Adopting Java for the Serverless world

from the perspective of the AWS developer

By Vadym Kazulkin, ip.labs GmbH

Contact

Vadym Kazulkin, ip.labs GmbH

v.kazulkin@gmail.com

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

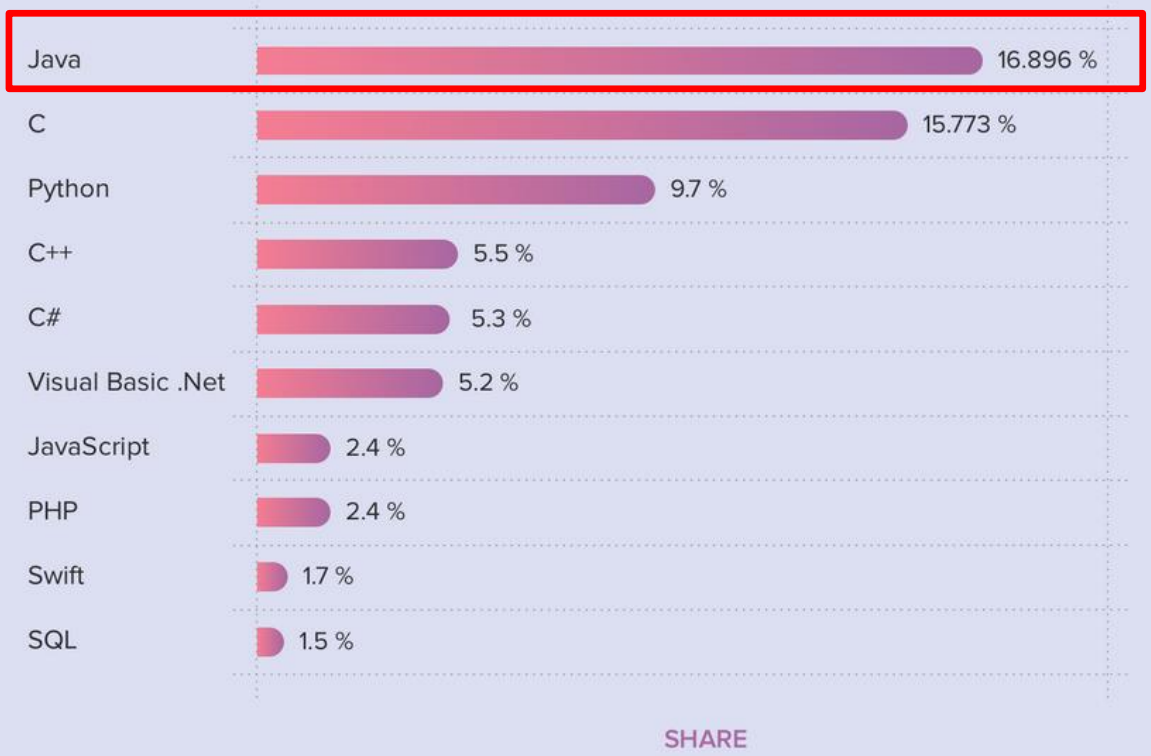
@VKazulkin

Co-Organizer of Java User Group Bonn &
Serverless Bonn Meetup

ip.labs GmbH

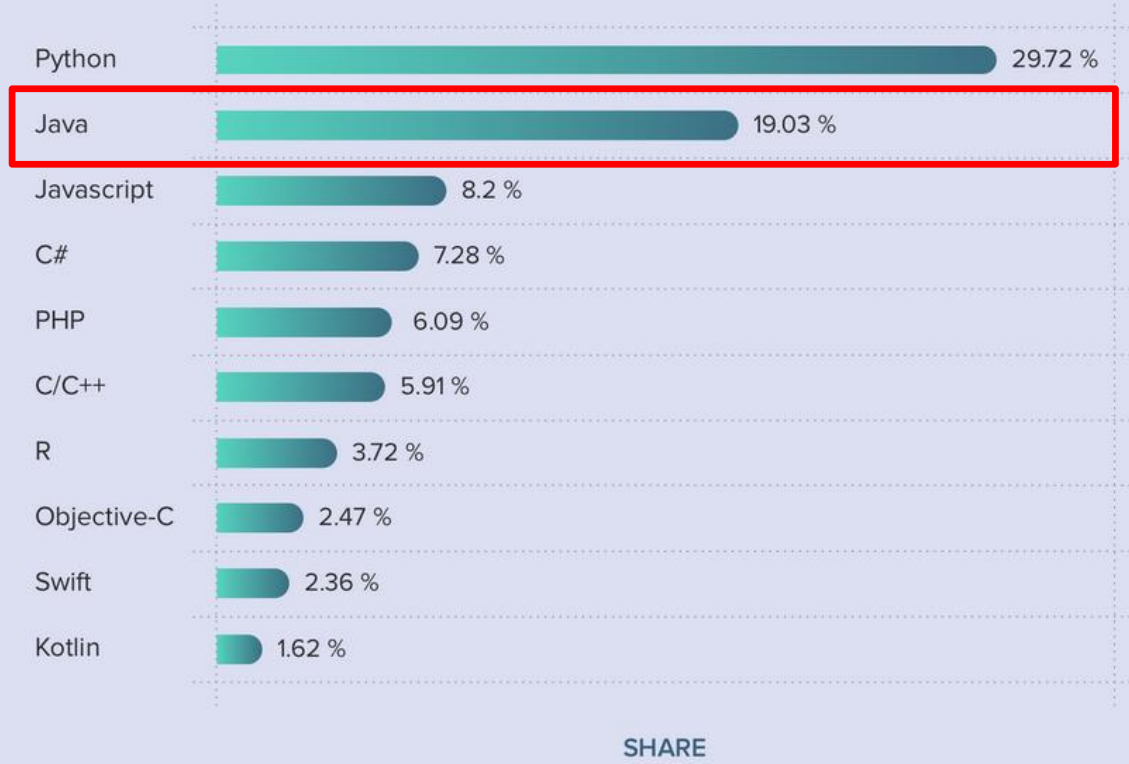
Java popluarity

Top programming languages, TIOBE



Programming language popularity, 2020 by TIOBE

Top programming languages, PYPL



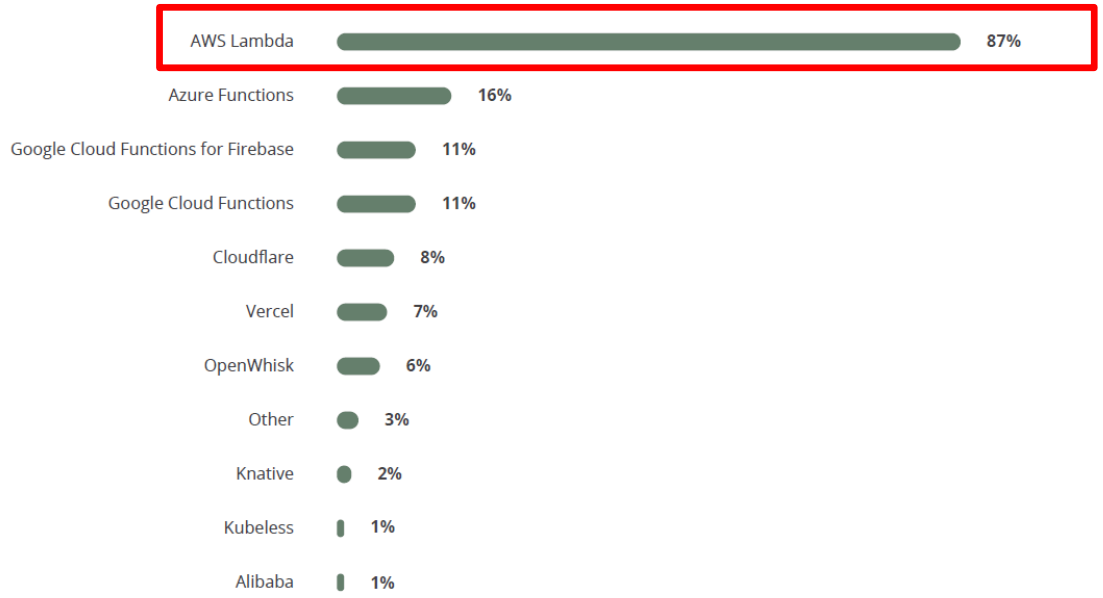
Most popular coding languages, 2020 by PYPL

AWS and Serverless

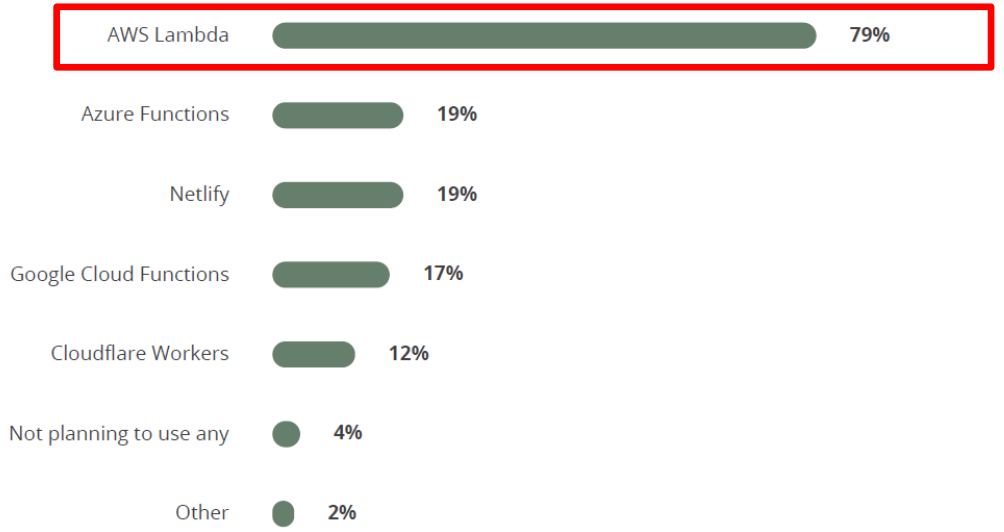
Figure 1. Magic Quadrant for Cloud Infrastructure and Platform Services

TOOLS AND SERVICES

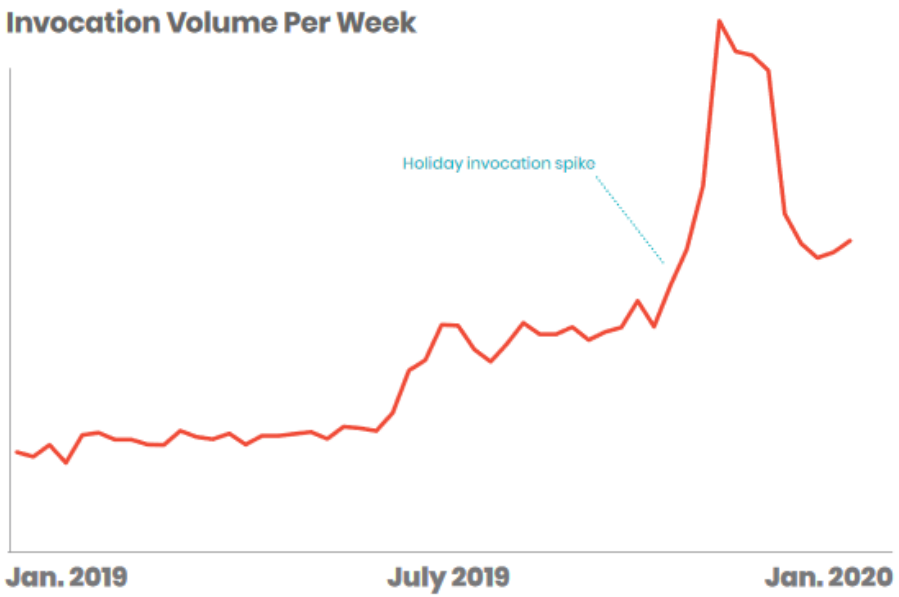
Which serverless vendor do you use?



