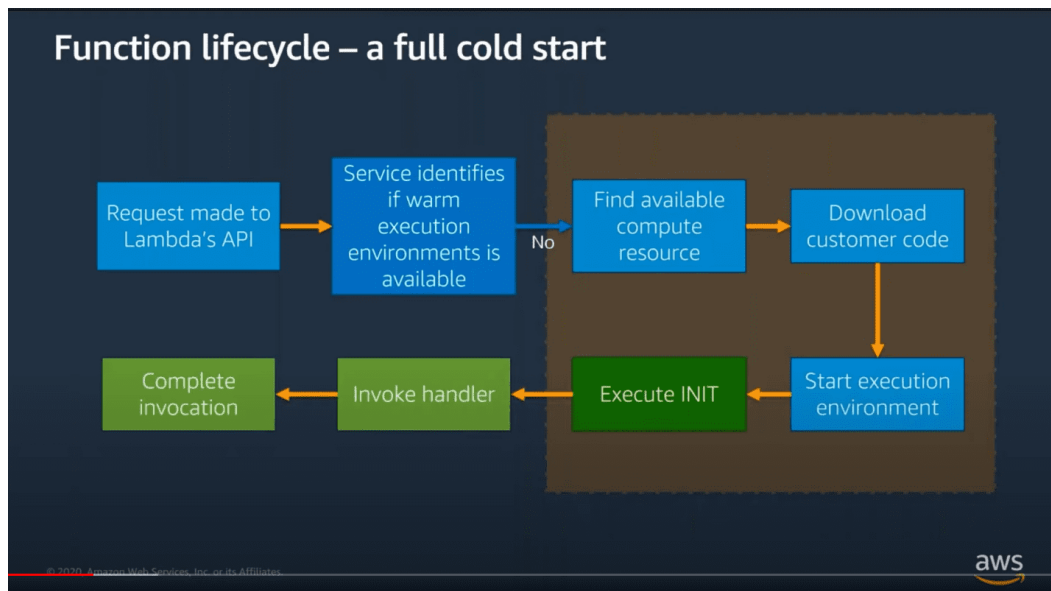
- Start Firecracker VM
- AWS Lambda starts the JVM
- Java runtime loads and initializes handler class
 - Static initializer block of the handler class is executed
 - Init-phase has **full CPU access up to 10 seconds for free** *for the managed execution environments*
- Lambda calls the handler method



The screenshot shows the 'Basic settings' configuration page for a Lambda function. The 'Memory (MB)' field is highlighted with a red box, showing a value of 128 MB. The 'Timeout' field is set to 3 seconds. The 'Description' field is empty.



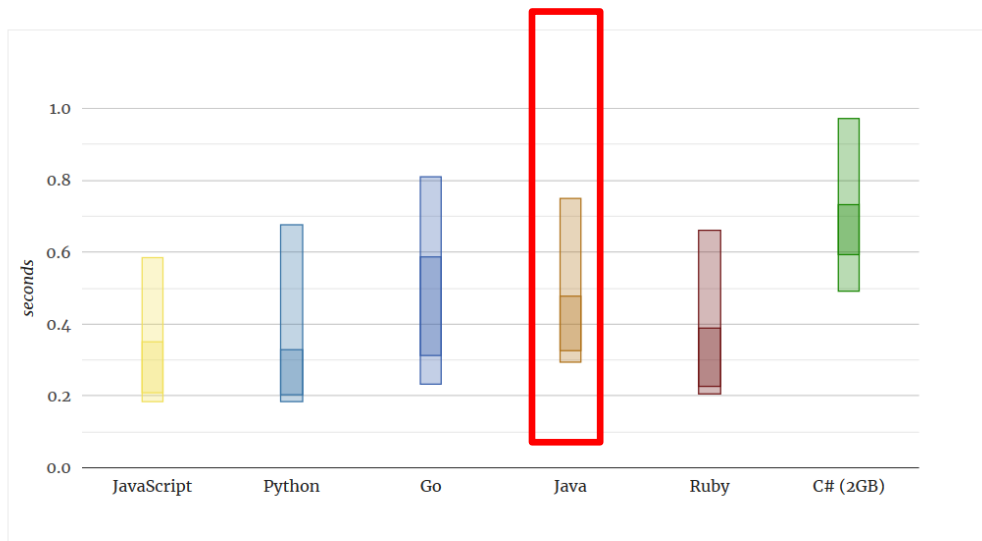
Function lifecycle- a full cold start





AWS Lambda cold start duration per programming language

The following chart shows the typical range of cold starts in AWS Lambda, broken down per language. The darker ranges are the most common 67% of durations, and lighter ranges include 95%.



Typical cold start durations per language



Cold start duration with Java

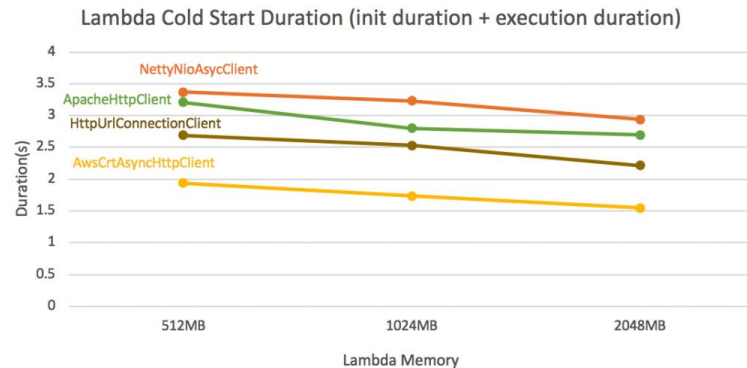
- Below 1 second is best-case cold start duration for very simple Lambda like HelloWorld with no dependencies
- It goes up significantly with more complex scenarios
 - Instantiation outside of the handler method (static instantiation) to communicate with other (AWS) services (i.e. DynamoDB, SNS, SQS, 3rd party)
 - Including more dependencies
- To minimize the cold start time apply best practices from this talk
 - Worst-case cold starts can be higher than 10 and even 20 seconds



Best Practices and Recommendations

- Switch to the AWS SDK 2.0 for Java
 - Lower footprint and more modular
 - Allows to configure HTTP Client of your choice (i.e. Java own Basic HTTP Client or *newly introduced AWS Common Runtime async HTTP Client*)

```
S3AsyncClient.builder()  
.httpClientBuilder(AwsCrtAsyncHttpClient.builder()  
.maxConcurrency(50))  
.build();
```





Best Practices and Recommendations

- Less (dependencies, classes) is more
 - Include only required dependencies (e.g. not the whole AWS SDK 2.0 for Java, but the dependencies to the clients to be used in Lambda)
 - Exclude dependencies, which you don't need at runtime e.g. test frameworks like Junit

```
<dependency>
```

```
<groupId>software.amazon.awssdk</groupId>
```

```
<artifactId>bom</artifactId>
```

```
<version>2.10.86</version>
```

```
<type>pom</type>
```

```
<scope>import</scope>
```

```
</dependency>
```

```
<dependency>
```

```
<groupId>software.amazon.awssdk</groupId>
```

```
<artifactId>dynamodb</artifactId>
```

```
<version>2.10.86</version>
```

```
</dependency>
```

```
<dependency>
```

```
<groupId>org.junit.jupiter</groupId>
```

```
<artifactId>junit-jupiter-api</artifactId>
```

```
<version>5.4.2</version>
```

```
<scope>test</scope>
```

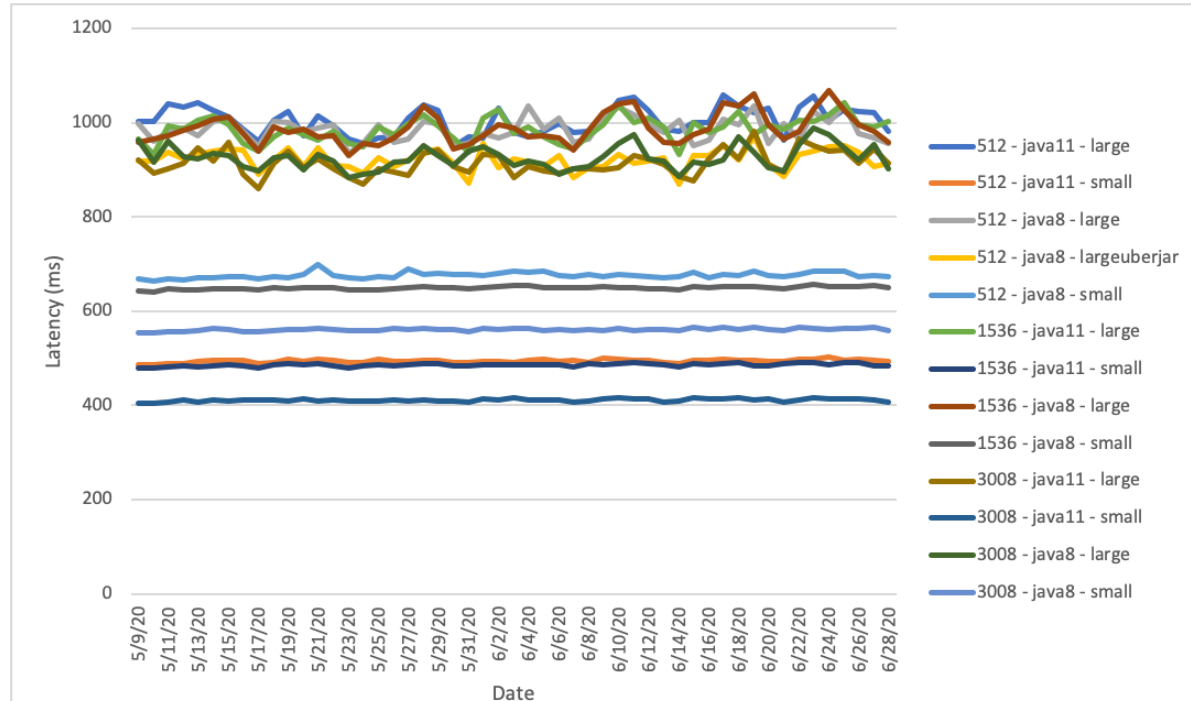
```
</dependency>
```



AWS Lambda cold starts by memory size, runtime and artifact size

Artifact Size:

