How to reduce cold starts for Java Serverless applications in AWS

GraalVM, AWS SnapStart and Co

Vadym Kazulkin, ip.labs, FrOSCon , 6 August 2023

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

https://dev.to/vkazulkin

https://github.com/Vadym79/

ip.labs

https://www.iplabs.de/

Top Programming Languages: Rankings In Comparison

Index	1st	2nd	3rd	4th	5th
IEEE Spectrum	Python	Java	C	C++	JavaScript
GitHut 2.0 (Pull Requests)	JavaScript	Python	Java	Go	Ruby
GitHut 2.0 (Pushes)	JavaScript	Python	Java	C++	PHP
RedMonk	JavaScript	Python	Java	PHP	C++/C#
PYPL	Python	Java	Javascript	C#	C/C++
TIOBE	C	Python	Java	C++	C#

https://distantjob.com/blog/programming-languages-rank/

Life of the Java (Serverless) developer on AWS

AWS Java Versions Support

- Corretto Java 8
  - With extended long-term support until 2026
- Coretto Java 11 (since 2019)
- Coretto Java 17 (April 2023)
- Only Long Term Support (LTS) by AWS

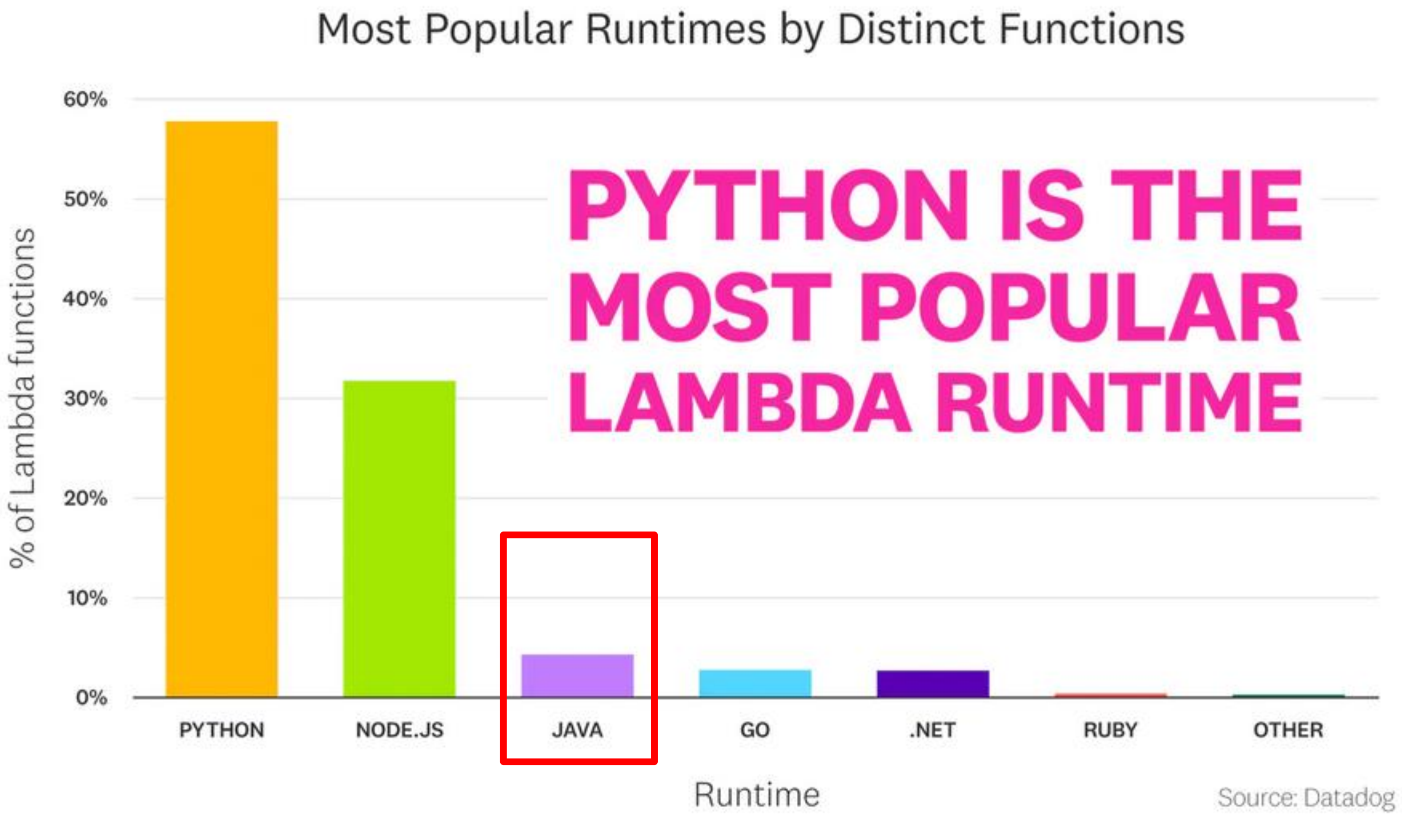
Amazon Corretto

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.

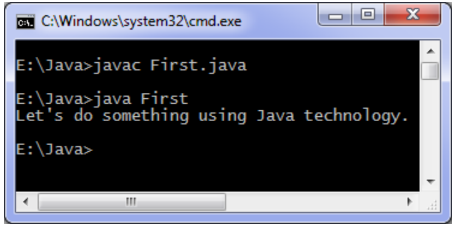
Java ist very fast and mature programming language…

… but Serverless adoption of Java looks like this



The State of Serverless 2021
https://www.datadoghq.com/state-of-serverless-2021

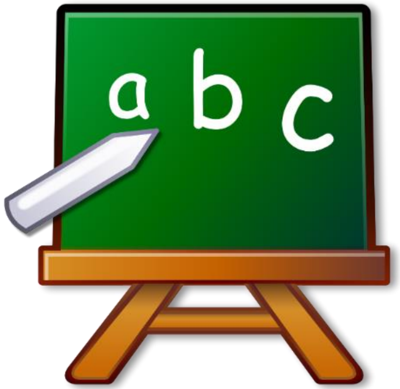
Developers love Java and will be happy to use it for Serverless



But what are the challenges ?

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.

MyTestJavaFunction

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 17

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.