Which FaaS products are you planning to use in the next 12 months?



The Rise in Serverless



Insights Snapshot

- Serverless adoption continues to rise, with a 209% increase in average weekly invocations over the last 12 months.
- Large spikes in invocations during the holiday season reinforce the serverless use case of auto-scaling to support peak holiday workloads across industries like retail, media, and logistics.

Invocation Growth

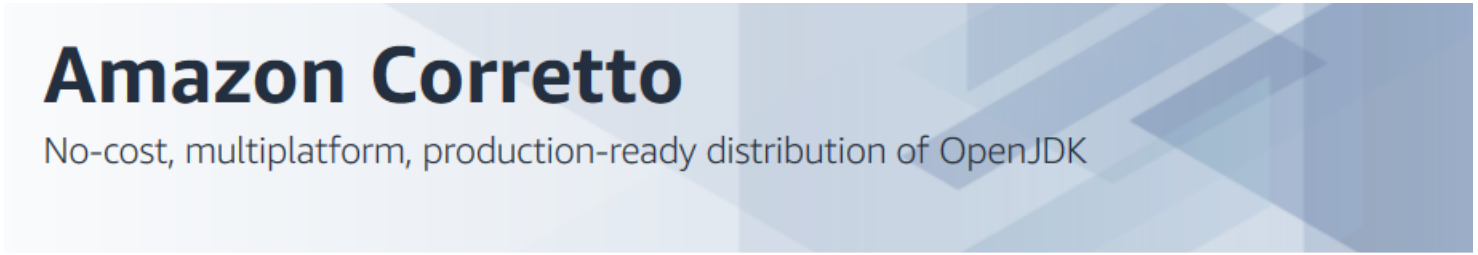
+ 209%

Average weekly invocation volume within sample set over the past 12 months.

Life of the Java Serverless developer on AWS

Java Versions Support

- Java 8
  - with long-term support
- Java 11 (since 2019)



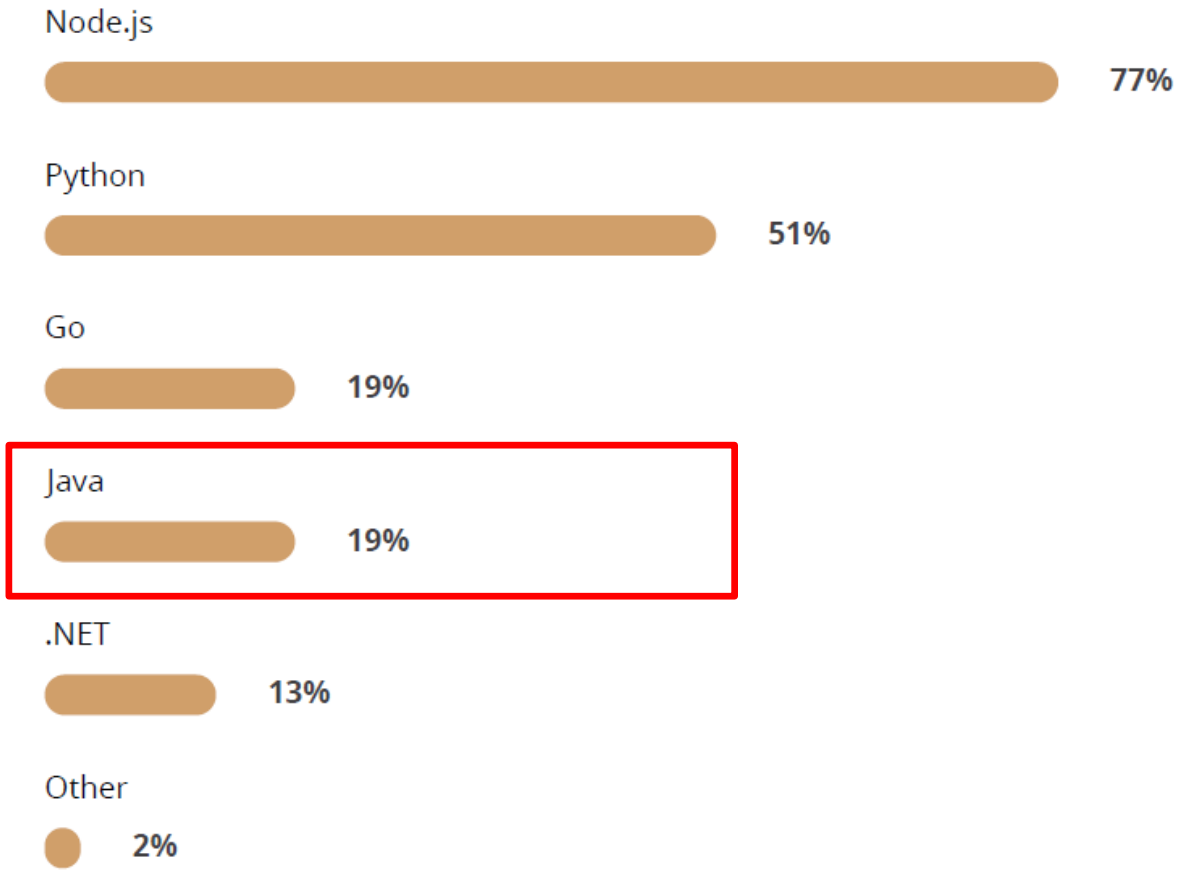
Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). Corretto comes with long-term support that will include performance enhancements and security fixes. Amazon runs Corretto internally on thousands of production services and Corretto 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.

Source: https://aws.amazon.com/de/corretto/

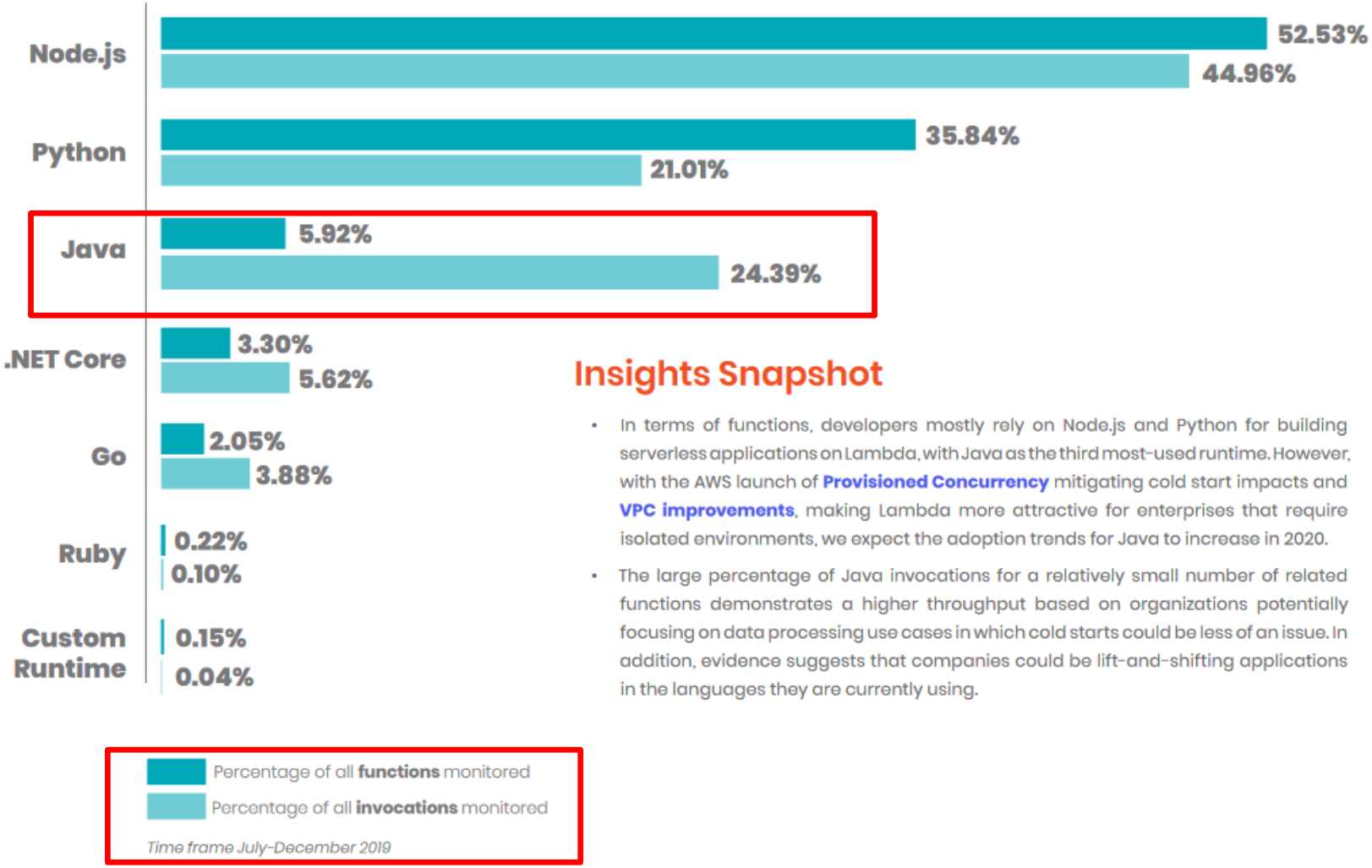
Java ist very fast and mature programming language…

… but Serverless adoption of Java looks like this

What programming languages does your organization use to develop serverless functions?