- Small zip (1KB)
- Large zip (48MB)
- Large uberjar (53MB)





Best Practices and Recommendations

- Initialize dependencies during initialization phase
 - Use static initialization in the handler class, instead of in the handler method (e.g. `handleRequest`) to take the advantage of the access to the full CPU core for max 10 seconds
 - In case of DynamoDB client put the following code outside of the handler method:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()...build();  
DynamoDB dynamoDB = new DynamoDB(client);
```



Best Practices and Recommendations

Provide all known values (for building clients i.e. DynamoDB client) to avoid auto-discovery

- credential provider, region, endpoint

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
.withRegion(Regions.US_WEST_2)  
.withCredentials(new ProfileCredentialsProvider("myProfile"))  
  
.build();
```



Best Practices and Recommendations Using Tiered Compilation

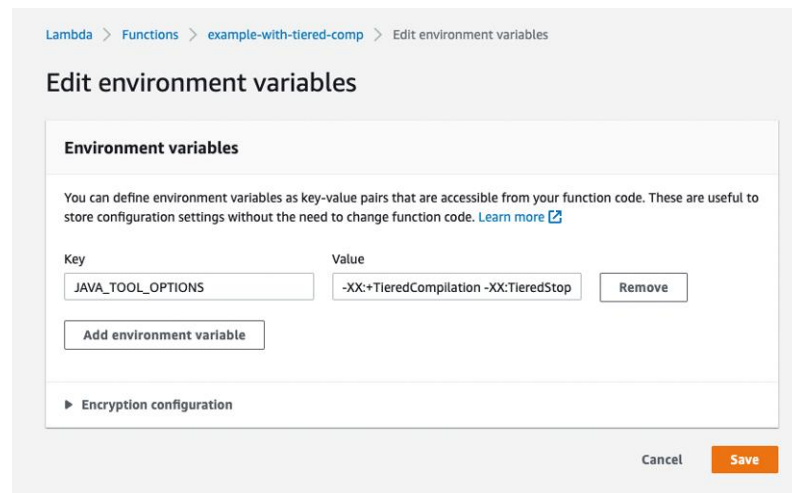
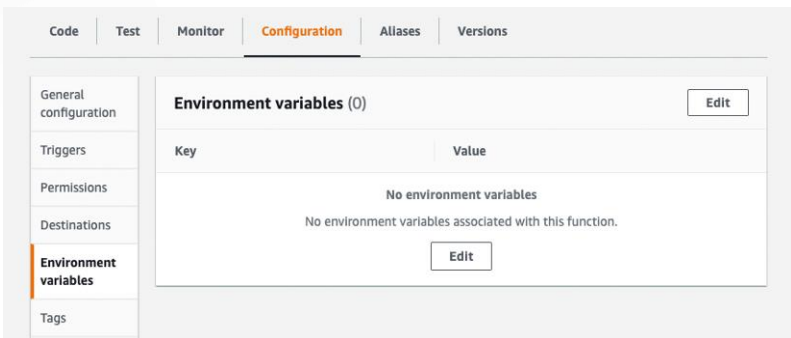
Achieve **up to 60%** faster startup times can use **level 1** compilation with little risk of reducing warm start performance

| |
|---------------------------------|
| Level 4 - C2 |
| Level 3 - C1 w/ full profiling |
| Level 2 - C1 w/ basic profiling |
| Level 1 - C1 w/o profiling |
| Level 0 - Interpreter |

Choose **Add environment variable**. Add the following:

Bash

- Key: JAVA_TOOL_OPTIONS
- Value: -XX:+TieredCompilation -XX:TieredStopAtLevel=1

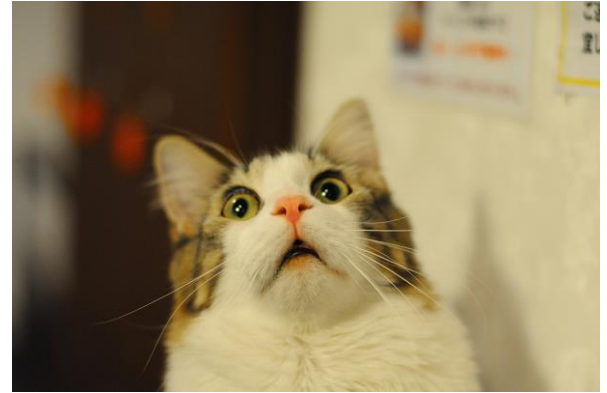




Best Practices and Recommendations

Avoid:

- reflection
- runtime byte code generation
- runtime generated proxies
- dynamic class loading



Use DI Frameworks which aren't reflection-based



Best Practices and Recommendations

Cost optimization techniques





Cost for Lambda



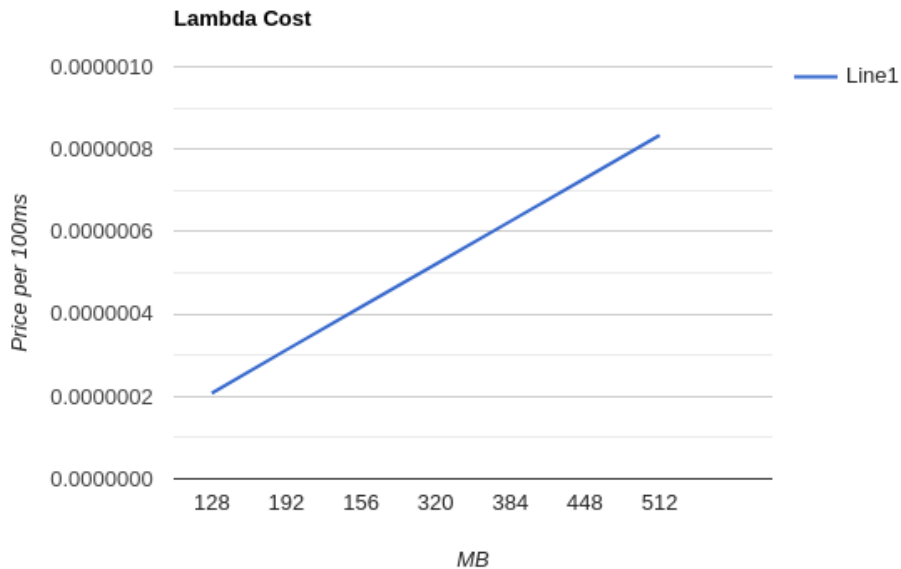
REQUEST



DURATION



Cost scales linearly with memory





More memory = more expensive?

Basic settings

Description

Memory (MB) [Info](#)

Your function is allocated CPU proportional to the memory configured.



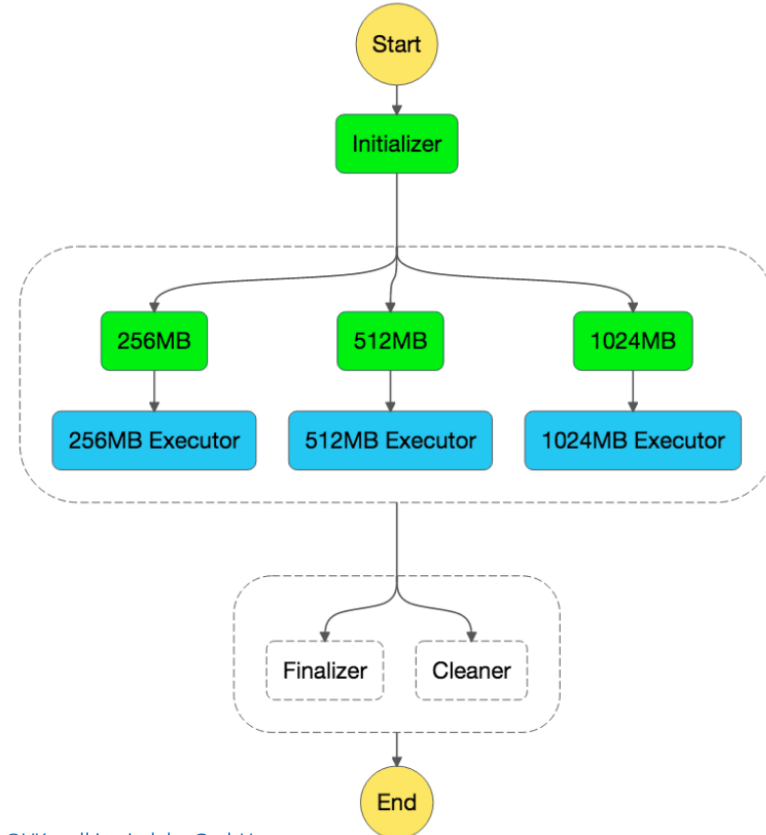
Timeout [Info](#)