- Java 17
- Node.js 18.x
- Python 3.10
- Ruby 2.7

Other supported

- Java 11 (Corretto)
- Java 8 on Amazon Linux 1
- Java 8 on Amazon Linux 2

Java 17

Basic settings

Description

Memory (MB) Info
Your function is allocated CPU proportional to the memory configured.

128 MB

Timeout Info
0 min 3 sec

Full CPU access only approx. at 1.8 GB memory allocated

Source https://blog.runscope.com/posts/how-to-write-your-first-aws-lambda-function

Creating AWS Lambda with Java 2/3

```java
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);
    }

}
```

```java
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

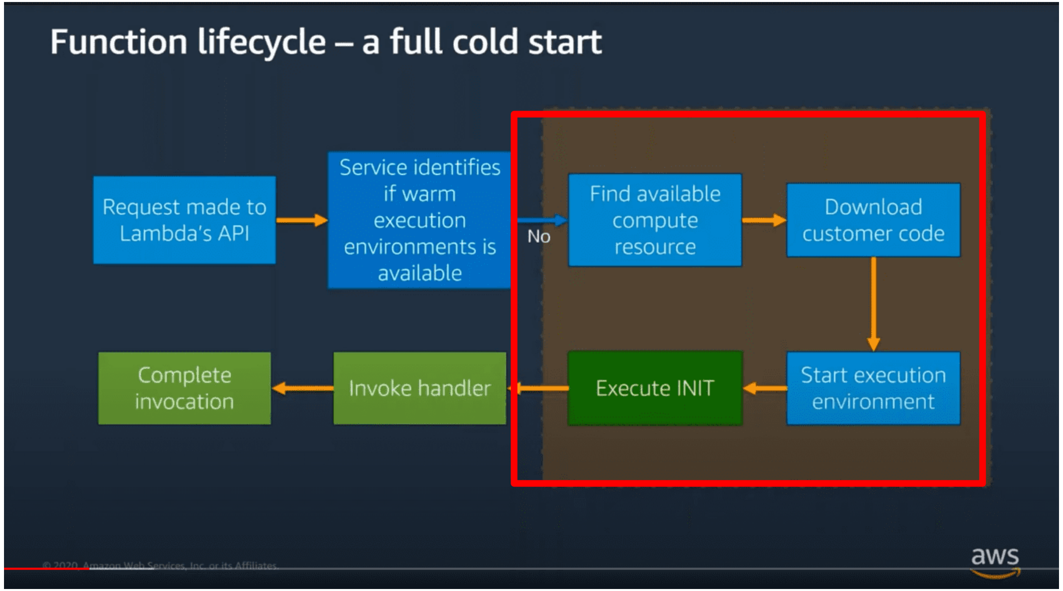
When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

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.

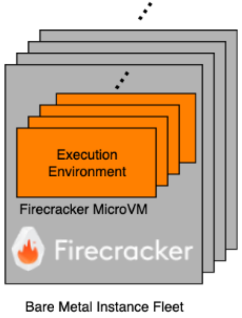
Challenge Number 1 with Java is a big cold-start

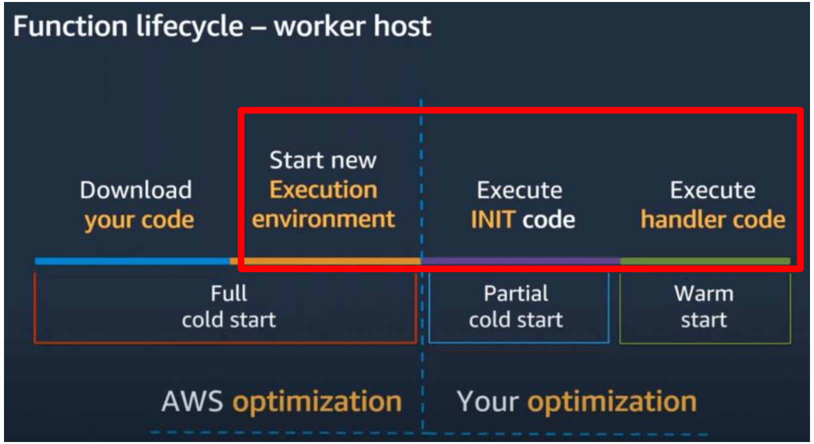
Function lifecycle- a full cold start

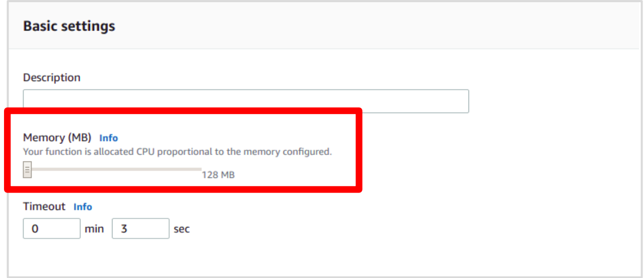


Sources: Ajay Nair „Become a Serverless Black Belt" https://www.youtube.com/watch?v=oQFORsso2go
Tomasz Łakomy "Notes from Optimizing Lambda Performance for Your Serverless Applications" https://tlakomy.com/optimizing-lambda-performance-for-serverless-applications

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

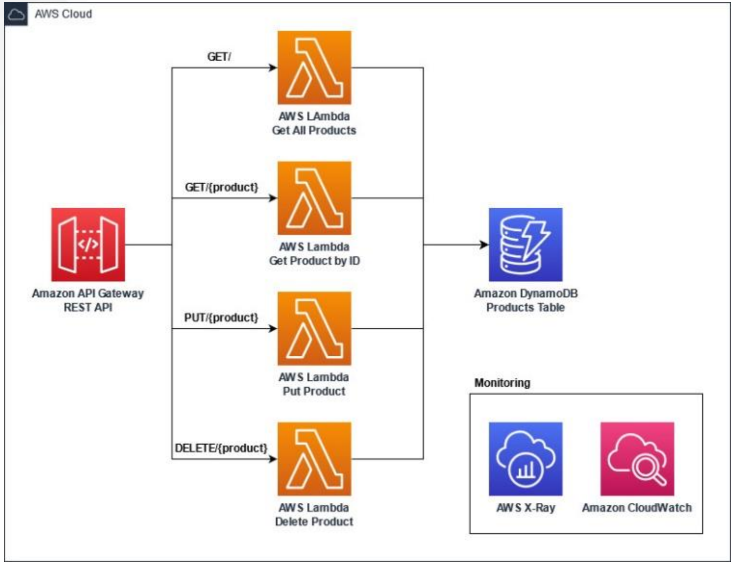






Sources: Ajay Nair „Become a Serverless Black Belt" https://www.youtube.com/watch?v=oQFORsso2go
Tomasz Łakomy "Notes from Optimizing Lambda Performance for Your Serverless Applications" https://tlakomy.com/optimizing-lambda-performance-for-serverless-applications
Michael Hart: „Shave 99.93% off your Lambda bill with this one weird trick" https://hichaelmart.medium.com/shave-99-93-off-your-lambda-bill-with-this-one-weird-trick-33c0acebb2ea

Lambda demo with common Java application frameworks



Cold start with Corretto Java 11

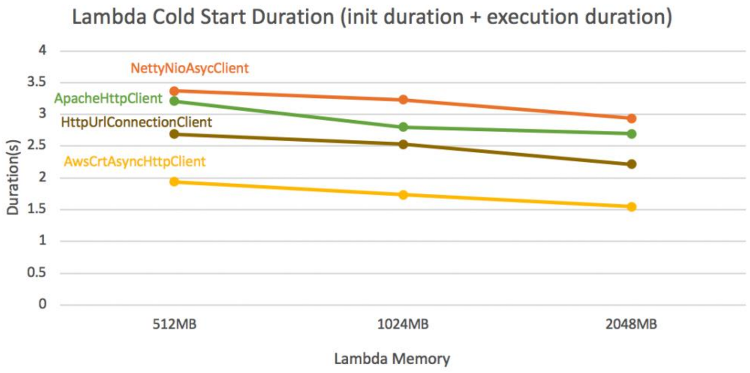
Framework	p50	p90	p99
Pure Java	3000-4500	3500-5000	3800-6000
Micronaut	3500-4800	3800-5400	4200-6500
Quarkus	3300-4600	3650-5200	4000-6300
Spring Boot	8000-10000	9200-12000	10000-13000

These are best case values after applying optimization techniques and best practices

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();
```