Lambda Adoption by Runtime

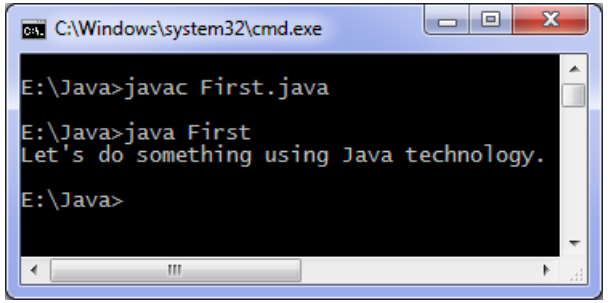


Insights Snapshot

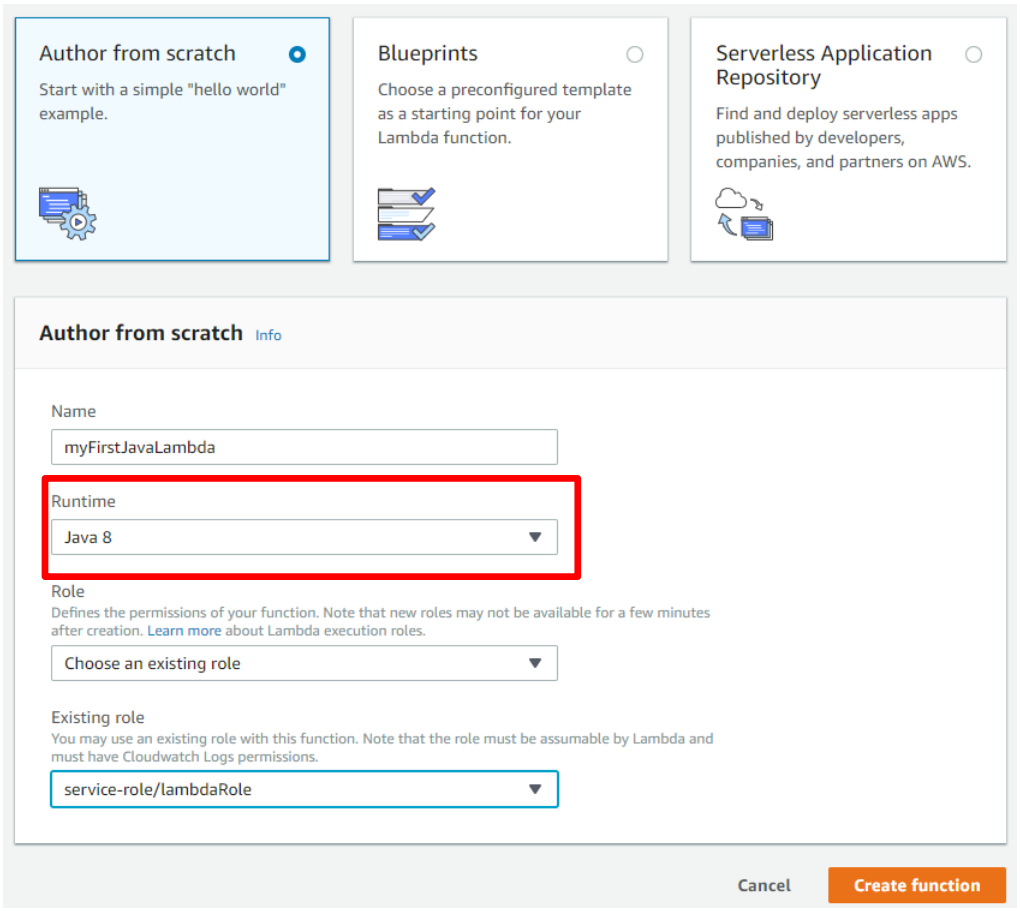
- In terms of functions, developers mostly rely on Node.js and Python for building serverless applications on Lambda, with Java as the third most-used runtime. However, with the AWS launch of **Provisioned Concurrency** mitigating cold start impacts and **VPC improvements**, making Lambda more attractive for enterprises that require isolated environments, we expect the adoption trends for Java to increase in 2020.
- The large percentage of Java invocations for a relatively small number of related functions demonstrates a higher throughput based on organizations potentially focusing on data processing use cases in which cold starts could be less of an issue. In addition, evidence suggests that companies could be lift-and-shifting applications in the languages they are currently using.

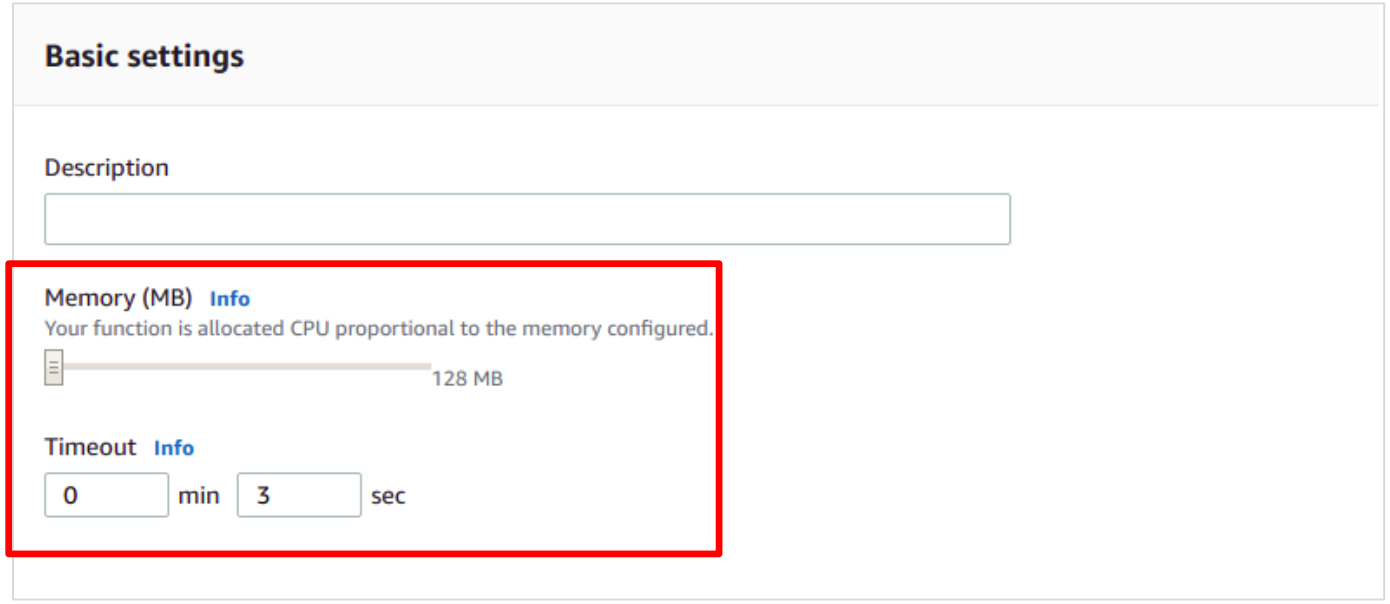
Developers love Java and will be happy

to use it for Serverless



Creating AWS Lambda with Java 1/2





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

ip.labs

Creating AWS Lambda with Java 2/2

```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);
    }
}
```

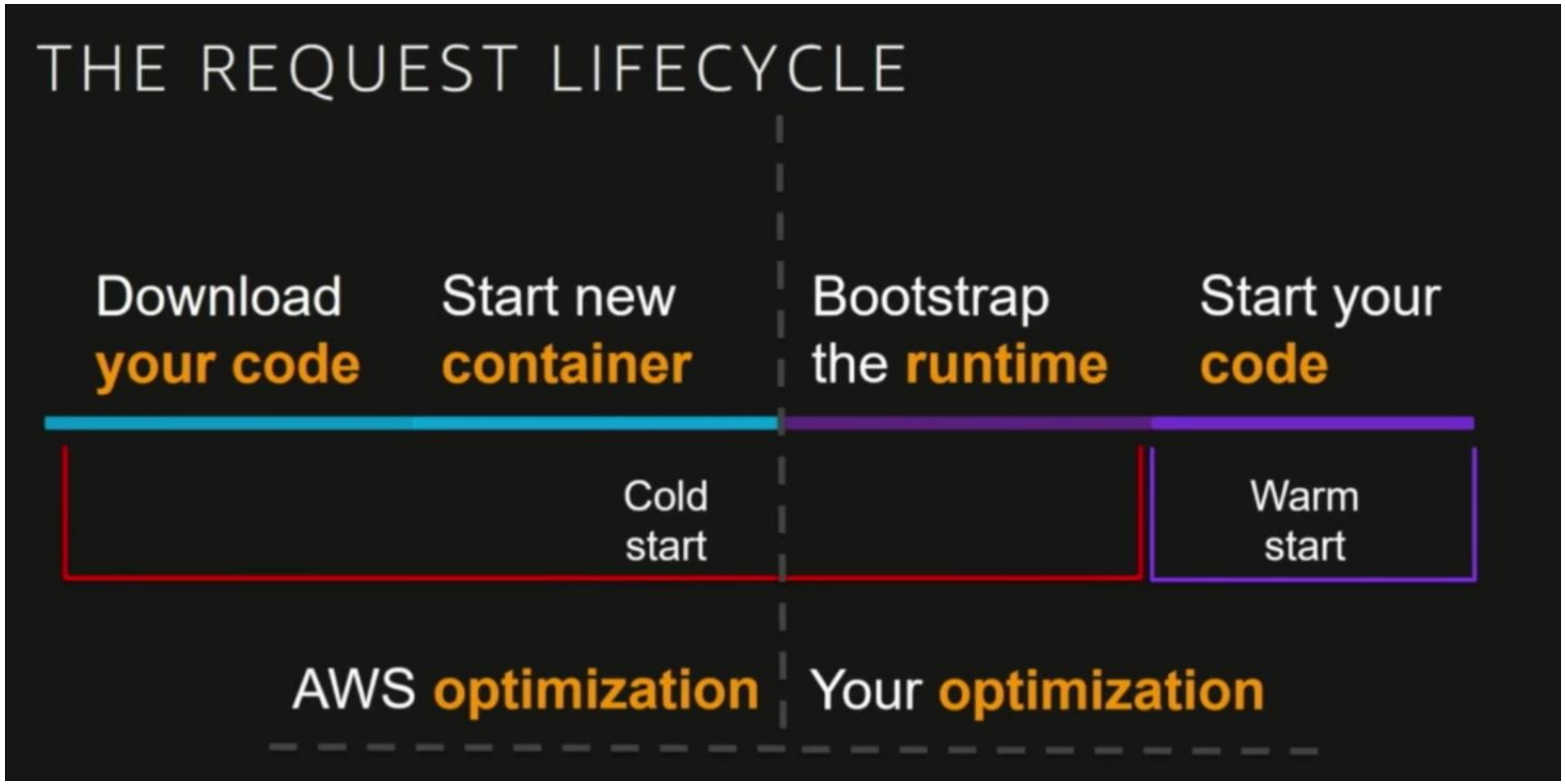
Source https://blog.runscope.com/posts/how-to-write-your-first-aws-lambda-function