min sec



Lambda Power Tuning 1/2

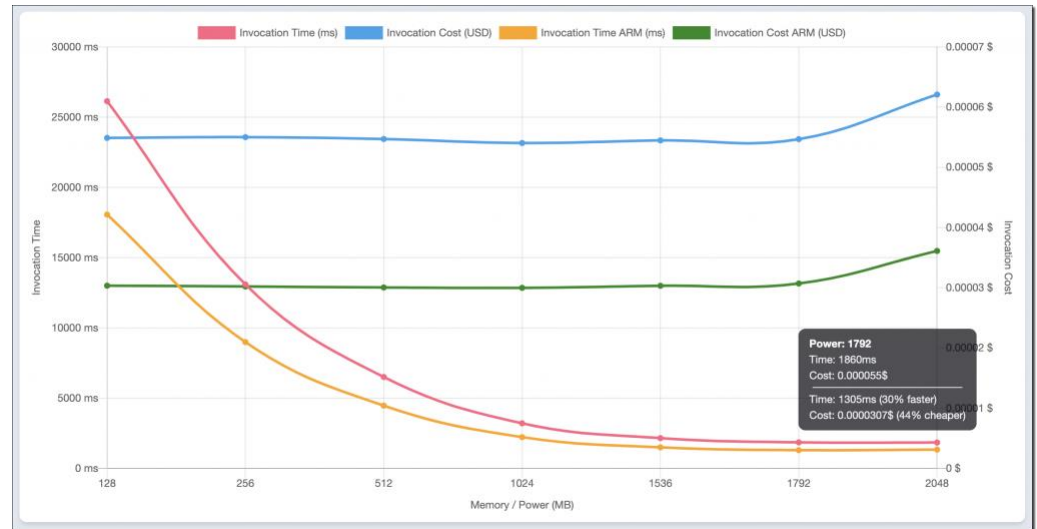
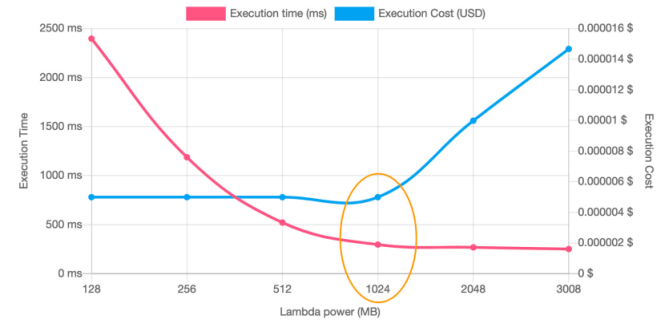
- Executes different settings in parallel
- Outputs the optimal setting





Lambda Power Tuning 2/2

- Executes different settings in parallel
- Outputs the optimal setting





Optimizing AWS Lambda cost and performance using AWS Compute Optimizer

Recommendations for Lambda functions (2) [Info](#)
Recommendations for current resources to improve cost and performance. Action View details

Filter by one or more Regions Memory over-provisioned < 1 ... > ⚙️

Region: US East (N. Virginia) × Clear filters

| Function name | Function version | Finding | Finding reason | Current configured memory | Recommended configured memory |
|----------------------------------|------------------|---------------|-------------------------|---------------------------|-------------------------------|
| lambda-recommendation-test-sleep | \$LATEST | Not optimized | Memory over-provisioned | 1024 MB | 900 MB |

[AWS Compute Optimizer](#) > [Dashboard](#) > [Recommendations for Lambda functions](#) > [lambda-recommendation-test-sleep details](#) Open in Lambda Console

lambda-recommendation-test-sleep details

Function version: \$LATEST

Compare current configured memory with recommended options [Info](#)
Consider an alternate memory configuration for the Lambda function.

| Options | Configured memory | Cost difference (%) | Used memory (maximum) | Duration (average) | Projected duration (expected) |
|---|-------------------|---------------------|-----------------------|----------------------|-------------------------------|
| <input type="radio"/> Current | 1024 MB | - | 819.0 MB | 31333.6 milliseconds | - |
| <input checked="" type="radio"/> Option 1 | 900 MB | -15.7% ~ -7.1% | - | - | 31515.9 milliseconds |

```
Bash
$ aws compute-optimizer \
  get-lambda-function-recommendations \
  --function-arns arn:aws:lambda:us-east-1:123456789012:function:lambda-recommendation-test-sleep

JSON
{
  "lambdaFunctionRecommendations": [
    {
      "utilizationMetrics": [
        {
          "name": "Duration",
          "value": 31333.63587049883,
          "statistic": "Average"
        },
        {
          "name": "Duration",
          "value": 32522.04,
          "statistic": "Maximum"
        },
        {
          "name": "Memory",
          "value": 817.67049838188,
          "statistic": "Average"
        }
      ]
    }
  ]
}
```



Cost optimization

- Java is well optimized for long running server applications
 - High startup times
 - High memory utilization

Even with all optimization applied we'll be left with seconds of the colds starts and high memory utilization



GraalVM enters the scene

GraalVM™



GraalVM

Goals:

Low footprint ahead-of-time mode for JVM-based languages

High performance for all languages

Convenient language interoperability and polyglot tooling



Community Edition

GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on [GitHub](#). We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is [experimental](#).

DOWNLOAD FROM GITHUB

LICENSE

- [Open Source Licenses](#)
- Free for development and production use

BENEFITS

- Open-source license
- Free community support via [public channels](#)
- Presence of all enterprise components
- Bug fixes and enhancements

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the [Oracle Technology Network](#). We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is [experimental](#).

DOWNLOAD FROM OTN

LICENSE

- [Oracle Master License Agreement](#)
- Free for evaluation and non-production use
- [Contact us](#) for commercial use and support options

BENEFITS

- Faster performance and smaller footprint
- Enhanced security features
- Managed capabilities for native code
- Premier 24x7x365 support via [MOS](#)

Available Distributions

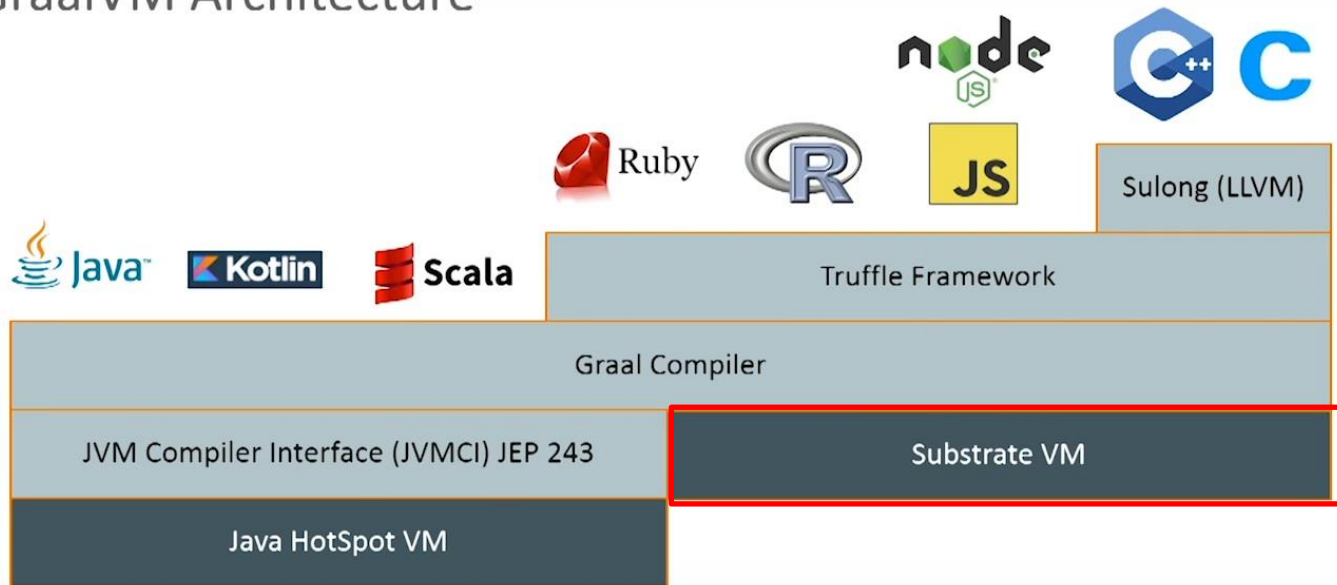
GraalVM is available as **GraalVM Enterprise** and **GraalVM Community** editions and includes support for [Java 11 and Java 17](#). GraalVM Enterprise is based on Oracle JDK while GraalVM Community is based on OpenJDK.