Source: https://aws.amazon.com/de/blogs/developer/announcing-availability-of-the-aws-crt-http-client-in-the-aws-sdk-for-java-2-x/
Vadym Kazulkin: https://dev.to/aws-builders/aws-sdk-for-java-2x-asynchronous-http-clients-and-their-impact-on-cold-start-times-and-memory-consumption-of-aws-lambda-366p

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>
```

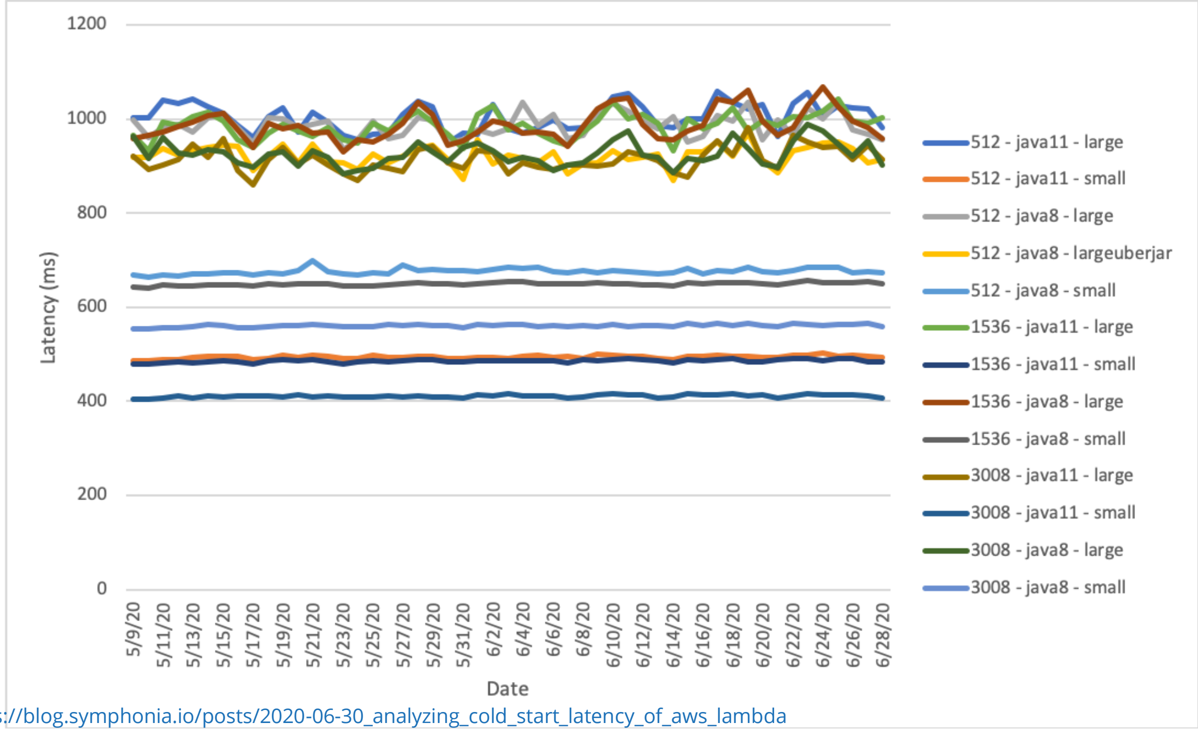
```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>
```

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8
Sean O'Toole „AWS Lambda Java Tutorial: Best Practices to Lower Cold Starts" https://www.capitalone.com/tech/cloud/aws-lambda-java-tutorial-reduce-cold-starts/
https://github.com/awsdocs/aws-doc-sdk-examples/tree/master/javav2

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

Artifact Size:

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



Source: Mike Roberts "Analyzing Cold Start latency of AWS Lambda" https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda

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();
```

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8
Sean O'Toole „AWS Lambda Java Tutorial: Best Practices to Lower Cold Starts" https://www.capitalone.com/tech/cloud/aws-lambda-java-tutorial-reduce-cold-starts/