Challenge Number 1 with Java is a

big cold-start

Source: https://www.serverless.com/blog/keep-your-lambdas-warm

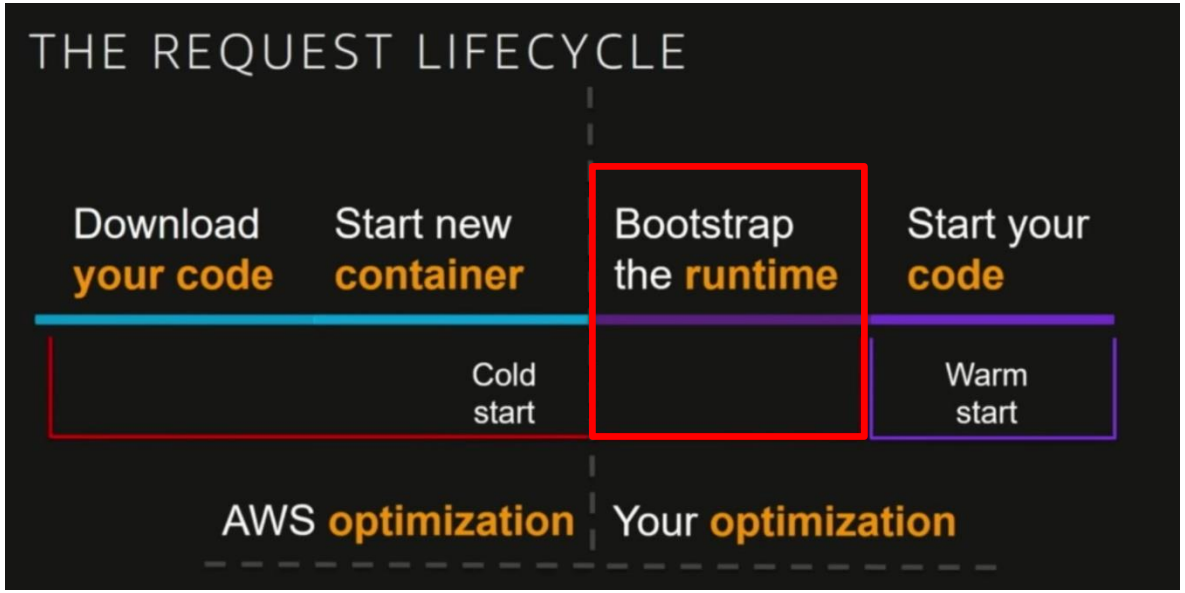
Cold Start

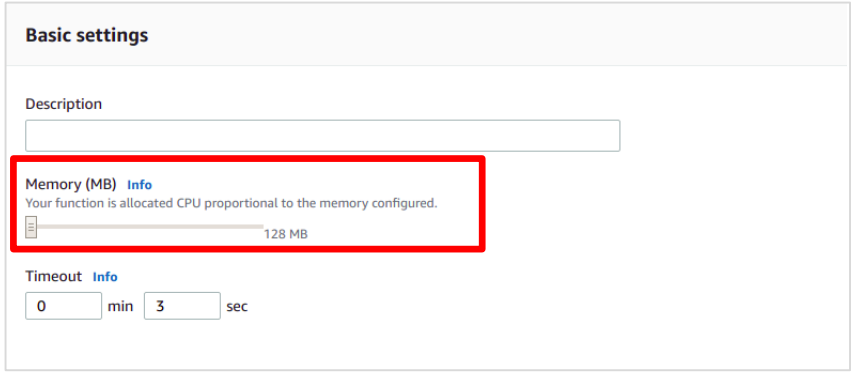


Source: Ajay Nair „Become a Serverless Black Belt” https://www.youtube.com/watch?v=oQFORsso2go

Bootstrap the Java Runtime Phase

- AWS Lambda starts the JVM
- Java runtime loads and initializes handler class
  - Static initializer block of the handler class is executed
  - Boosted host full CPU access up to 10 seconds
- Lambda calls the handler method
  - Full CPU access only approx. from 1.8 GB “assigned” memory to the function

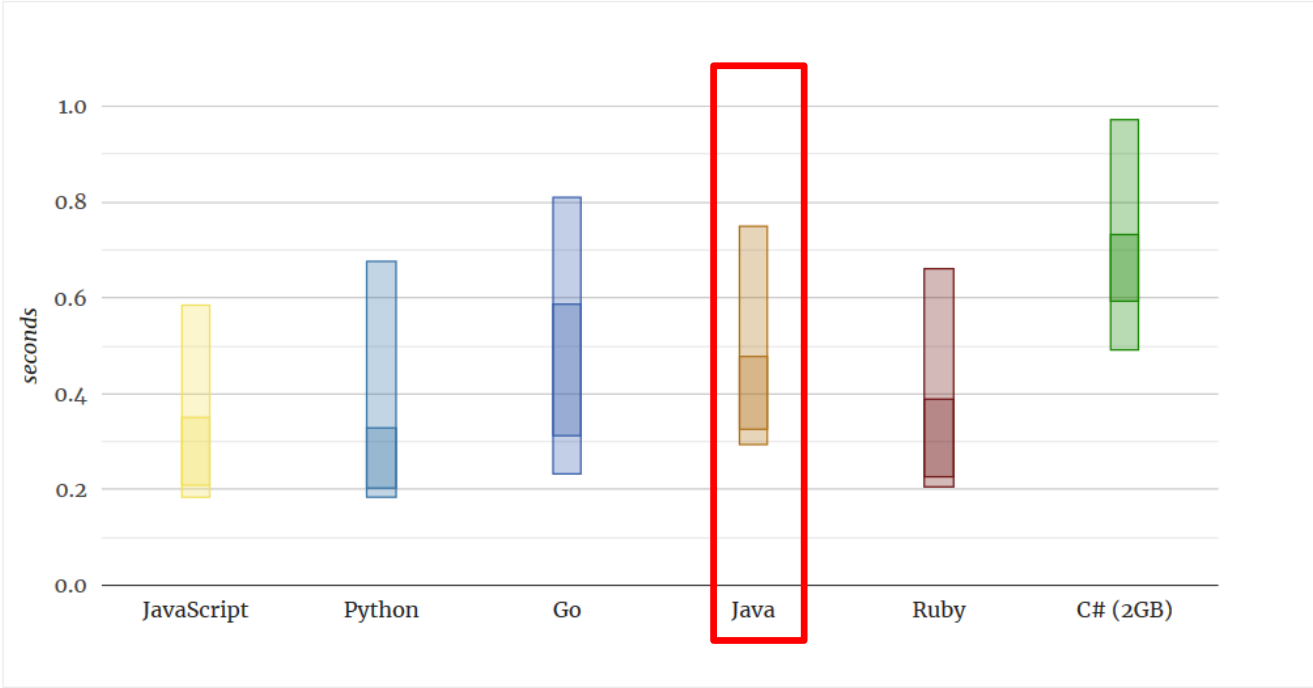




Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8

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

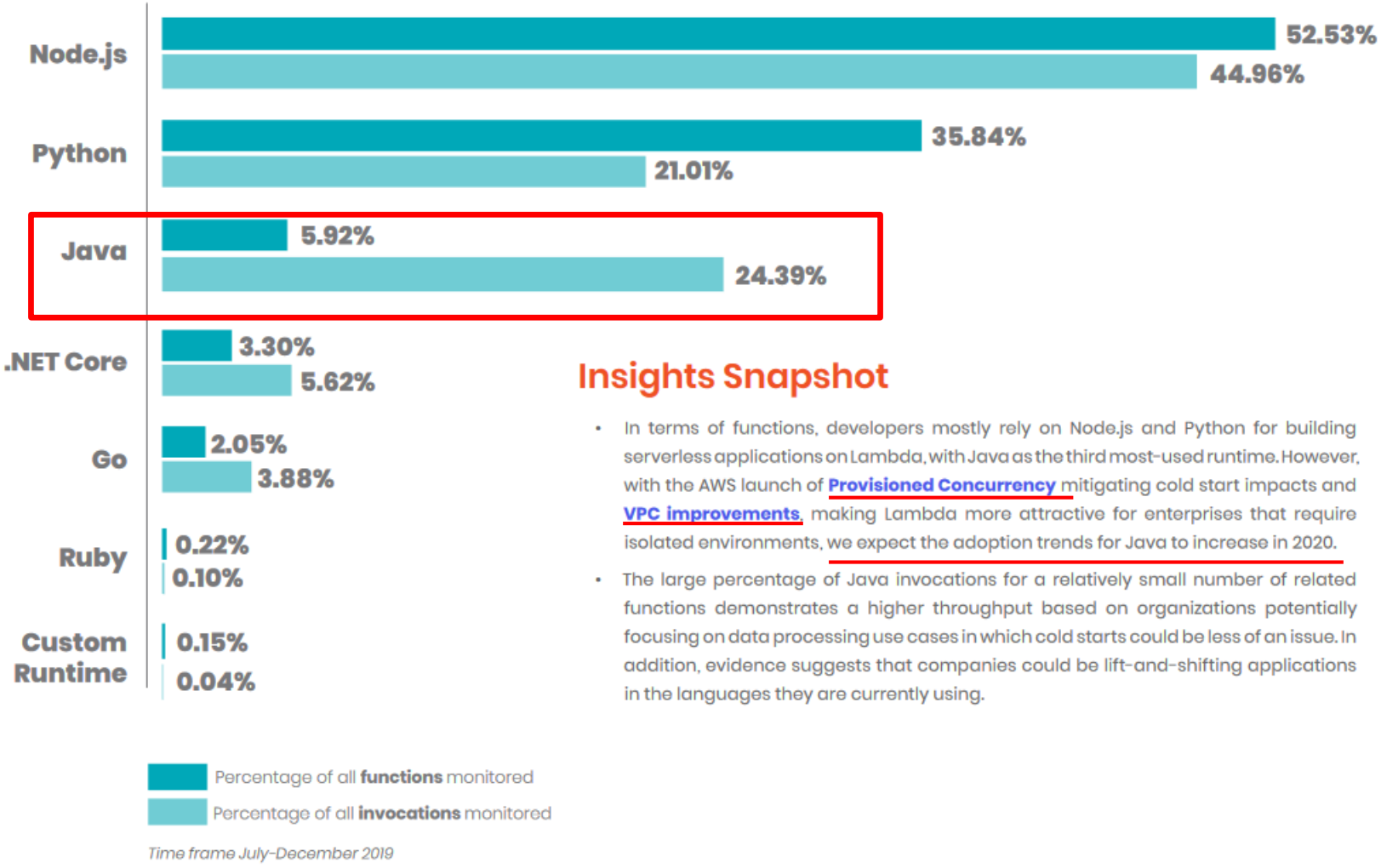
Source: Mikhail Shilkov: „AWS Lambda: Cold Start Duration per Language. 2020 edition" https://mikhail.io/serverless/coldstarts/aws/languages/