GraalVM is available for Linux and macOS on x86 64-bit and ARM 64-bit systems, and for Windows on x86 64-bit systems. Depending on the platform, the distributions are shipped as *.tar.gz* or *.zip* archives. See the [Getting Started guide](#) for installation instructions.



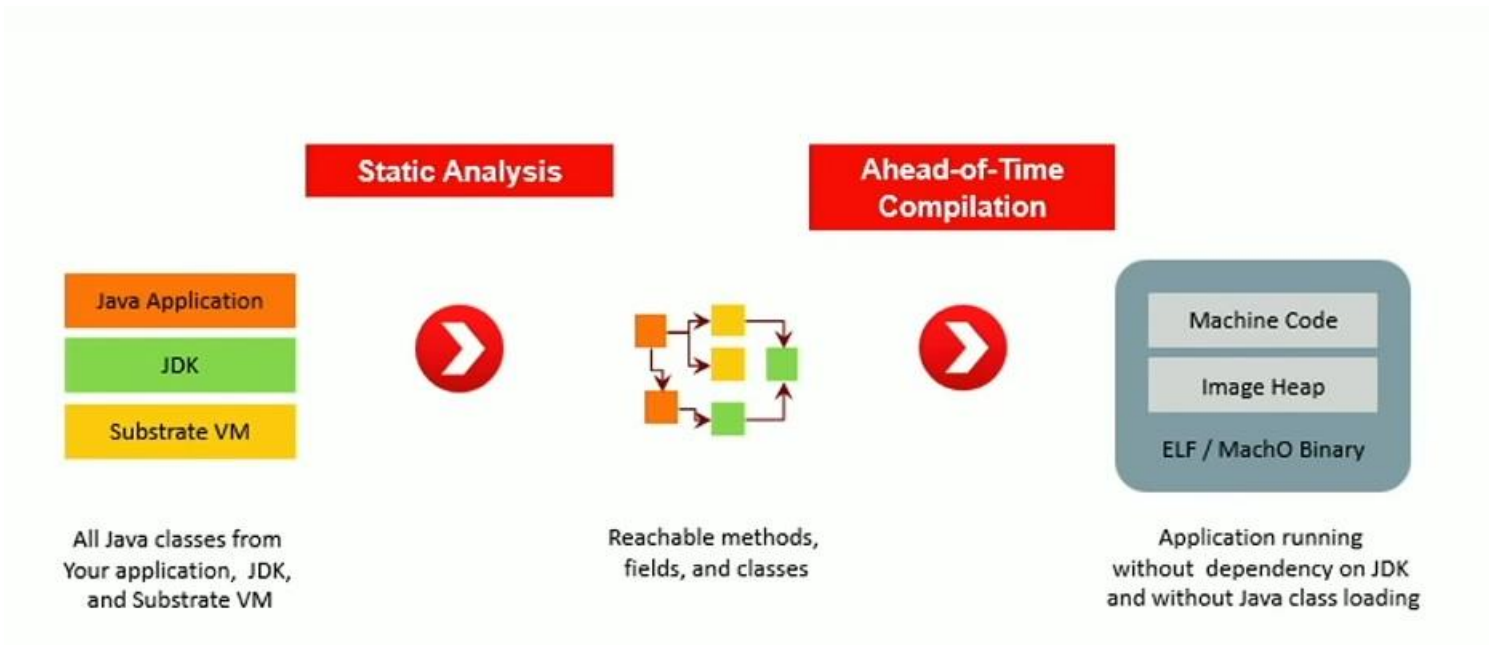
GraalVM Architecture

GraalVM Architecture





SubstrateVM





GraalVM on SubstrateVM

A game changer for Java & Serverless?

Java Function compiled into a **native executable** using **GraalVM on SubstrateVM** reduces

- “cold start” times
- memory footprint

by order of magnitude compared to running on JVM.



Current challenges with native executable using GraalVM

- AWS doesn't provide GraalVM (Native Image) as Java Runtime out of the box
- AWS provides Custom Runtime Option





Custom Lambda Runtimes

Custom AWS Lambda runtimes

You can implement an AWS Lambda runtime in any programming language. A runtime is a program that runs a Lambda function's handler method when the function is invoked. You can include a runtime in your function's deployment package in the form of an executable file named `bootstrap`.

A runtime is responsible for running the function's setup code, reading the handler name from an environment variable, and reading invocation events from the Lambda runtime API. The runtime passes the event data to the function handler, and posts the response from the handler back to Lambda.

Your custom runtime runs in the standard Lambda [execution environment](#). It can be a shell script, a script in a language that's included in Amazon Linux, or a binary executable file that's compiled in Amazon Linux.

To get started with custom runtimes, see [Tutorial – Publishing a custom runtime](#). You can also explore a custom runtime implemented in C++ at [awslabs/aws-lambda-cpp](#) on GitHub.

Topics

- [Using a custom runtime](#)
- [Building a custom runtime](#)

Using a custom runtime

To use a custom runtime, set your function's runtime to `provided`. The runtime can be included in your function's deployment package, or in a [layer](#).

Example function.zip

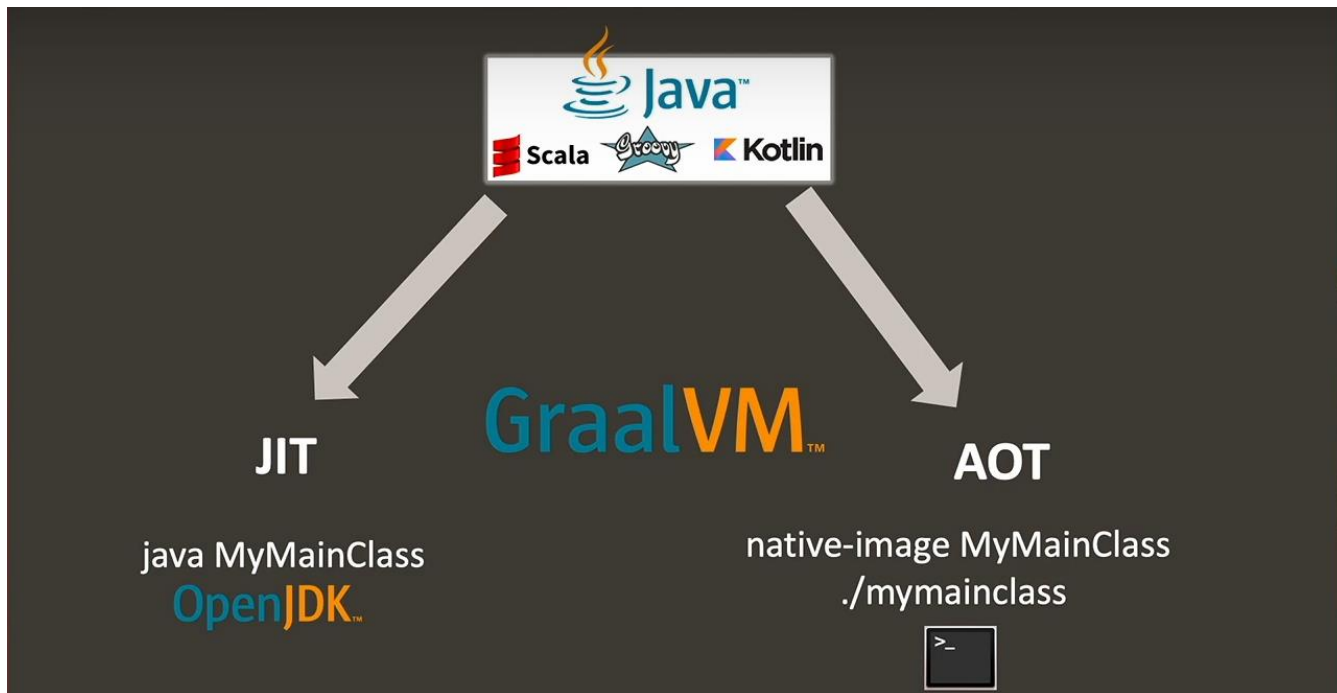
```
.
├─ bootstrap
└─ function.sh
```



If there's a file named `bootstrap` in your deployment package, Lambda executes that file. If not, Lambda looks for a runtime in the function's layers. If the `bootstrap` file isn't found or isn't executable, your function returns an error upon invocation.