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);
```

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8
Sean O'Toole „AWS Lambda Java Tutorial: Best Practices to Lower Cold Starts" https://www.capitalone.com/tech/cloud/aws-lambda-java-tutorial-reduce-cold-starts/

Best Practices and Recommendations

- Prime dependencies during initialization phase (when it worth doing)
  - „Fake“ the calls to pre-initalize „some other expensive stuff“
  - In case of DynamoDB client put the following code outside of the handler method to pre-initialize the Jackson Marshaller:

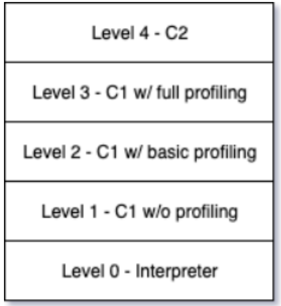
```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()...build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable(„mytable");

Item item = table.getItem("Id", 210);
```

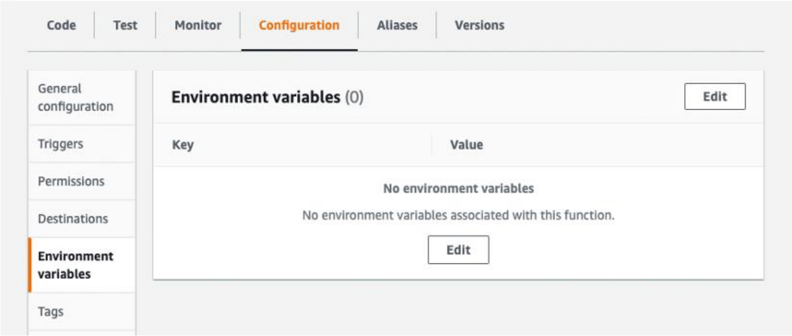
getItem() call forces Jackson Marshallers to initialize

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8
Sean O'Toole „AWS Lambda Java Tutorial: Best Practices to Lower Cold Starts" https://www.capitalone.com/tech/cloud/aws-lambda-java-tutorial-reduce-cold-starts/

Best Practices and Recommendations Using Tiered Compilation

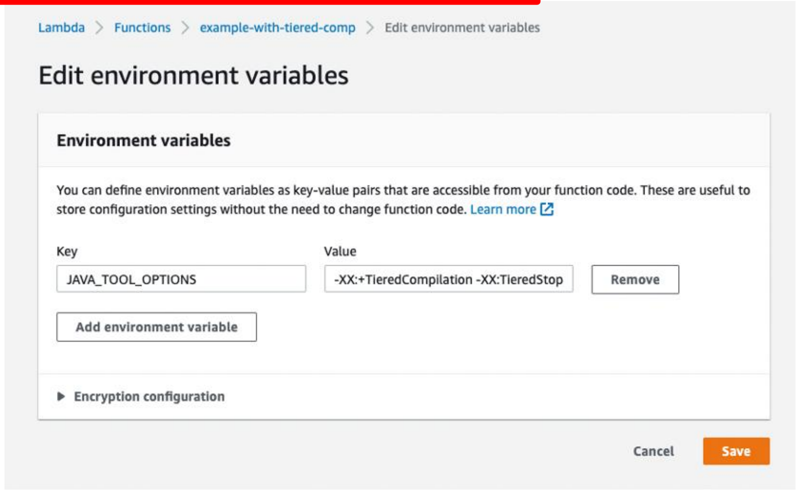
Achieve **up to 60%** faster startup times can use **level 1** compilation with little risk of reducing warm start performance





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