Cold start duration with Java

- Below 1 second is best-case cold start duration for very simple Lambda like HelloWorld
- It goes up significantly with more complex scenarios
  - Dependencies to multiple OS projects
  - Clients instantiation to communicate with other (AWS) services (e.g. DynamoDB, SNS, SQS, 3rd party)
- To achieve the minimal cold start duration apply all best practices from these talk
  - Worst-case cold starts can be higher than 10 and even 20 seconds

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/

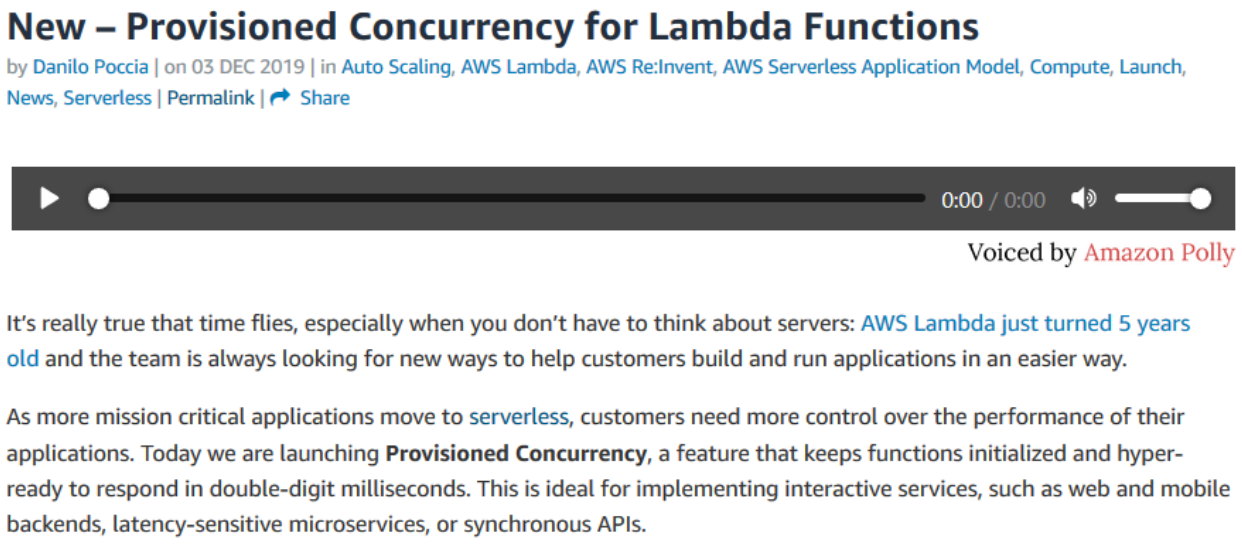
Lambda Adoption by Runtime



Insights Snapshot

- In terms of functions, developers mostly rely on Node.js and Python for building serverless applications on Lambda, with Java as the third most-used runtime. However, with the AWS launch of **Provisioned Concurrency** mitigating cold start impacts and **VPC improvements**, making Lambda more attractive for enterprises that require isolated environments, we expect the adoption trends for Java to increase in 2020.
- The large percentage of Java invocations for a relatively small number of related functions demonstrates a higher throughput based on organizations potentially focusing on data processing use cases in which cold starts could be less of an issue. In addition, evidence suggests that companies could be lift-and-shifting applications in the languages they are currently using.

Provisioned Concurrency for Lambda Functions



New – Provisioned Concurrency for Lambda Functions

by Danilo Poccia | on 03 DEC 2019 | in Auto Scaling, AWS Lambda, AWS Re:Invent, AWS Serverless Application Model, Compute, Launch, News, Serverless | Permalink | Share

0:00 / 0:00

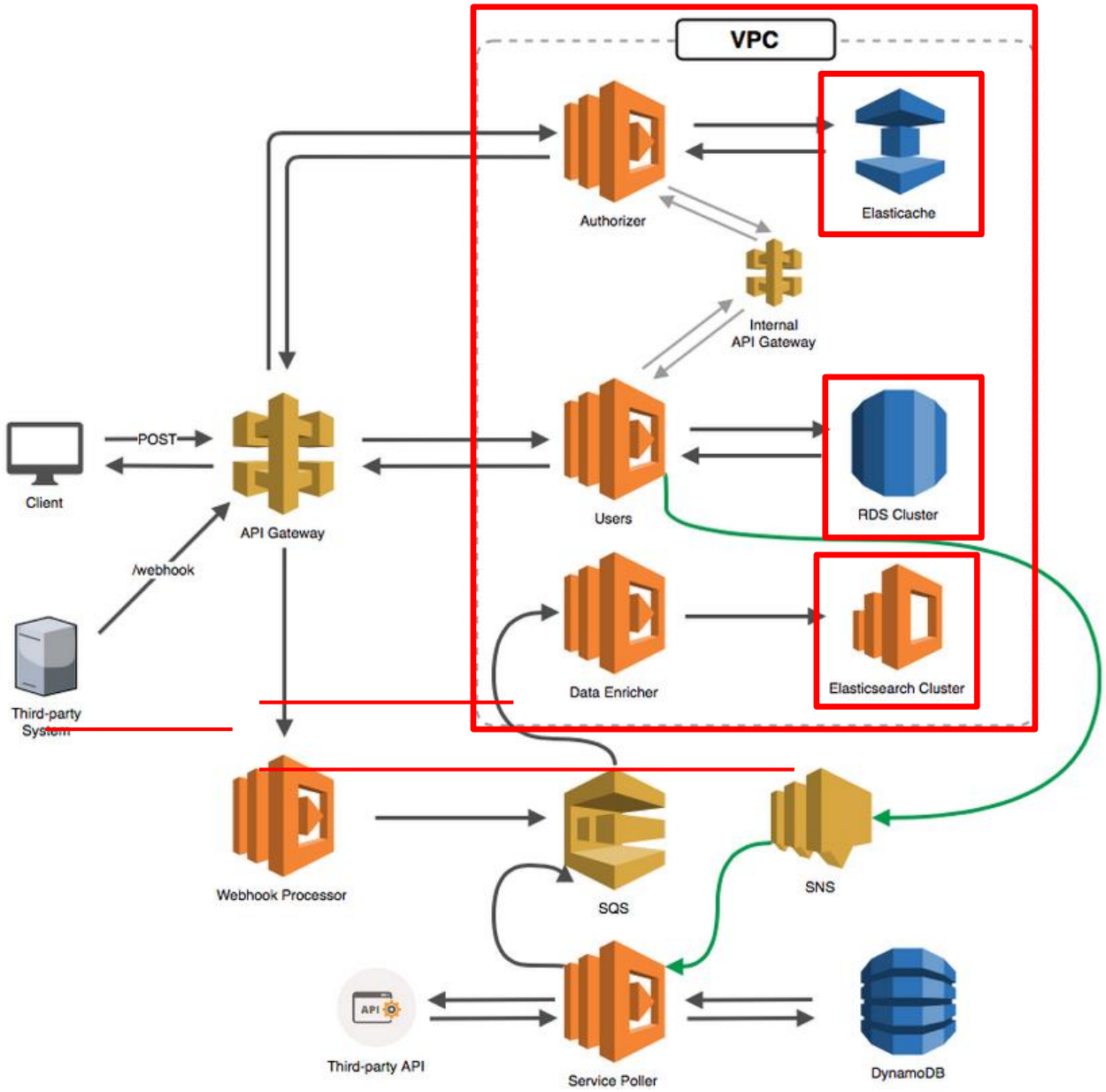
Voiced by Amazon Polly

It's really true that time flies, especially when you don't have to think about servers: AWS Lambda just turned 5 years old and the team is always looking for new ways to help customers build and run applications in an easier way.

As more mission critical applications move to serverless, customers need more control over the performance of their applications. Today we are launching **Provisioned Concurrency**, a feature that keeps functions initialized and hyper-ready to respond in double-digit milliseconds. This is ideal for implementing interactive services, such as web and mobile backends, latency-sensitive microservices, or synchronous APIs.

Source: Yan Cui: https://lumigo.io/blog/provisioned-concurrency-the-end-of-cold-starts/

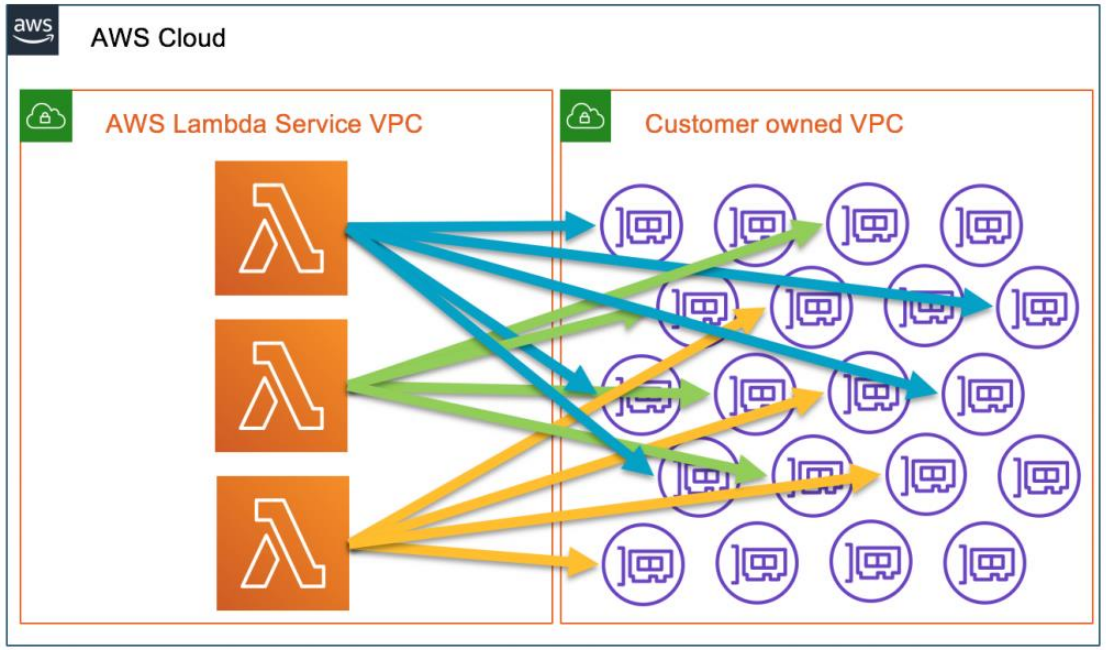
Lambda behind the Virtual Private Cloud (VPC)