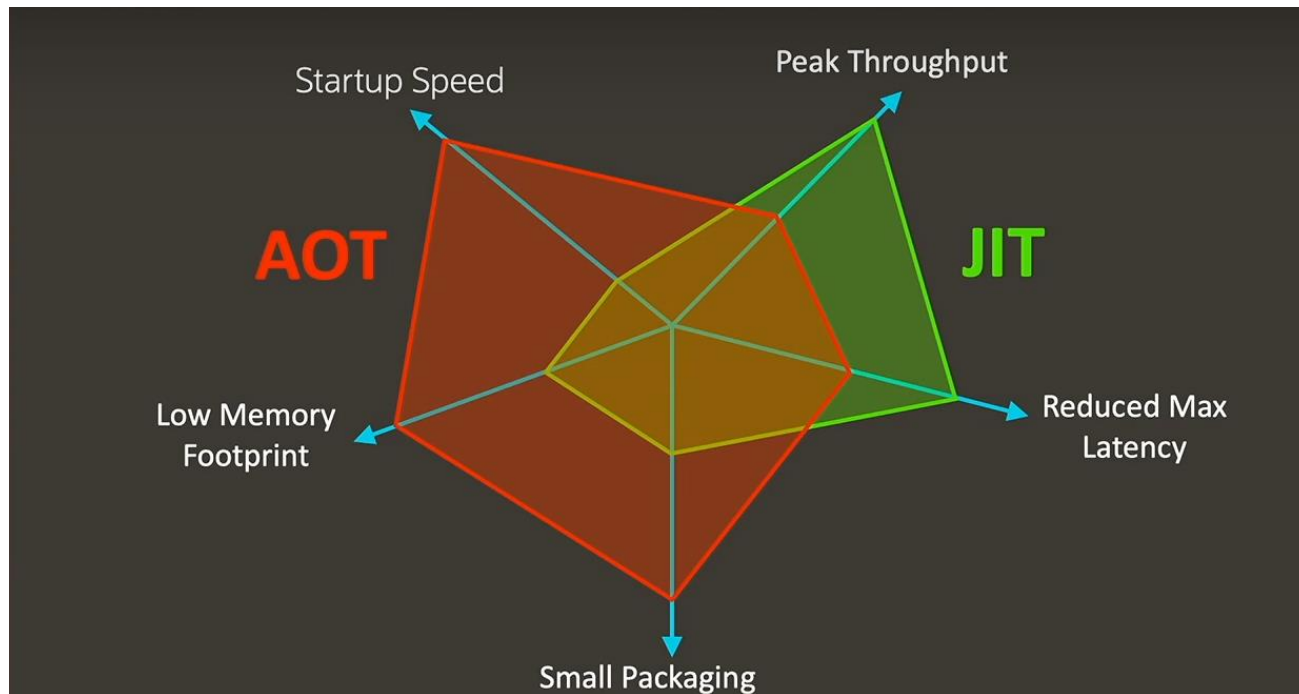


GraalVM Complilation Modes



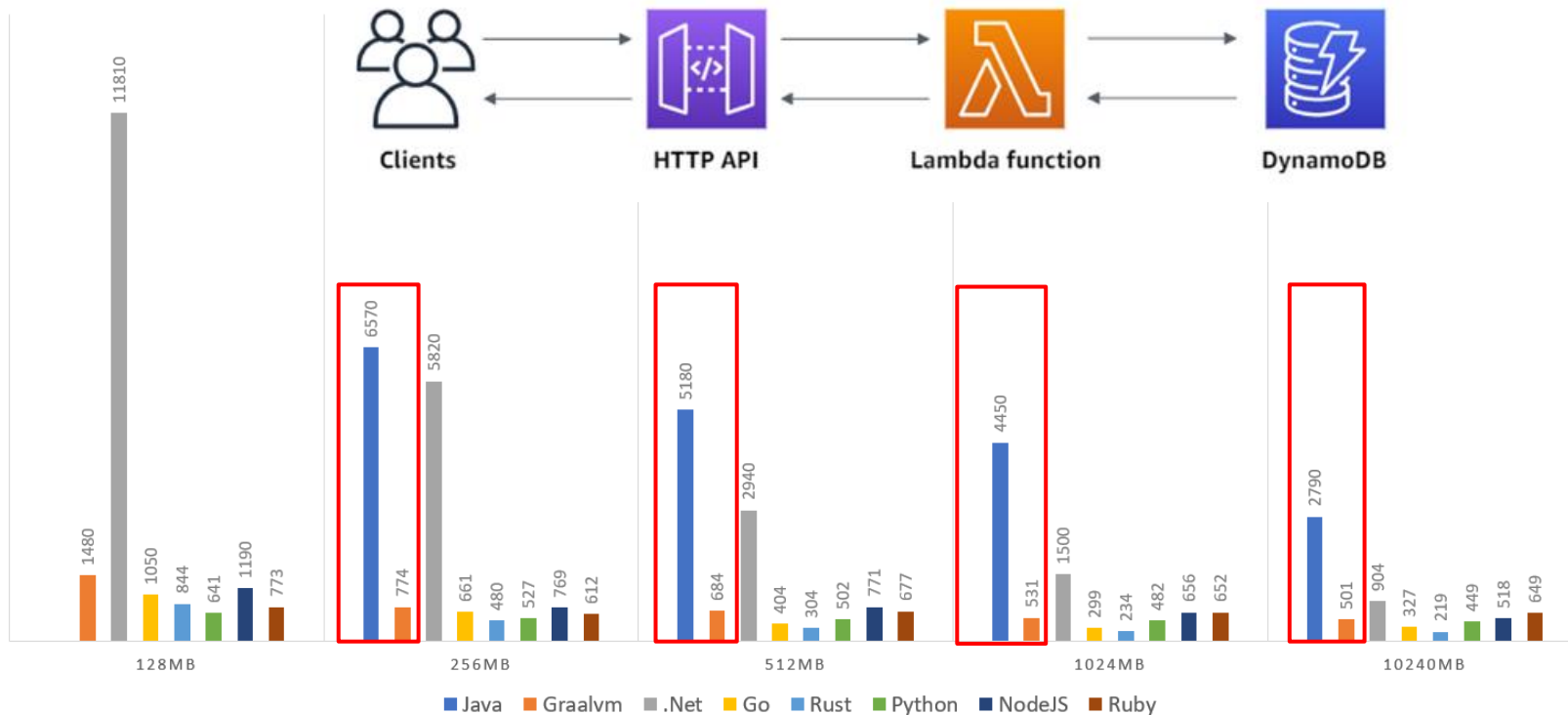


AOT vs JIT





GraalVM Native Cold Start 2021



Source: Aleksandr Filichkin: "AWS Lambda battle 2021: performance comparison for all languages (cold and warm start)"
<https://filia-aleks.medium.com/aws-lambda-battle-2021-performance-comparison-for-all-languages-c1b441005fd1>



Support of GraalVM native images in Frameworks

Spring Boot/ Spring Framework : Ongoing work on experimental **Spring Native** project.

Quarkus: a Kubernetes Native Java framework developed by Red Hat tailored for GraalVM and HotSpot, crafted from best-of-breed Java libraries and standards.

Micronaut: a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications.



Common principles for all frameworks

- Rely on as little reflection as possible
- Avoid runtime byte code generation, runtime generated proxies and dynamic class loading as much as possible
- Process annotations at compile time
- **The common goals:**
 - increase developer productivity
 - May decrease cold start times compared to plain Java solution (with and without the usage of GraalVM Native Image) using various compile-time optimization techniques
 - Currently only available for Micronaut



Steps to deploy to AWS

- Installation prerequisites
 - Framework of your choice (Micronaut, Quarkus, Spring Native)
 - GraalVM and Native Image
 - Apache Maven or Gradle
 - AWS CLI and AWS SAM CLI (or SAM local for local testing)
- Build Linux executable of your application with GraalVM native-image
 - Use Maven or Gradle plugin
- Deploy Linux executable as AWS Lambda Custom Runtime
 - Function.zip with bootstrap Linux executable

Example function.zip

```
├─ bootstrap
└─ function.sh
```



Quarkus





Quarkus Example with Spring Annotations

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.*;
```

```
@RestController  
public class PetsController {  
  
    private PetData petData;  
  
    @Autowired  
    public PetsController(PetData data) {  
        petData = data;  
    }  
  
    @RequestMapping(path = "/pets", method = RequestMethod.POST)  
    public Pet createPet(@RequestBody Pet newPet) {  
        if (newPet.getName() == null || newPet.getBreed() == null) {  
            return null;  
        }  
  
        Pet dbPet = newPet;  
        dbPet.setId(UUID.randomUUID().toString());  
        return dbPet;  
    }  
  
    @RequestMapping(path = "/pets/{petId}", method = RequestMethod.GET)  
    public Pet getPet(@RequestParam("petId") String petId) {  
        Pet newPet = new Pet();  
        newPet.setId(UUID.randomUUID().toString());  
        newPet.setBreed(petData.getRandomBreed());  
        newPet.setDateOfBirth(petData.getRandomDoB());  
        newPet.setName(petData.getRandomName());  
        return newPet;  
    }  
  
    @RequestMapping(path = "/pets", method = RequestMethod.GET)  
    public Pet[] listPets(@RequestParam("limit") Optional<Integer> limit) {  
        int queryLimit = 10;
```




Build GraalVM Native Image with Quarkus