```bash
– Key: JAVA_TOOL_OPTIONS
– Value: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
```

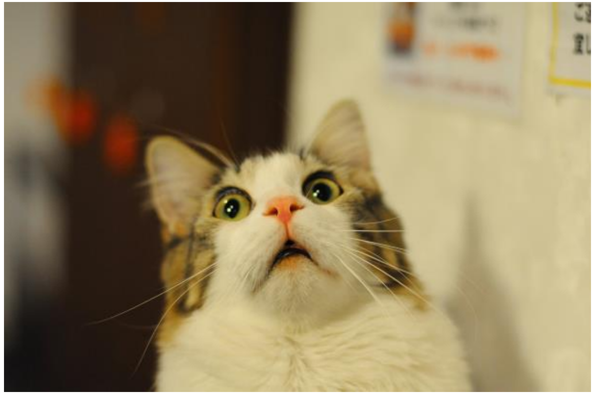


Mark Sailes: "Optimizing AWS Lambda function performance for Java"
https://aws.amazon.com/de/blogs/compute/optimizing-aws-lambda-function-performance-for-java/

Best Practices and Recommendations

Avoid:

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



spring Guice

Use DI Frameworks which aren‘t reflection-based

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8
Sean O'Toole „AWS Lambda Java Tutorial: Best Practices to Lower Cold Starts" https://www.capitalone.com/tech/cloud/aws-lambda-java-tutorial-reduce-cold-starts/

GraalVM enters the scene

GraalVM™

GraalVM

Goals:

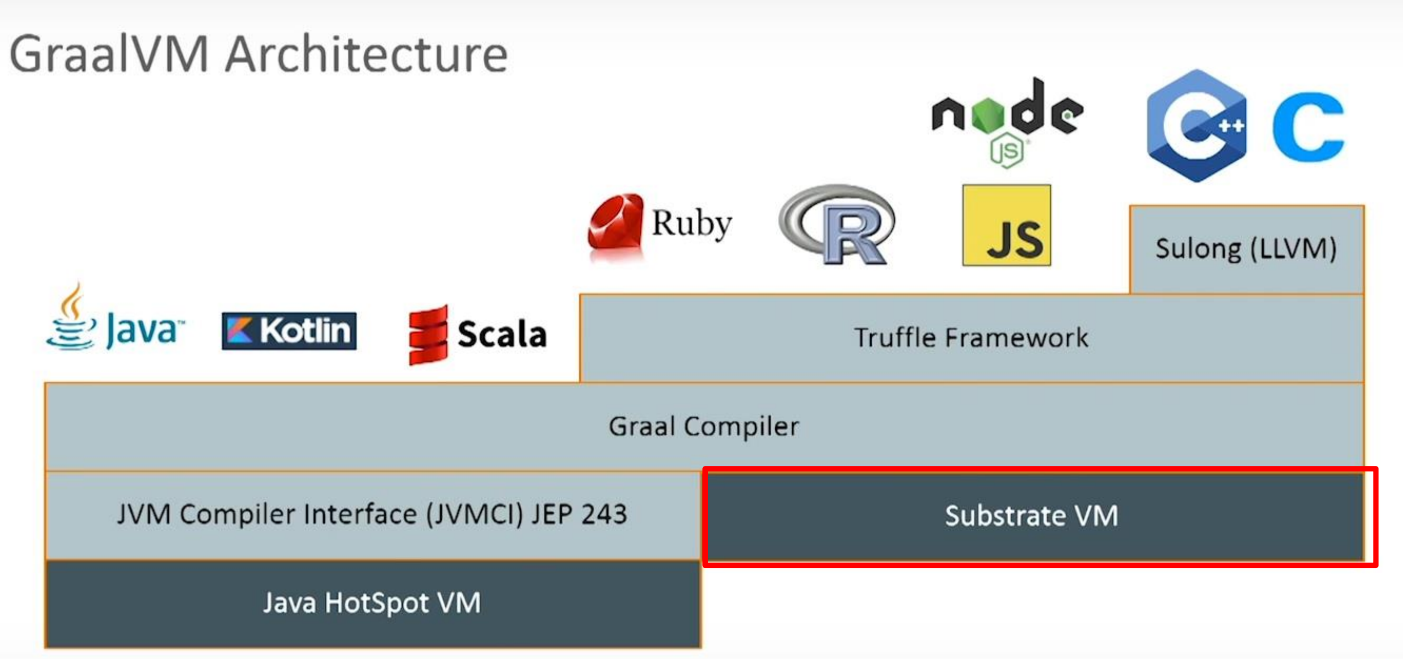
Low footprint ahead-of-time mode for JVM-based languages

High performance for all languages

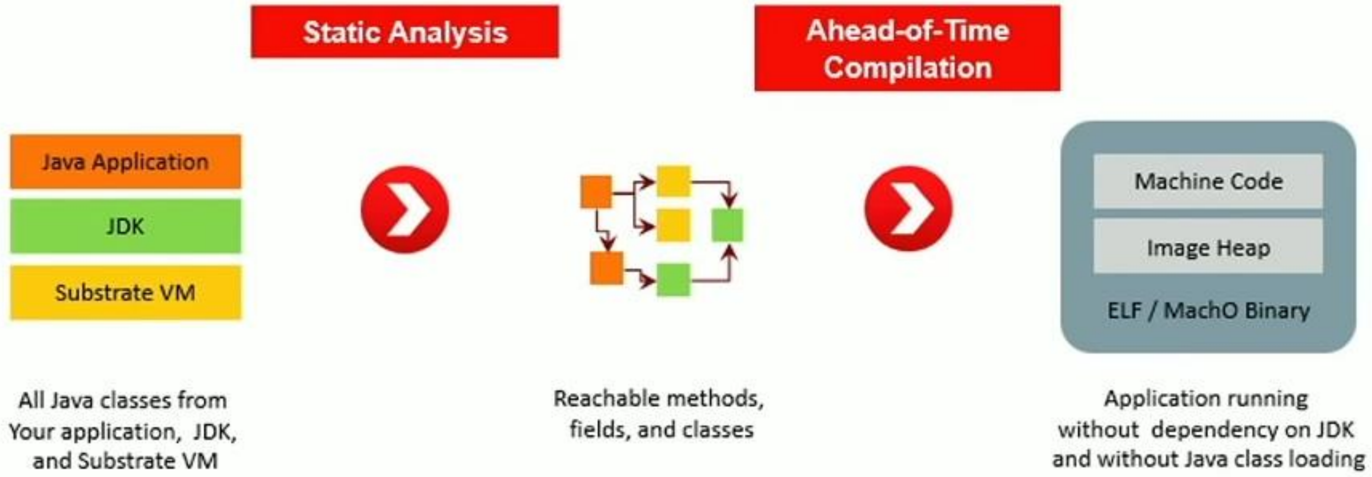
Convenient language interoperability and polyglot tooling

Source: „Everything you need to know about GraalVM by Oleg Šelajev & Thomas Wuerthinger" https://www.youtube.com/watch?v=ANN9rxYo5Hg

GraalVM Architecture



SubstrateVM



Source: Oleg Šelajev, Thomas Wuerthinger, Oracle: “Deep dive into using GraalVM for Java and JavaScript”
https://www.youtube.com/watch?v=a-XEZobXspo

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

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.

Support of GraalVM native images in Frameworks

Spring Native project for Spring (Boot)

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

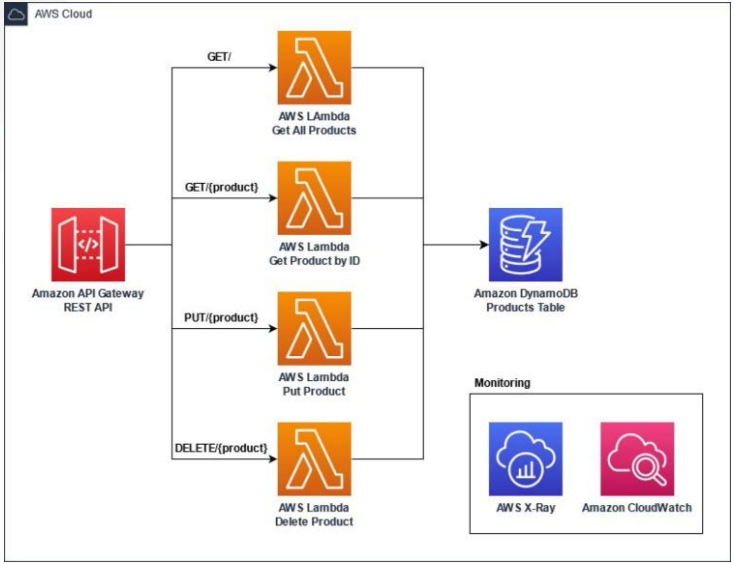
Micronaut: a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications.

Helidon: a cloud-native, open-source set of Java libraries for writing microservices that run on a fast web core powered by Netty.

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
- Provide GraalVM Native Image support out of the box (Gradle and Maven plugins)

Lambda demo with common Java application frameworks



Cold start with Corretto Java 11

Framework	p50	p90	p99
Pure Java	3000-4500	3500-5000	3800-6000
Micronaut	3500-4800	3800-5400	4200-6500
Quarkus	3300-4600	3650-5200	4000-6300
Spring Boot	8000-10000	9200-12000	10000-13000

Cold start with GraalVM Native Image with using Custom Runtime

Framework	p50	p90	p99
Pure Java	433	470	531
Micronaut	604	659	700
Quarkus	437	476	520
Spring Boot	621	685	722

https://github.com/aws-samples/serverless-java-frameworks-samples

Frameworks Ready for Graal VM Native Image

GraalVM Docs Community Videos Blog Download

Frameworks Ready for Native Image

The following frameworks are ready to work with GraalVM Native Image. These frameworks also provide an out-of-the-box experience for many third-party libraries and frameworks. For more details on what they offer, please refer to their project launchers.