Lambda in VPC

As function’s execution environment scales

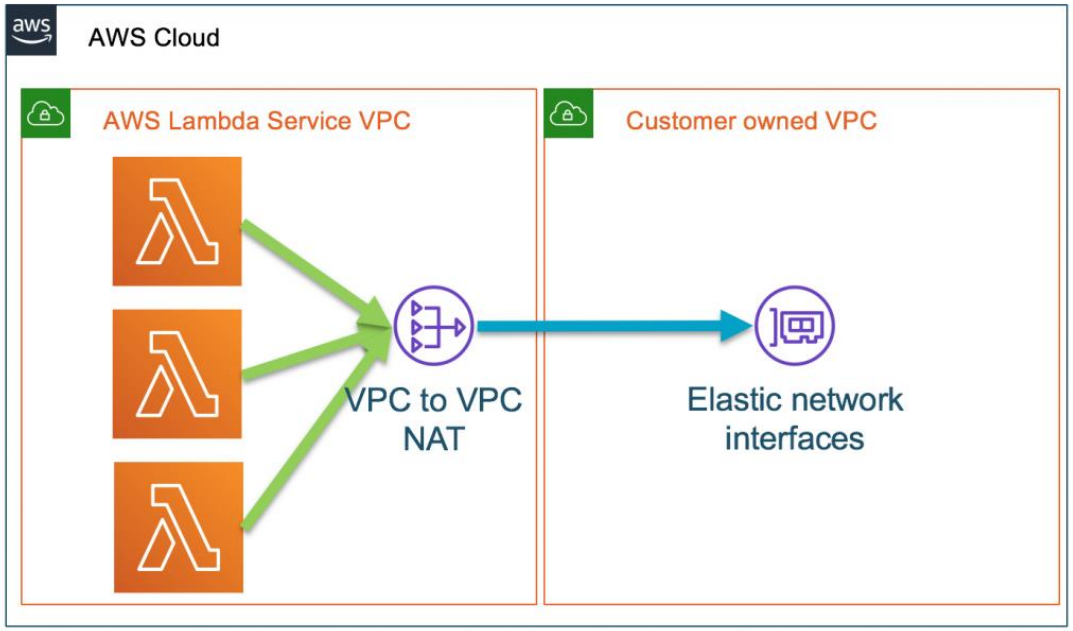
- More network interfaces are created and attached to the Lambda infrastructure
- The exact number of network interfaces created and attached is a factor of your function configuration and concurrency
- Caused additional the cold start up to approx. 10 seconds



Chris Munns: "Announcing improved VPC networking for AWS Lambda functions”
https://aws.amazon.com/de/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/

Lambda in VPC Improvements:

- The network interface creation happens when Lambda function is created or its VPC settings are updated.
- Because the network interfaces are shared across execution environments, only a handful of network interfaces are required per function
- Reduced additional cold start from approx. 10 seconds to below 1 second



Chris Munns: "Announcing improved VPC networking for AWS Lambda functions"
https://aws.amazon.com/de/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/

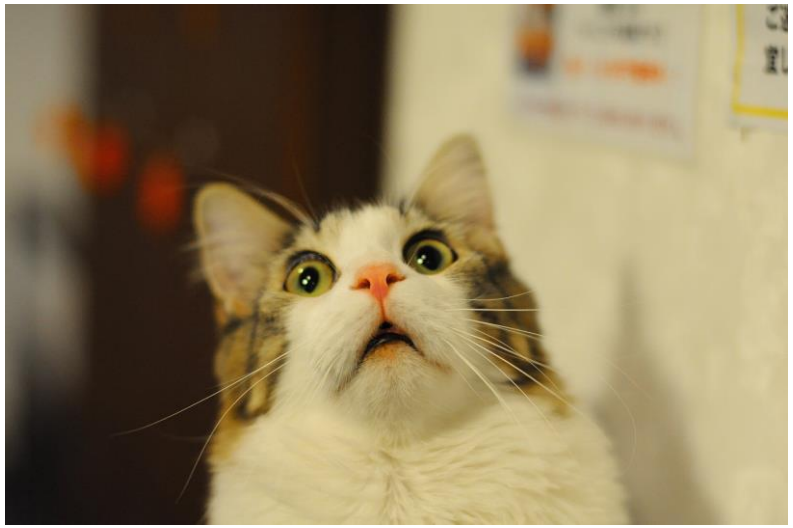
Improvements 1/4

- Switch to the AWS SDK 2.0 for Java
  - Lower footprint and more modular
  - Allows to configure HTTP Client of our choice (e.g. Java own Basic HTTP Client)
- Initialize and prime dependencies during initialization phase
  - Use static initialization in the handler class
- Provide all known values (for building clients e.g. DynamoDBClient) to avoid auto-discovery
  - credential provider, region, endpoint
- Less (dependencies, classes) is more

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/

Improvements 2/4

Avoid Reflection



Or use DI Frameworks like Dagger 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/