```
<profile>
  <id>native</id>
  <activation>
    <property>
      <name>native</name>
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-maven-plugin</artifactId>
        <version>${quarkus.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>native-image</goal>
            </goals>
            <configuration>
              <enableHttpUrlHandler>true</enableHttpUrlHandler>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

mvn **-Pnative** package
and optionally
-Dquarkus.native.container-
build=true

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.1.0</version>
  <executions>
    <execution>
      <id>zip-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>function</finalName>
        <descriptors>
          <descriptor>src/assembly/zip.xml</descriptor>
        </descriptors>
        <attach>false</attach>
        <appendAssemblyId>>false</appendAssemblyId>
      </configuration>
    </execution>
  </executions>
</plugin>
```



Build GraalVM Native Image with Quarkus

<dependencies>

<dependency>

<groupId>io.quarkus</groupId>

<artifactId>quarkus-resteasy</artifactId>

</dependency>

<dependency>

<groupId>io.quarkus</groupId>

<artifactId>quarkus-amazon-lambda-http</artifactId>

</dependency>

<dependency>

<groupId>io.quarkus</groupId>

<artifactId>quarkus-spring-web</artifactId>

</dependency>

<dependency>

<groupId>io.quarkus</groupId>

<artifactId>quarkus-junit5</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>rest-assured</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

```
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Running Quarkus native-image plugin on GraalVM 22.1.0 Java 17 CE (Java Version 17.0.3+7-jvmci-22.1-b06)
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] /usr/lib/jvm/graalvm-ce-java17-22.1.0/bin/native-image -J-Dsun.nio.ch.maxDirectArraySize=100 -J-Djava.util.logging.manager=org.jboss.logmanager.log
Manager -J-Dio.netty.leakDetection.level=DISABLED -J-Dio.netty.allocator.maxOrders=3 -J-Dvertx.logger-delegate-factory-class-name=io.quarkus.vertx.core.runtime.VertxLogDelegateFactory -J-Dvertx.disableS
Resolver -J-Duser.language=en -J-Dfile.encoding=UTF-8 -H:-ParseOnce -J--add-exports=java.security.jgss/sun.security.krb5=ALL-UNNAMED -J--add-opens=java.base/java.text=ALL-UNNAMED -H:InitialCollectionPolicy=com.orac
le.svm.core.gencavenge.collectionPolicy$B$SpaceAndTime -H:+JNI -H:+AllowFoldMethods -J-Djava.awt.headless=true -H:FallbackThreshold=0 --link-at-build-time -H:+ReportExceptionStackTraces -H:-AddAllCharsets -H:En
ableURLProtocols=http -H:NativeLinkerOptions=no-pie -H:-UseServiceLoaderFeature -H:+StackTrace quarkus-lambda-1.0.0-SNAPSHOT-runner -jar quarkus-lambda-1.0.0-SNAPSHOT-runner.jar
.....[GraalVM Native Image: Generating 'quarkus-lambda-1.0.0-SNAPSHOT-runner' (executable)...
.....[1/7] Initializing...
(18.8s @ 0.25GB) Version info: 'GraalVM 22.1.0 Java 17 CE'
C compiler: gcc (linux, x86_64, 9.4.0)
Garbage collector: Serial GC
3 user-provided feature(s)
- io.quarkus.runner.AutoFeature
- io.quarkus.runtime.graal.DisableLoggingAutoFeature
- io.quarkus.runtime.graal.ResourcesFeature
[2/7] Performing analysis... [*****] (95.3s @ 2.33GB) 11,273 (90.14%) of 12,506 classes reachable
16,340 (59.28%) of 27,564 fields reachable
58,839 (56.99%) of 103,241 methods reachable
467 classes, 160 fields, and 1,821 methods registered for reflection
64 classes, 75 fields, and 55 methods registered for JNI access
[3/7] Building universe... (9.0s @ 2.83GB)[4/7] Parsing methods... [*****] (21.3s @ 1.73GB)[6/7] Compiling methods... [*****] (18.1s @ 2.19GB) 22.97MB (44.69%) for code area
(66.7s @ 2.15GB)[5/7] Inlining methods... [*****] (65.1s @ 3.41GB)[7/7] Creating image...
39,122 compilation units
23.48MB (45.97%) for image heap: 7,758 classes and 290,542 objects
5.65MB (10.85%) for other data
52.89MB in total
```



AWS Lambda Deployment of Custom Runtime with SAM

Resources:

PetStoreNativeFunction:

Type: AWS::Serverless::Function

Properties:

Handler: not.used.in.provided.runtime

Runtime: provided

CodeUri: target/function.zip

MemorySize: 128

Policies: AWSLambdaBasicExecutionRole

Tracing: Active

Timeout: 15

Environment:

Variables:

DISABLE_SIGNAL_HANDLERS: true

Events:

GetResource:

Type: Api

Properties:

Path: /{proxy+}

Method: any

Outputs:

PetStoreNativeApi:

Description: URL for application

Value: !Sub 'https://\${ServerlessRestApi}.execute-api
.\${AWS::Region}.amazonaws.com/Prod/'

Export:

Name: PetStoreNativeApi

Local testing:

sam local start-api -t sam.native.yaml

curl localhost:3000/{yourURI}

Cloud deployment:

sam deploy -g -t sam.native.yaml

curl https://xxxxxxxxx.execute-api.xx-xxxx-
1.amazonaws.com/Prod/pets/5



Quarkus Additional Features

- AWS Lambda currently works by implementing `com.amazonaws.services.lambda.runtime.RequestHandler` interface or by using Spring Web annotations model like `@RestController`, `@RequestMapping`

- Doesn't support Lambda function implementing `Java 8 Function` Interface

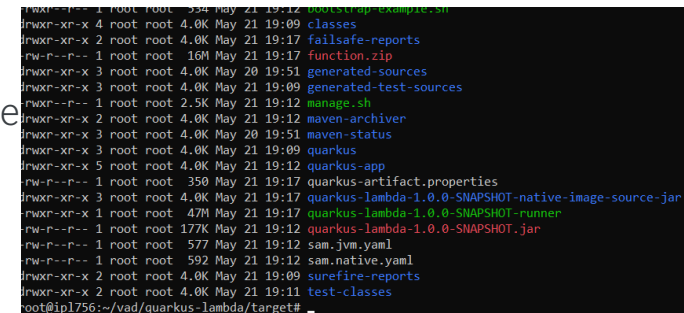
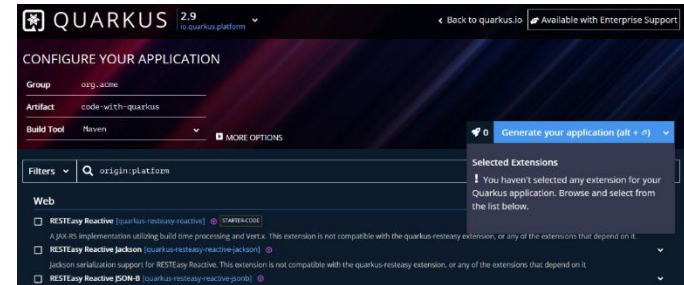
- Website (<https://code.quarkus.io/>)

- CLI for creating the App

- `quarkus create app`
- use `quarkus-amazon-lambda` extension in `pom.xml`
- `quarkus build --native -Dquarkus.native.container-build=true`

- Eclipse MicroProfile compatible

- Funqy for multi cloud solutions





Micronaut Framework





Micronaut Example

```
package example.micronaut;

import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;

@Controller("/book")
public class BookController {

    @Post("/saveBook/{name}/{isbn}")
    public Book save(String name, int isbn) {
        Book book = new Book();
        book.setName(name);
        book.setIsbn(isbn);
        return book;
    }

    @Get ("/getBook/{isbn}")
    public Book get(int isbn) {
        Book book = new Book ();
        book.setName("New Vadym's book");
        book.setIsbn(isbn);
        return book;
    }
}
```



Build GraalVM Native Image with Quarkus

```
<dependencies>
  <dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-inject</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>io.micronaut</groupId>
    <artifactId>micronaut-validation</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>io.micronaut.aws</groupId>
    <artifactId>micronaut-function-aws-api-proxy</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>io.micronaut.aws</groupId>
    <artifactId>micronaut-function-aws-custom-runtime</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>io.micronaut.aws</groupId>
    <artifactId>micronaut-function-aws-api-proxy-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

`./mvnw package -Dpackaging=native-image`
`-Dmicronaut.runtime=lambda`

Packaging can also have `docker` or
`docker-native` value



Micronaut Additional Features

- AWS Lambda currently works by implementing its own annotations (very similar to Spring Web) and should potentially work with Spring Web annotations model like **@RestController**, **@RequestMapping**
- Website (<https://micronaut.io/launch>) or CLI for creating the App
- Custom Validators
- No support for MicroProfile
- Micronaut AOT

The screenshot shows the Micronaut Launch web interface. At the top, there is a logo for 'MICRONAUT LAUNCH' and a row of social media icons. Below the logo, there are four columns of configuration options:

- Application Type:** Micronaut Application
- Java Version:** 17
- Name:** demo
- Base Package:** com.example

Below these are four rows of radio button options:

- Micronaut Version:** 3.4.3, 3.4.4-SNAPSHOT, 2.5.13
- Language:** Java, Groovy, Kotlin
- Build Tool:** Gradle, Gradle Kotlin, Maven
- Test Framework:** JUnit, Spock, Kotest

At the bottom, there are four buttons: '+ FEATURES', '- DIFF', 'Q PREVIEW', and 'GENERATE PROJECT'. Below the buttons, it says 'Included Features (4)' and lists: aws-lambda, aws-lambda-custom-runtime, spring, and spring-boot, each with a close button.