Micronaut	Project Launcher	Reachability Metadata
Spring	Project Launcher	Reachability Metadata
Quarkus	Project Launcher	
Helidon	Project Launcher	Reachability Metadata

Libraries and Frameworks Tested with Native Image

The following table lists libraries and frameworks from the Java ecosystem that are tested with GraalVM Native Image. Each item in the list is annotated with a *test level*, as follows:

- *Tested (★★)*: The library or framework is continuously tested by its maintainers. (This is the best test level.)
- *Community-tested (★)*: The library or framework is continuously tested as part of the GraalVM Reachability Metadata Repository or some other community-driven project.

If you would like to add your library and framework to this list, open a pull request and add an entry to this file according to this schema.

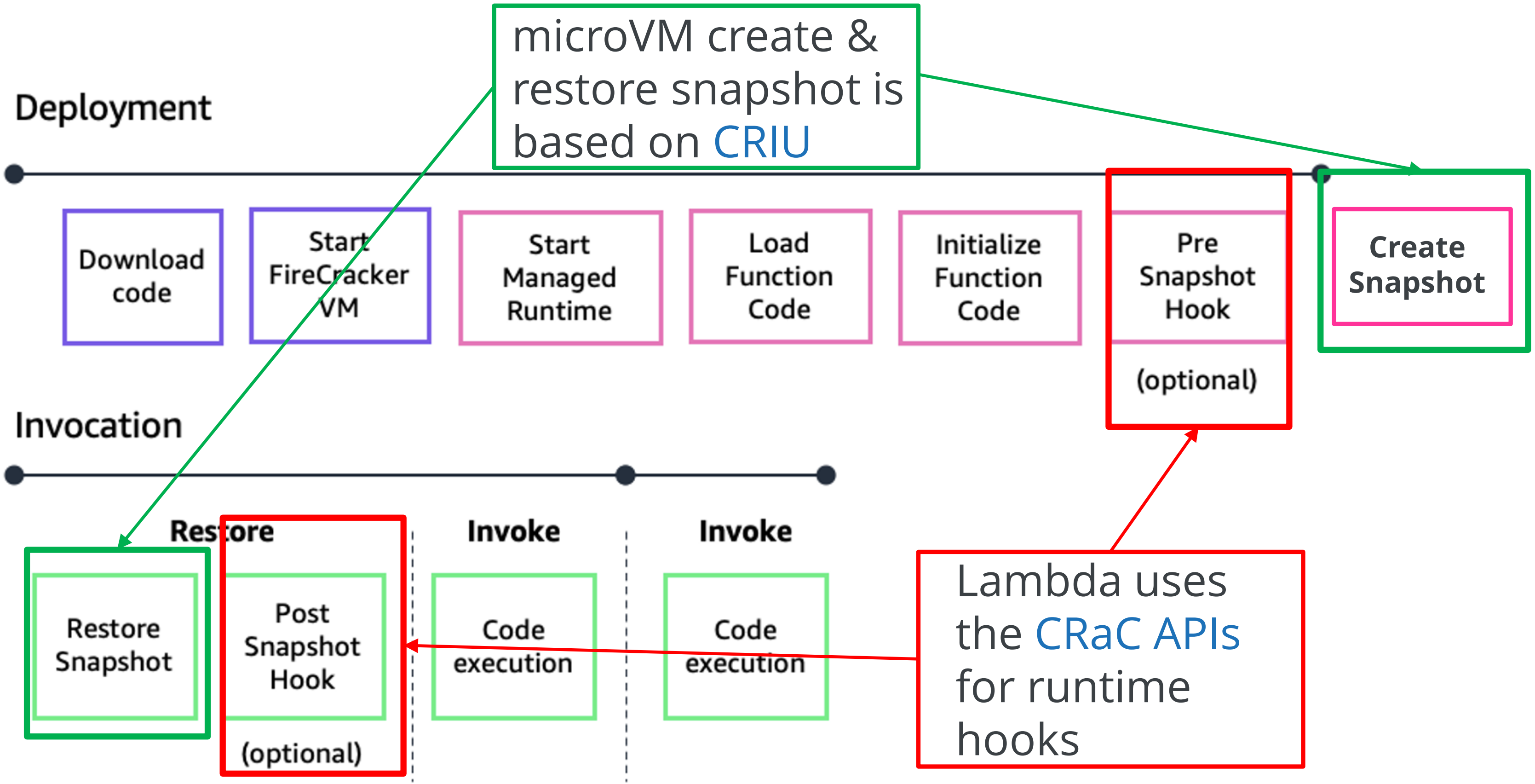
Name	Version	Test Level
ch.qos.logback:logback-classic [1)]	1.2.11 - latest	★
com.datastax.oss:java-driver-core	4.1.5 - latest	★
com.ecwid.consul:consul-api [1)]	1.4.5 - latest	★
com.github.ben-manes.caffeine:caffeine [1)]	3.1.2 - latest	★
com.github.luben:zstd-jni [1)]	1.5.2-5 - latest	★
com.google.protobuf:protobuf-java-util [1)]	3.21.12 - latest	★
com.graphql-java:graphql-java [1)]	19.2 - latest	★
com.graphql-java:graphql-java-extended-validation [1)]	19.1 - latest	★

Graal VM Conclusion

- GraalVM and Frameworks are really powerful with a lot of potential
- GraalVM Native Image improves cold starts and memory footprint significally
- 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 , Java 11 and Java17 (Corretto)

AWS SnapStart

AWS SnapStart Deployment and Invocation



https://aws.amazon.com/de/blogs/compute/reducing-java-cold-starts-on-aws-lambda-functions-with-snapstart/

CRIU (Checkpoint/Restore in Userspace)

- Linux CRIU available since 2012 allows a running application to be paused and restarted at some point later in time, potentially on a different machine.
- The overall goal of the project is to support the migration of containers.
- When performing a checkpoint, essentially, the full context of the process is saved: program counter, registers, stacks, memory-mapped and shared memory
- To restore the application, all this data can be reloaded and (theoretically) it continues from the same point.
- Challenges
  - open files
  - network connections
  - sudden change in the value of the system clock
  - time-based caches

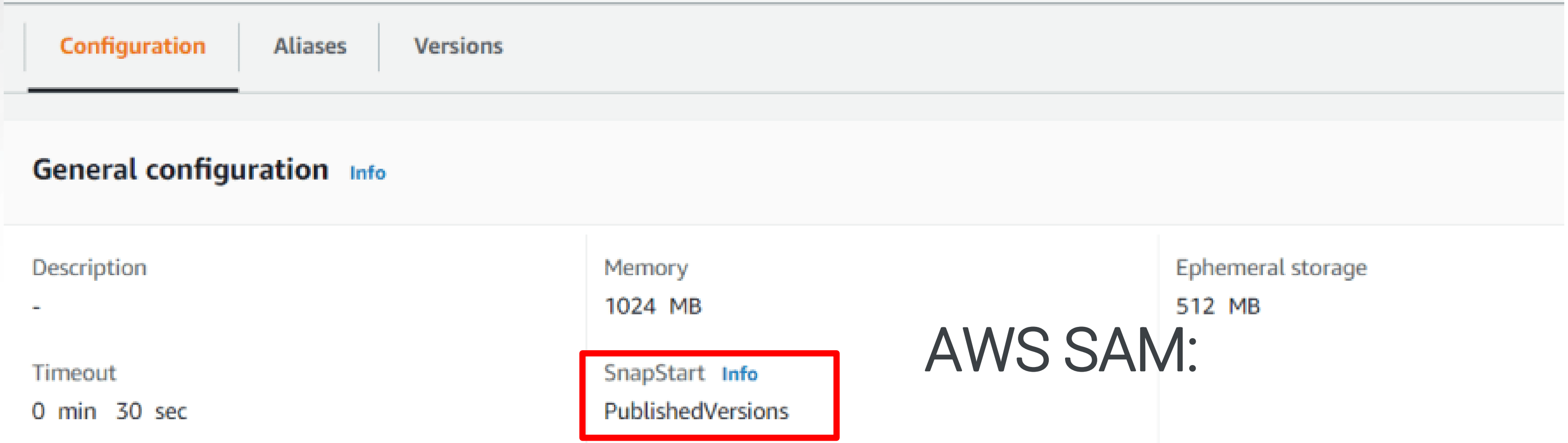
https://criu.org/Main_Page

Ideas behind CRaC (Coordinated Restore at Checkpoint)

- 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

https://www.azul.com/blog/superfast-application-startup-java-on-crac/ https://github.com/CRaC/docs

AWS SnapStart Deployment and Invocation



AWS SAM:

```
GetProductByIdFunction:
 Type: AWS::Serverless::Function
  Properties:
    AutoPublishAlias: SnapStart
    SnapStart:
      ApplyOn: PublishedVersions
```

AWS SnapStart Deployment and Invocation

Cold start with Corretto Java 11

Framework	p50	p90	p99
Pure Java	3000-4500	3500-5000	3800-6000
Micronaut	3500-4800	3800-5400	4200-6500
Quarkus	3300-4600	3650-5200	4000-6300
Spring Boot	8000-10000	9200-12000	10000-13000

Cold start with SnapStart enabled without using priming

Framework	p50	p90	p99
Pure Java	1266	1307	1237
Micronaut	1468	1596	1641
Quarkus	1337	1375	1474
Spring Boot	1222	1877	1880

Source: Vadym Kazulkin
https://dev.to/aws-builders/measuring-java-11-lambda-cold-starts-with-snapstart-part-5-priming-end-to-end-latency-and-deployment-time-jem

AWS SnapStart enabled with Pure Java Priming

```
public class GetProductByIdWithPrimingHandler implements
                RequestHandler<APIGatewayProxyRequestEvent, Optional<Product>>, Resource {

    private static final ProductDao productDao = new DynamoProductDao();

    public GetProductByIdWithPrimingHandler () {
        Core.getGlobalContext().register(this);
    }

    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context) throws Exception {
        productDao.getProduct("0");
    }

    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context) throws Exception {
    }

    @Override
    public Optional<Product> handleRequest(APIGatewayProxyRequestEvent event, Context context) {
        String id = event.getPathParameters().get("id");
        Optional<Product> optionalProduct = productDao.getProduct(id);
```

```
<groupId>io.github.crac</groupId>
<artifactId>org-crac</artifactId>
<version>0.1.3</version>
```

Source: Vadym Kazulkin
https://dev.to/aws-builders/measuring-java-11-lambda-cold-starts-with-snapstart-part-5-priming-end-to-end-latency-and-deployment-time-jem

AWS SnapStart enabled comparison

Cold start with SnapStart enabled without using priming

Framework	p50	p90	p99
Pure Java	1266	1307	1237
Micronaut	1468	1596	1641
Quarkus	1337	1375	1474
Spring Boot	1222	1877	1880

Cold start with SnapStart enabled with using DynamoDB getItem Request in priming

Framework	p50	p90	p99
Pure Java	352	401	434
Micronaut	598	732	756
Quarkus	459	493	510
Spring Boot	601	1065	1174

Source: Vadym Kazulkin
https://dev.to/aws-builders/measuring-java-11-lambda-cold-starts-with-snapstart-part-5-priming-end-to-end-latency-and-deployment-time-jem

AWS SnapStart with Micronaut extended Priming