Improvements 3/4

Strive for cost optimization

AWS Lambda pricing model

Cost for Lambda

REQUEST

DURATION

Request Tier

$ 0.20

Per 1 Mio Requests

Duration Tier

$ 0.00001667

Per GB-Second

GB-Second

ONE SECOND

ONE GB

Example

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

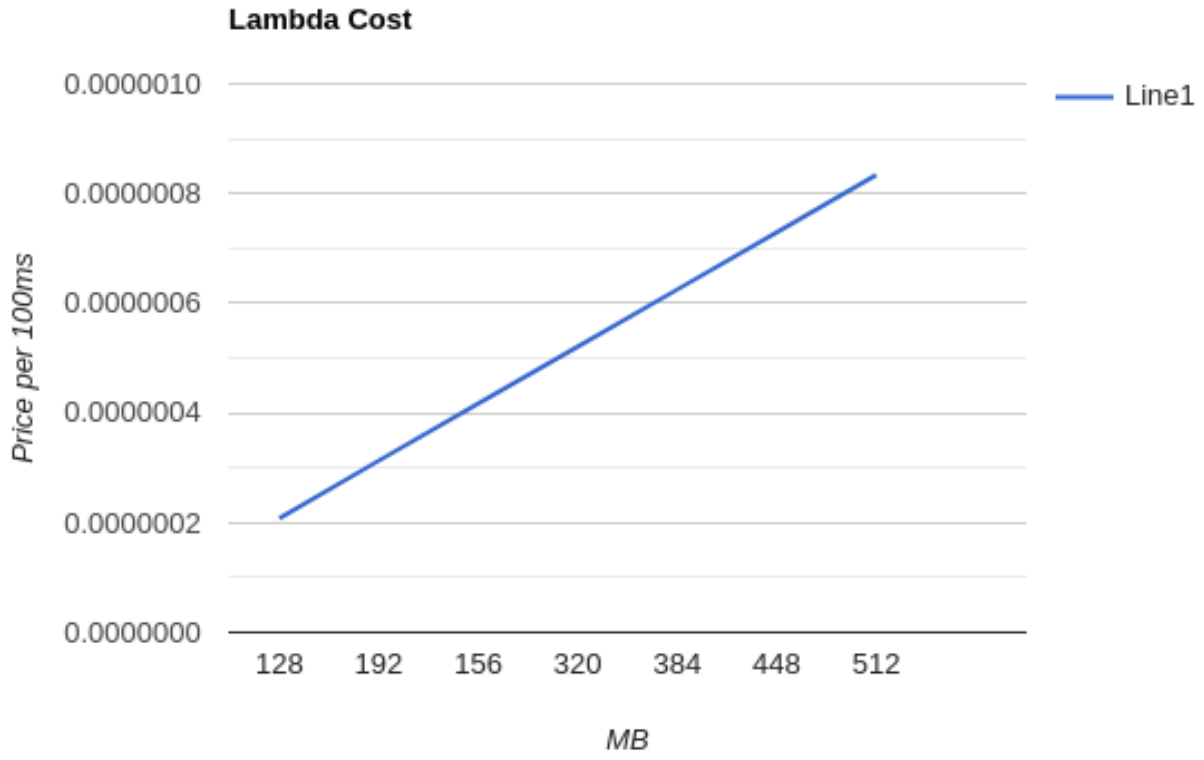
Requests:
$0.20

GB-Seconds:
$1.67

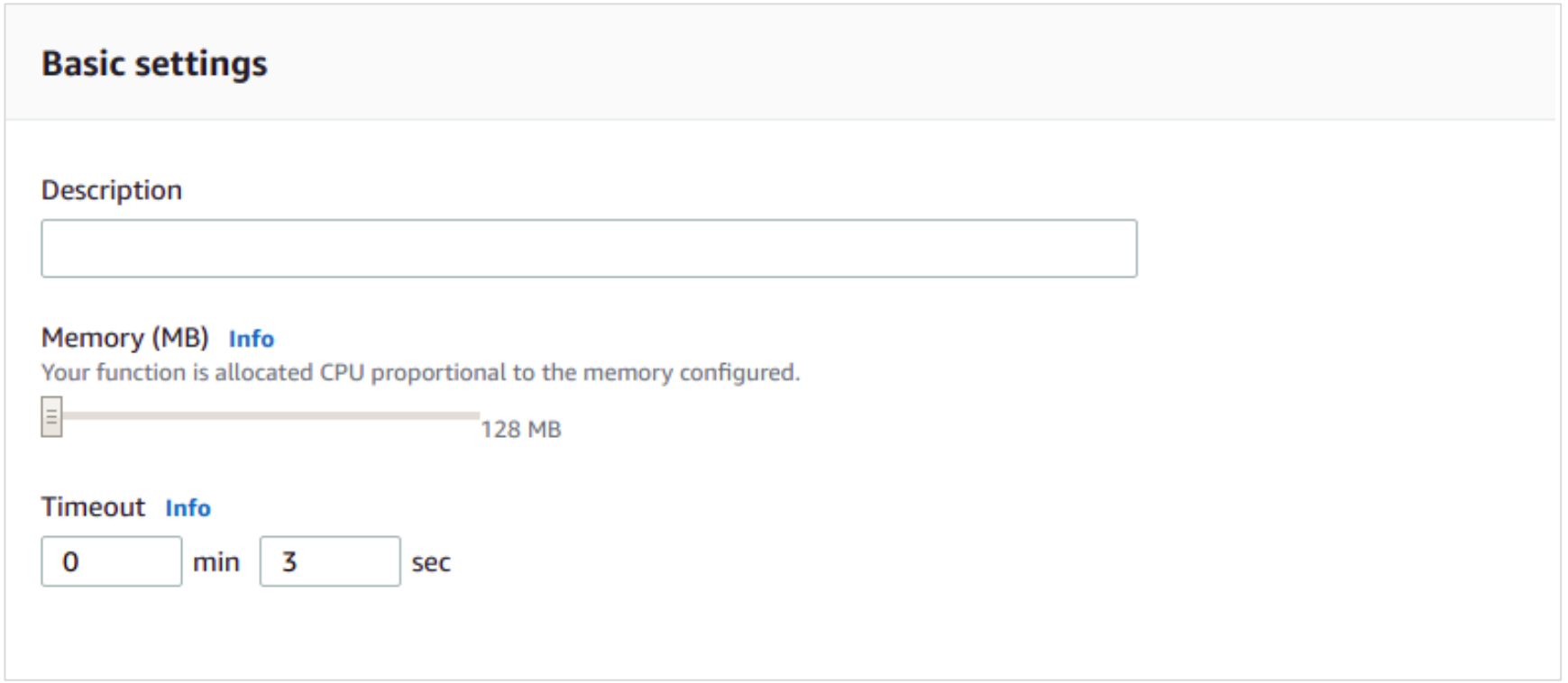
0.5 GiB * 0.2 sec * 1 Mio

= 100 000 GB-Seconds

Cost scales linearly with memory

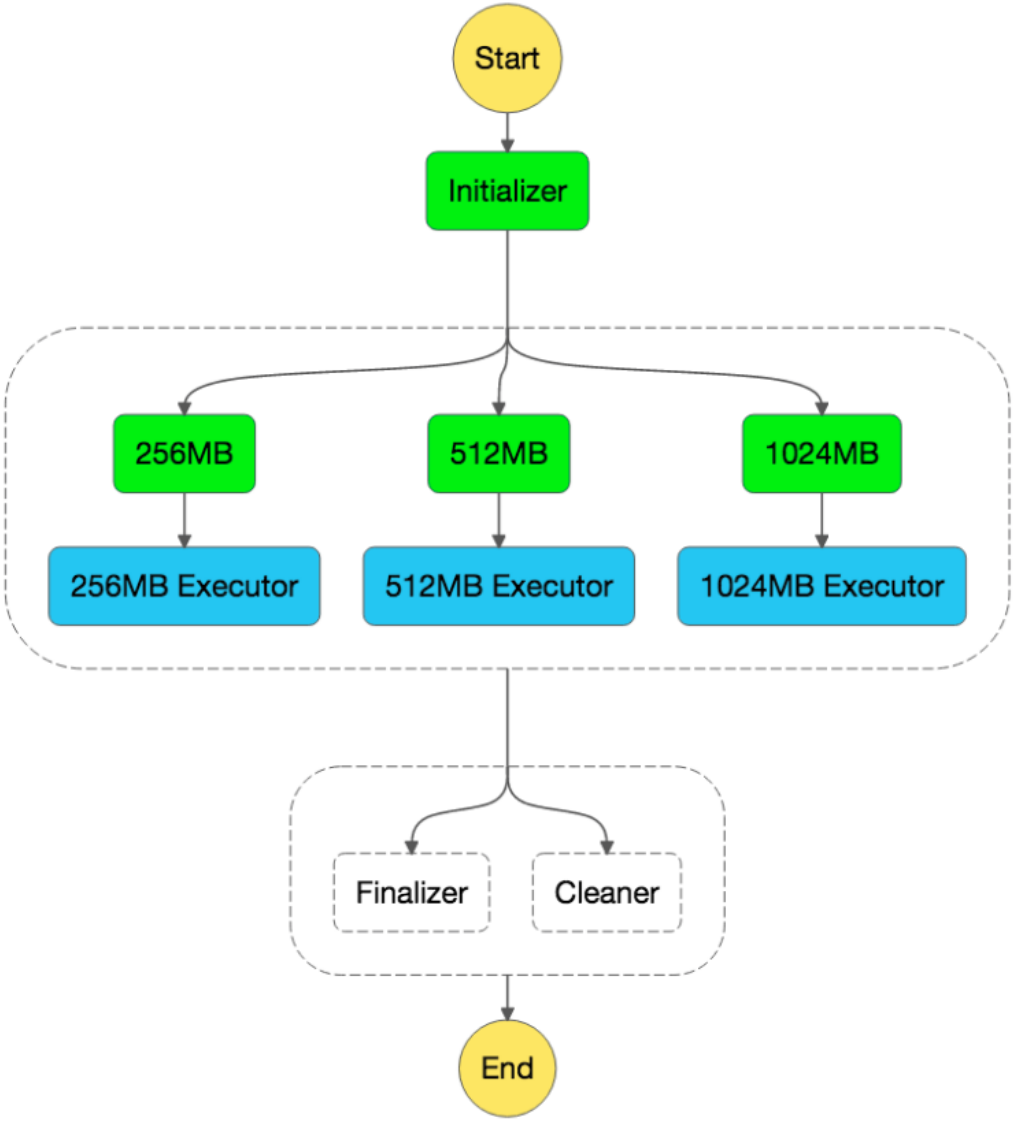


Improvements 4/4
More memory = more expensive?

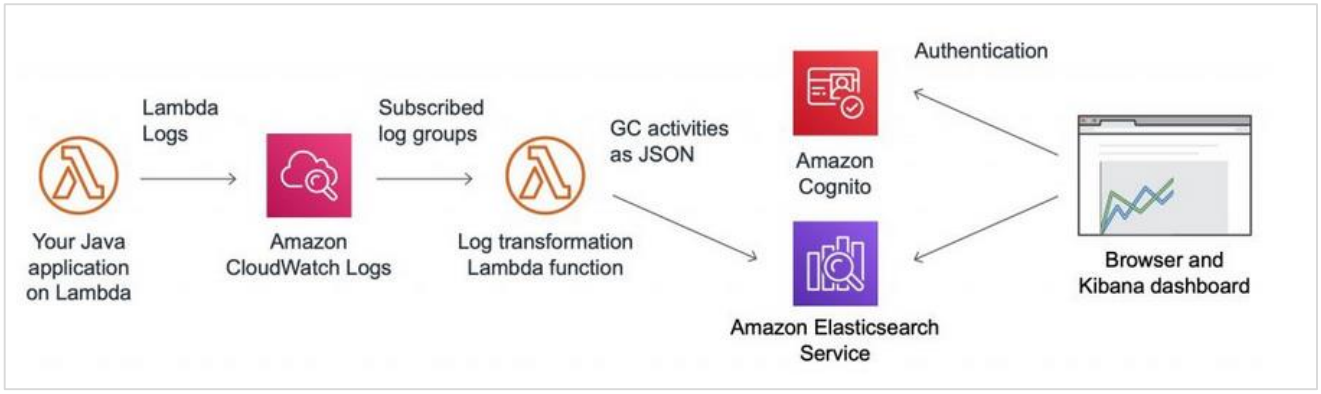


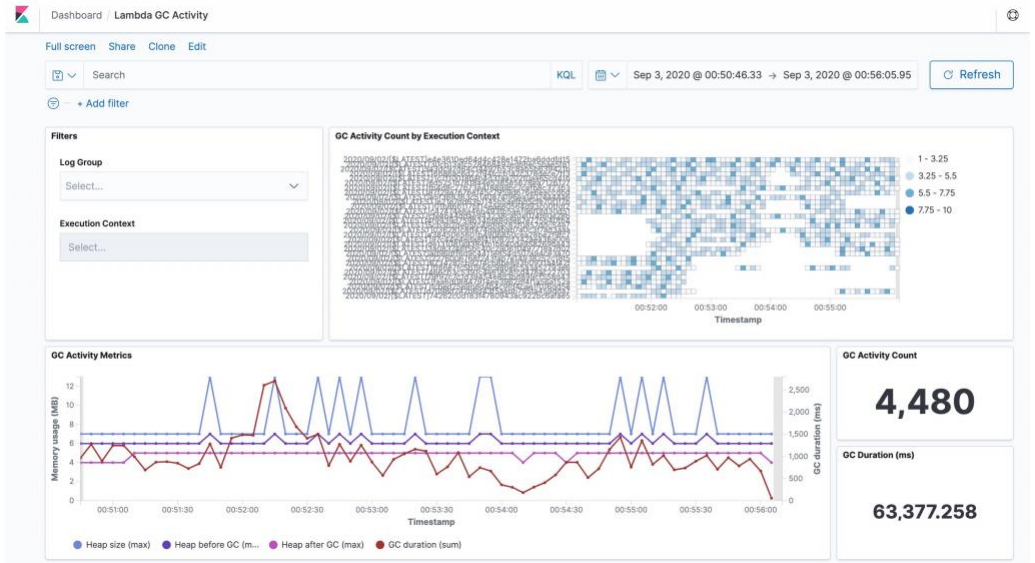
Lambda Power Tuning

- Executes different settings in parallel
- Outputs the optimal setting



Monitor the Java Virtual Machine Garbage Collection on AWS Lambda





Source: Steffen Grunwald „Monitoring the Java Virtual Machine Garbage Collection on AWS Lambda”
https://aws.amazon.com/de/blogs/architecture/field-notes-monitoring-the-java-virtual-machine-garbage-collection-on-aws-lambda/

ip.labs

Cost optimization

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

And both memory and execution time are cost dimensions, when using Serverless in the cloud