Micronaut® AOT: build-time optimizations for Micronaut applications

Micronaut AOT is an extension to the Micronaut Framework which is the foundation to many optimizations that can be implemented at build time but weren't possible solely with annotation processing.

By effectively analyzing the deployment environment, AOT is capable of reducing startup times or distribution size for both native and JVM deliverables.

```
./mvnw package -Dpackaging=native-image  
-Dmicronaut.runtime=lambda -Dmicronaut.aot.enabled=true
```

Packaging can also have `docker` or `docker-native` value

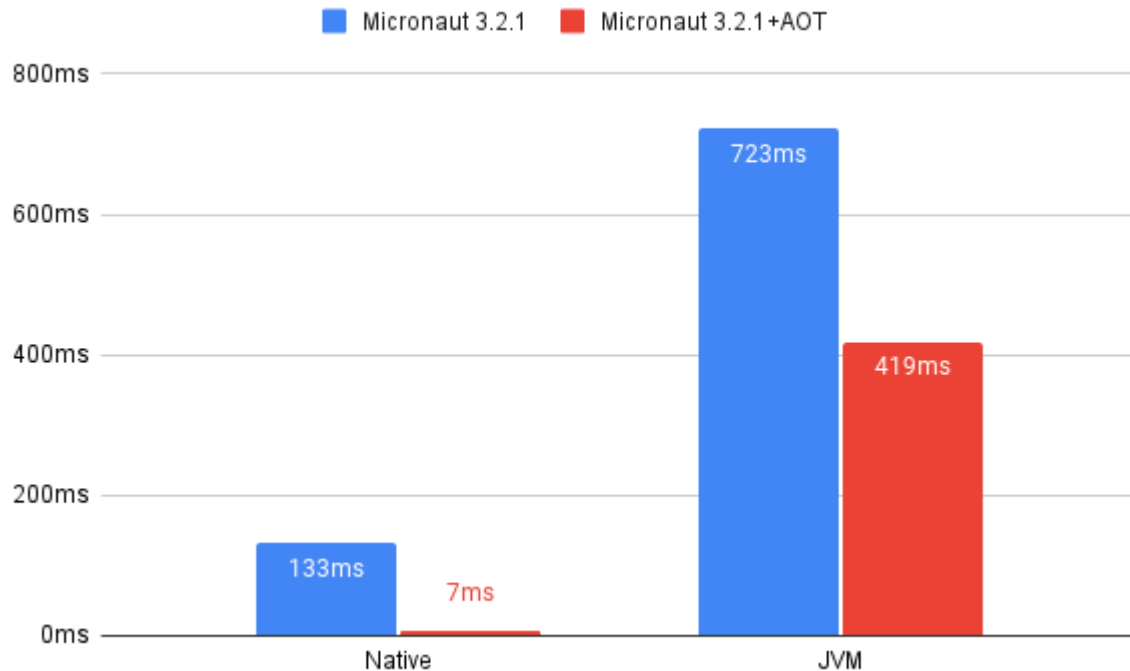


Micronaut® AOT: build-time optimizations for Micronaut applications

- **optimize service loading** by pre-scanning the list of available services and implementing a loading strategy that's fully parallel in the JVM and serial in GraalVM native executables (because classloading is effectively free in native executables)
- **convert YAML configuration to Java configuration** to make apps startup faster, while reducing the final binary size because YAML parsing is no longer necessary
- **cache the environment** so that once the application is started, the framework assumes that system properties and environment variables won't change, saving time when performing lookups
- **precompute bean requirements** to eliminate beans whose requirements won't be met at runtime (this can happen if you have transitive dependencies bringing beans that you don't use, for example)
- **deduce the environment at build time**, which is the major optimization we used in the example above
- **precompute some expensive operations**, like converting environment variable names to Micronaut configuration properties
- **optimize classloading** by avoiding lookup for classes that we know are not on classpath



Micronaut® AOT: build-time optimizations for Micronaut applications





Spring (Boot) Framework





Spring GraalVM Native Project

build failing documentation

Spring Native provides beta support for compiling Spring applications to native executables using [GraalVM native-image](#) compiler, in order to provide a native deployment option typically designed to be packaged in lightweight containers. In practice, the target is to support your Spring Boot application, almost unmodified, on this new platform.

Watch the [video](#) and read the [blog post](#) of Spring Native Beta announcement to learn more.

Announcing
Spring Native Beta!



Spring Boot 3 and Spring Framework 6, due in late 2022, will have built-in support for native Java.



Quick start



Spring Native Example

`@SpringBootApplication`

```
public class SpringBootNativeApp {  
  
    public static void main(String[] args) {  
        SpringApplication.run(SpringBootNativeApp.class, args);  
    }  
}
```

`@Bean`

```
public GetBookByIdFunction getBookById() {  
    return new GetBookByIdFunction();  
}
```

```
GetBookByIdFunction:  
Type: AWS::Serverless::Function  
Properties:  
Environment:  
Variables:  
    DEFAULT_HANDLER: getBookById  
Events:  
    GetRequestById:  
        Type: Api  
        Properties:  
            RestApiId: !Ref MyApi  
            Path: /book/{id}  
            Method: get
```

```
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;  
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;  
  
import java.util.function.Function;
```

```
@Component  
public class GetBookByIdFunction implements Function<APIGatewayProxyRequestEvent,  
    APIGatewayProxyResponseEvent> {
```

```
@Override  
public APIGatewayProxyResponseEvent apply(APIGatewayProxyRequestEvent requestEvent) {  
    String id = requestEvent.getPathParameters().get("id");  
  
    return new APIGatewayProxyResponseEvent()  
        .withStatusCode(HttpStatusCode.OK)  
        .withBody(objectMapper.writeValueAsString("book with id "+id+  
            "found and has title "+" Vadym"));  
} catch (Exception je) {  
    return new APIGatewayProxyResponseEvent()  
        .withStatusCode(HttpStatusCode.INTERNAL_SERVER_ERROR)  
        .withBody("Internal Server Error :: " + je.getMessage());  
}
```

`curl https://xxxxxxxxxx.execute-api.
xx-xxxx-1.amazonaws.com/prod/book/5`



Build GraalVM Native Image with Spring

mvn -Pnative package

```
<profiles>
  <profile>
    <id>native</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <executions>
            <execution>
              <id>native-zip</id>
              <phase>package</phase>
              <goals>
                <goal>single</goal>
              </goals>
              <inherited>>false</inherited>
            </execution>
          </executions>
          <configuration>
            <descriptors>
              <descriptor>src/assembly/native.xml</descriptor>
            </descriptors>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-native</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-aws</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

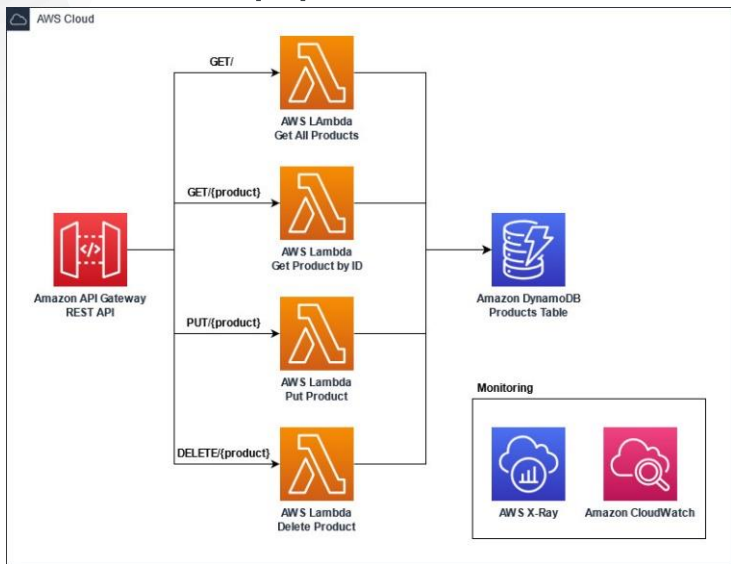


Spring Native

- AWS Lambda currently only works by implementing **Java 8 Functional Interface**
 - Doesn't support Lambda function implementing `com.amazonaws.services.lambda.runtime.RequestHandler` interface
 - Doesn't support Spring Web Annotations model like **@RestController**, **@RequestMapping**, which Quarkus and Micronaut do



Lambda demo with common Java application frameworks



Artillery is used to make 100 requests / second for 10 minutes to our API endpoints.

Results from Managed Java Runtime

| | Cold Start (ms) | | | | Warm Start (ms) | | | |
|-------------|-----------------|----------|----------|----------|-----------------|-------|-------|--------|
| | p50 | p90 | p99 | max | p50 | p90 | p99 | max |
| Micronaut | 8505.57 | 8977.26 | 9685.76 | 10512.48 | 9.38 | 14.86 | 40.39 | 553.75 |
| Quarkus | 6384.45 | 6671.49 | 7055.55 | 8303.17 | 10.41 | 19.07 | 48.45 | 317.69 |
| Spring Boot | 12673.61 | 13098.60 | 13497.31 | 14118.06 | 10.99 | 21.75 | 75.00 | 419.90 |

Results from GraalVM Native images running in custom runtime

| | Cold Start (ms) | | | | Warm Start (ms) | | | |
|-------------|-----------------|--------|--------|--------|-----------------|-------|-------|--------|
| | p50 | p90 | p99 | max | p50 | p90 | p99 | max |
| Micronaut | 604.16 | 659.02 | 700.45 | 893.70 | 6.30 | 8.00 | 15.88 | 69.9 |
| Quarkus | 437.45 | 475.76 | 519.50 | 528.03 | 7.45 | 12.60 | 21.32 | 93.45 |
| Spring Boot | 620.66 | 684.53 | 721.77 | 751.98 | 9.10 | 14.22 | 23.61 | 259.16 |



Lambda Container Image Support

AWS News Blog

New for AWS Lambda – Container Image Support

by Danilo Poccia | on 01 DEC 2020 | in [Announcements](#), [AWS Lambda](#), [AWS ReInvent](#), [Compute](#), [Containers](#), [Serverless](#) | [Permalink](#) | [Share](#)



Voiced by Amazon Polly

With [AWS Lambda](#), you upload your code and run it without thinking about servers. [Many customers enjoy the way this works](#), but if you've invested in container tooling for your development workflows, it's not easy to use the same approach to build applications using Lambda.

To help you with that, you can now package and deploy Lambda functions as **container images** of up to **10 GB** in size. In this way, you can also easily build and deploy larger workloads that rely on sizable dependencies, such as machine learning or data intensive workloads. Just like functions packaged as ZIP archives, functions deployed as container images benefit from the same operational simplicity, automatic scaling, high availability, and native integrations with many services.

We are providing **base images** for all the supported **Lambda runtimes (Python, Node.js, Java, .NET, Go, Ruby)** so that you can easily add your code and dependencies. We also have base images for custom runtimes based on [Amazon Linux](#) that you can extend to include your own runtime implementing the [Lambda Runtime API](#).

Create function Info

Choose one of the following options to create your function.

Author from scratch

Start with a simple Hello World example.

Use a blueprint

Build a Lambda application from sample code and configuration presets for common use cases.

Container image

Select a container image to deploy for your function.

Browse serverless app repository

Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name

Enter a name that describes the purpose of your function.

Use only letters, numbers, hyphens, or underscores with no spaces.

Container image URI Info

The location of the container image to use for your function.

Requires a valid Amazon ECR image URI

Source: „<https://aws.amazon.com/de/blogs/aws/new-for-aws-lambda-container-image-support/>“



Lambda Container Image Support

- What about the support of the current Java version for Lambda?
 - Amazon Corretto provides Long Term Support (LTS)
 - Currently only Java 8, Java 11
 - Use Container (Docker) Image with i.e. Java 19

Amazon Corretto

No-cost, multiplatform, production-ready distribution of OpenJDK

Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). Corretto comes with long-term support that will include performance enhancements and security fixes. Amazon runs Corretto internally on thousands of production services and Corretto is certified as compatible with the Java SE standard. With Corretto, you can develop and run Java applications on popular operating systems, including Linux, Windows, and macOS.



Lambda Container Image Support with Java 19

1. Download the desired Java version and copy the local application code to the Docker environment and build it with Maven:

```
FROM amazonlinux:2
...
# Update packages and install Amazon Corretto 18, Maven and Zip
RUN yum -y update
RUN yum install -y java-18-amazon-corretto-devel maven zip
...
# Copy the software folder to the image and build the function
COPY software software
WORKDIR /software/example-function
RUN mvn clean package
```

2. This step results in an uber-jar (function.jar) that you can use as an input argument for `jdeps`. The output is a file containing all the Java modules that the function depends on:

```
RUN jdeps -q \
  --ignore-missing-deps \
  --multi-release 18 \
  --print-module-deps \
  target/function.jar > jre-deps.info
```

3. Create an optimized Java runtime based on those application modules with `jlink`. Remove unnecessary information from the runtime, for example header files or man-pages:

```
RUN jlink --verbose \
  --compress 2 \
  --strip-java-debug-attributes \
  --no-header-files \
  --no-man-pages \
  --output /jre18-slim \
  --add-modules $(cat jre-deps.info)
```

4. This creates your own custom Java 18 runtime in the `/jre18-slim` folder. You can apply additional optimization techniques such as [Class-Data-Sharing \(CDS\)](#) to generate a `classes.jsa` file to accelerate the class loading time of the JVM.

```
RUN /jre18-slim/bin/java -Xshare:dump
```

<https://aws.amazon.com/de/blogs/compute/build-a-custom-java-runtime-for-aws-lambda/>



Conclusion

- GraalVM and Frameworks are really powerful with a lot of potential
- GraalVM Native Image improves cold starts and memory footprint significantly
- GraalVM Native Image is currently not without challenges
 - AWS Lambda Custom Runtime requires Linux executable only
 - Building Custom Runtime requires some additional effort
 - e.g. you need to scale CI pipeline to build memory-intensive native image yourself
 - Build time is a factor
 - You pay for the init-phase of the function packaged as AWS Lambda Custom and Docker Runtime
 - Init-phase is free for the managed runtimes like Java 8 and Java 11 (Corretto)



Personal Recommendations (highly opinionated)

- By default start with **plain managed** Java Long Term Support Version with Amazon Corretto 11 + optionally your favorite framework (Micronaut, Quarkus)
- If you don't want to miss years of innovation and use the newest Java Version?
 - Use Lambda Docker (Container) Image Support
- If your function needs constantly low response times for the known period of time ?
 - Use Provisioned Concurrency additionally
- If your function needs constantly low response time and low cost is a requirement?
 - Use GraalVM Native Image + optionally your favorite framework (Micronaut, Quarkus, Spring Boot Native) and AWS Lambda Custom Runtime
- The usage of the frameworks (Micronaut, Quarkus, Spring Boot GraalVM Native) may improve your productivity but may add up to longer build time)



Try it yourselves

- Quarkus
 - <https://github.com/aws-samples/aws-quarkus-demo/tree/main/lambda>
 - <https://quarkus.io/guides/amazon-lambda>
- Micronaut
 - <https://github.com/micronaut-guides/micronaut-function-aws-lambda>
- Spring Native
 - <https://github.com/spring-projects-experimental/spring-native/tree/main/samples/cloud-function-aws>
- Misc examples with all frameworks
 - <https://github.com/aws-labs/aws-serverless-java-container/tree/master/samples>



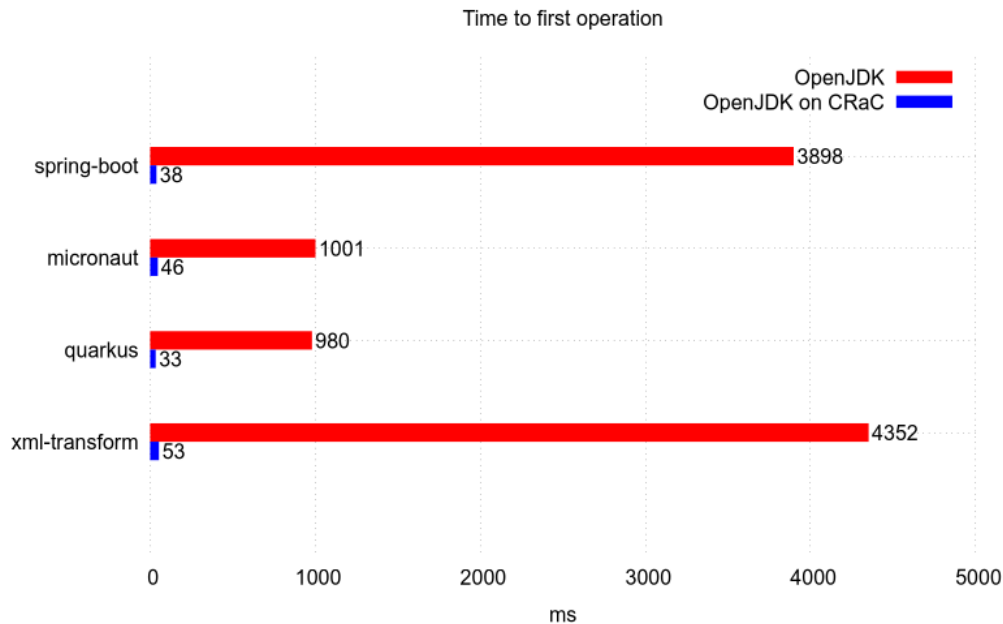
What's may come next

- Speed up warmup time of the Java applications
 - The C2 compiler is used for very hot methods, which uses profiling data collected from the running application to optimize as much as possible.
 - Techniques like aggressive method inlining and speculative optimizations can easily lead to better performing code than generated ahead of time (AOT) using a static compiler.
 - JVM needs both time and compute resources to determine which methods to compile and compiling them. This same work has to happen every time we run an application
 - Ideally, we would like to run the application and then store all the state about the compiled methods, even the compiled code and state associated with the application and then we'd like to be able to restore it



What's may come next

- CRaC (Coordinated Restore at Checkpoint) Project
 - Based on Linux Checkpoint/Restore in Userspace (CRIU)
 - Simple API: `beforeCheckpoint()` and `afterRestore()` methods







Accelerate Your Photo Business

[Get in Touch](#)

www.iplabs.de