```java
import com.amazonaws.serverless.proxy.internal.testutils.MockLambdaContext;
import com.amazonaws.serverless.proxy.model.ApiGatewayRequestIdentity;
import com.amazonaws.serverless.proxy.model.AwsProxyRequest;
import com.amazonaws.serverless.proxy.model.AwsProxyRequestContext;

@Singleton
public class ProductAPIResource implements OrderedResource  {

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        System.out.println("Before Checkpoint");
        try (MicronautLambdaHandler micronautLambdaHandler = new MicronautLambdaHandler()) {
            micronautLambdaHandler.handleRequest(getAwsProxyRequest(), new MockLambdaContext());
        }
    }

    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
    }

    private static AwsProxyRequest getAwsProxyRequest () {
        final AwsProxyRequest awsProxyRequest = new AwsProxyRequest ();
        awsProxyRequest.setHttpMethod("GET");
        awsProxyRequest.setPath("/products/0");
        awsProxyRequest.setResource("/products/{id}");
        awsProxyRequest.setPathParameters(Map.of("id","0"));
        final AwsProxyRequestContext awsProxyRequestContext = new AwsProxyRequestContext();
        final ApiGatewayRequestIdentity apiGatewayRequestIdentity= new ApiGatewayRequestIdentity();
        apiGatewayRequestIdentity.setApiKey("blabla");
        awsProxyRequestContext.setIdentity(apiGatewayRequestIdentity);
        awsProxyRequest.setRequestContext(awsProxyRequestContext);
        return awsProxyRequest;
    }
```

Source: Vadym Kazulkin
https://dev.to/aws-builders/measuring-java-11-lambda-cold-starts-with-snapstart-part-6-priming-the-request-invocation-30od

AWS SnapStart enabled with Priming comparison

Cold start with SnapStart enabled with using the priming of DynamoDB getItem Request vs priming of the whole request invocation

Framework	d p50	r p50	d p90	r p90	d p99	r p99
Pure Java	352	352	401	401	434	434
Micronaut	598	432	732	516	756	526
Quarkus	459	413	493	458	510	500
Spring Boot	601	419	1065	583	1174	622

Source: Vadym Kazulkin
https://dev.to/aws-builders/measuring-java-11-lambda-cold-starts-with-snapstart-part-6-priming-the-request-invocation-30od

AWS SnapStart Challenges & Limitations

One big challenge: not the complete cold start time is shown in the CloudWatch queries measuring it with SnapStart enabled

- Snapshot restore outside of Lambda are currently not captured
- Measure end to end Amazon API Gateway request latency to see

  the total cold +warm start

AWS SnapStart Challenges & Limitations

- SnapStart supports the Java 11 and 17 (Corretto) managed runtime only
- Deployment with SnapStart enabled takes more than 2,5 minutes additionally
- Snapshot is deleted from cache if Lambda function is not invoked for 14 days
- Pricing is a bit difficult to understand
- SnapStart currently does not support :
    - Provisioned concurrency
    - arm64 architecture (supports only x86)
    - Amazon Elastic File System (Amazon EFS)
    - Ephemeral storage greater than 512 MB

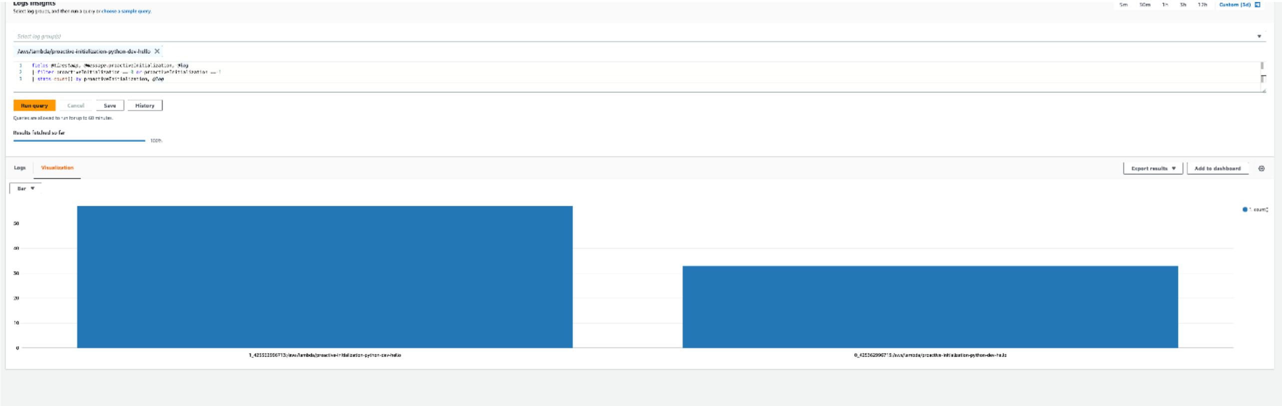
https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html

AWS SnapStart Possible Next Steps

- Perform Priming out of the box without writing the logic on our own
- If snapshot not found do regular cold start and create snapshot under the hood
- Currently snapshot is taken after 1 or several Lambda invocations
    - No C2 compiler optimization took place -> no peak performance to expect
    - Peak performance can be achieved after 10.000 function invocations
    - Optionally provide the possibility to snapshot the function after C2 finished optimization
        - Big trade off involved between additional Lambda cost, deployment frequency and duration until snapshot is taken and function performance gain

NEW: Lambda Proactive initialization

In June 2023 AWS updated the documentation for the Lambda Function lifecycle and included this new statement: for functions using unreserved (on-demand) concurrency, Lambda may **proactively initialize** a function instance, **even if there's no invocation**. When this happens, you can observe an unexpected time gap between your function's initialization and invocation phases. This gap can appear similar to what you would observe when using **provisioned concurrency**.



Running this query over several days across multiple runtimes and invocation methods, **between 50% and 75% of initializations were *<u>proactive</u>*** (versus 50% to 25% which were true cold starts)

https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html
https://aaronstuyvenberg.com/posts/understanding-proactive-initialization

Accelerate Your Photo Business

[Get in Touch]

www.iplabs.de