GraalVM enters the scene

Source: https://www.graalvm.org/

Project Metropolis

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

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

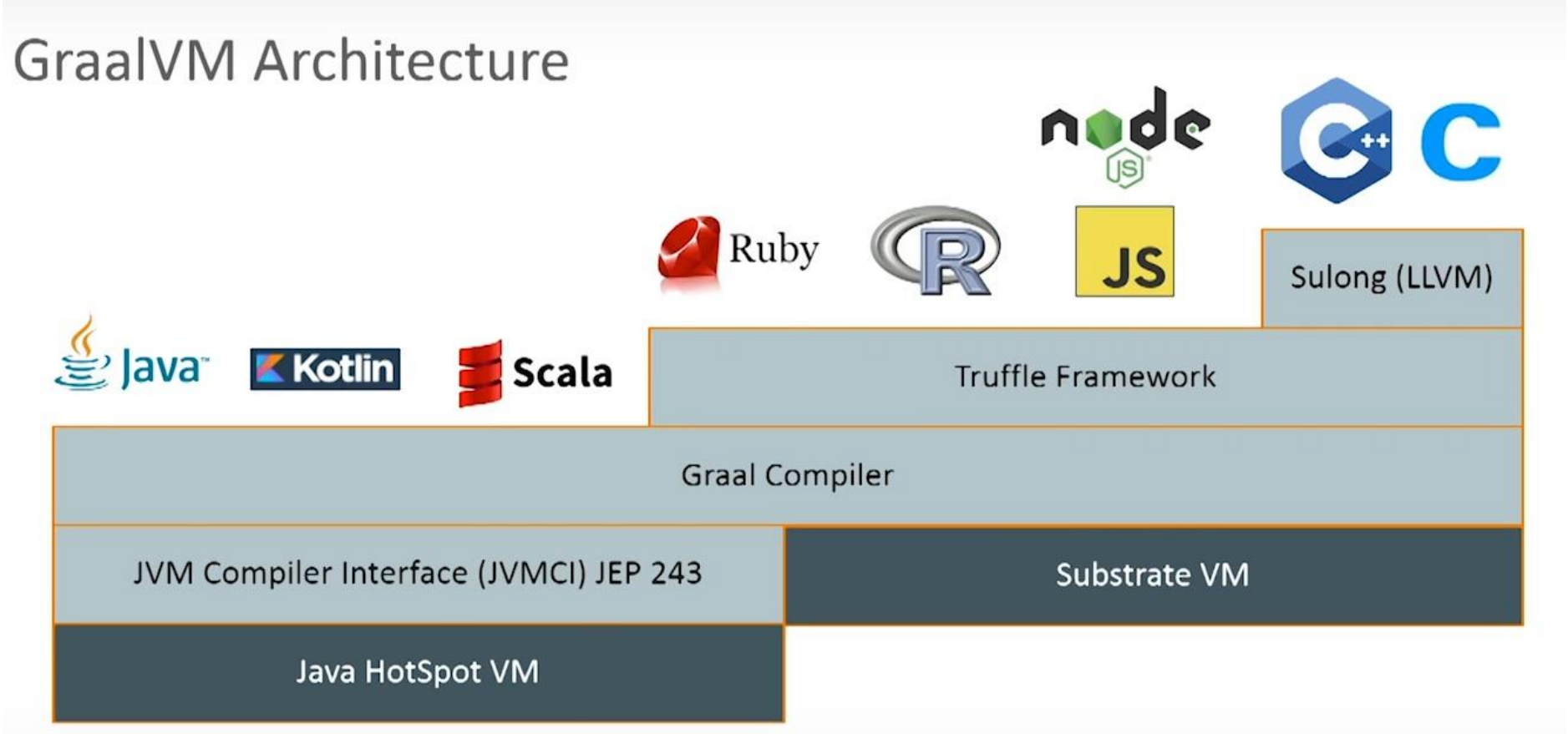
LICENSE

- Oracle Master License Agreement
- Free for evaluation and non-production use
- Contact us for commercial use and support options

BENEFITS

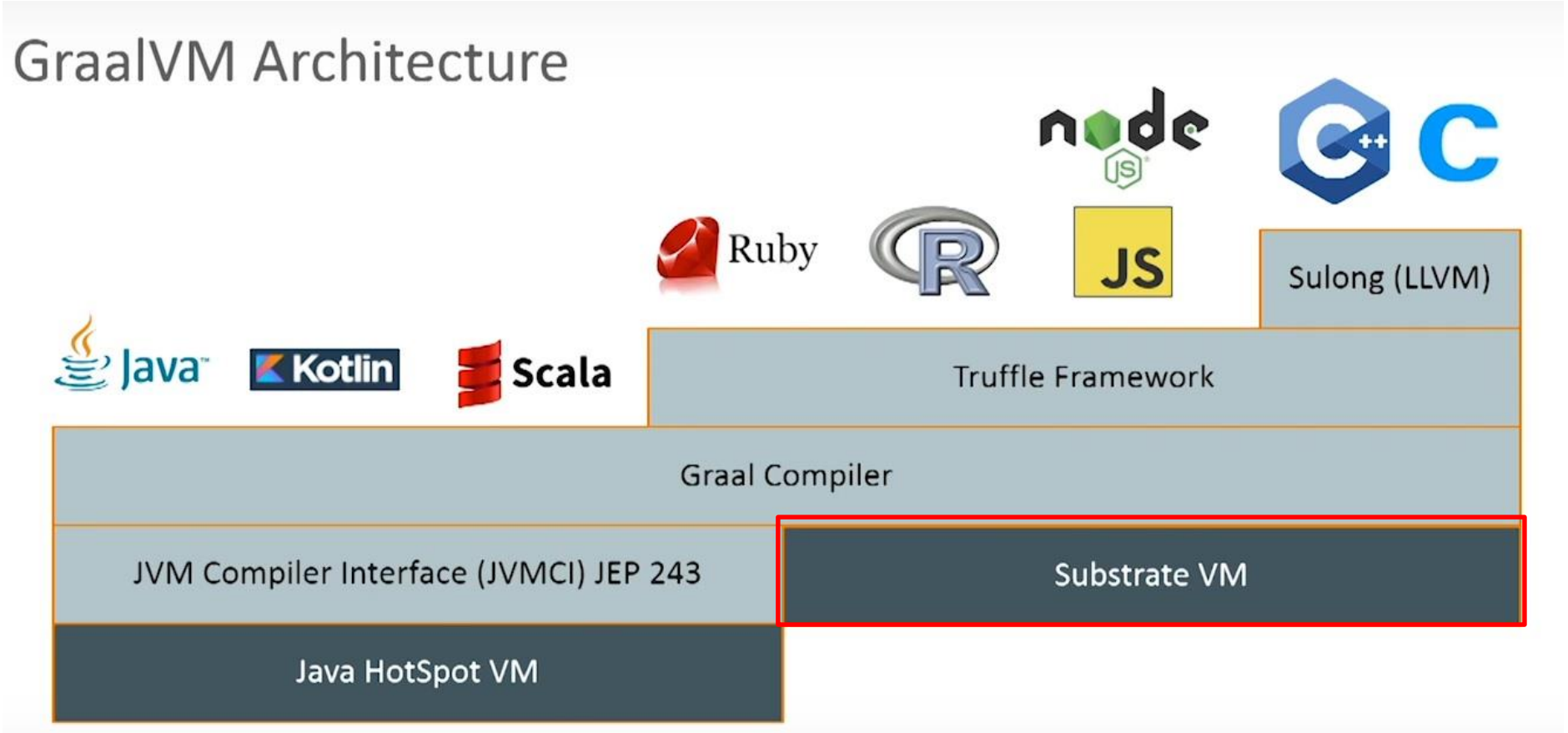
- Faster performance and smaller footprint
- Enhanced security features
- Managed capabilities for native code
- Premier 24x7x365 support via MOS

GraalVM Architecture



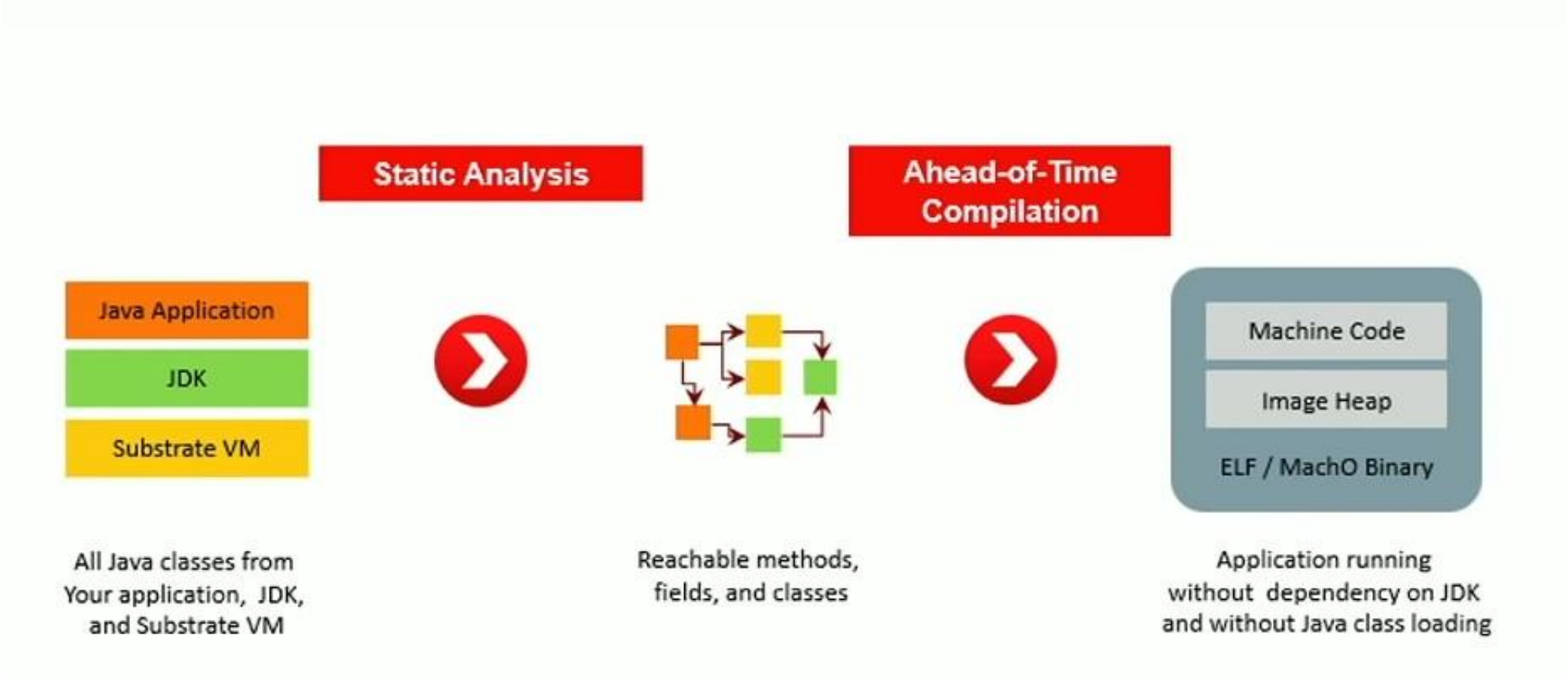
Sources: Practical Partial Evaluation for High-Performance Dynamic Language Runtimes http://chrisseaton.com/rubytruffle/pldi17-truffle/pldi17-truffle.pdf
„The LLVM Compiler Infrastructure“ https://llvm.org/

GraalVM Architecture



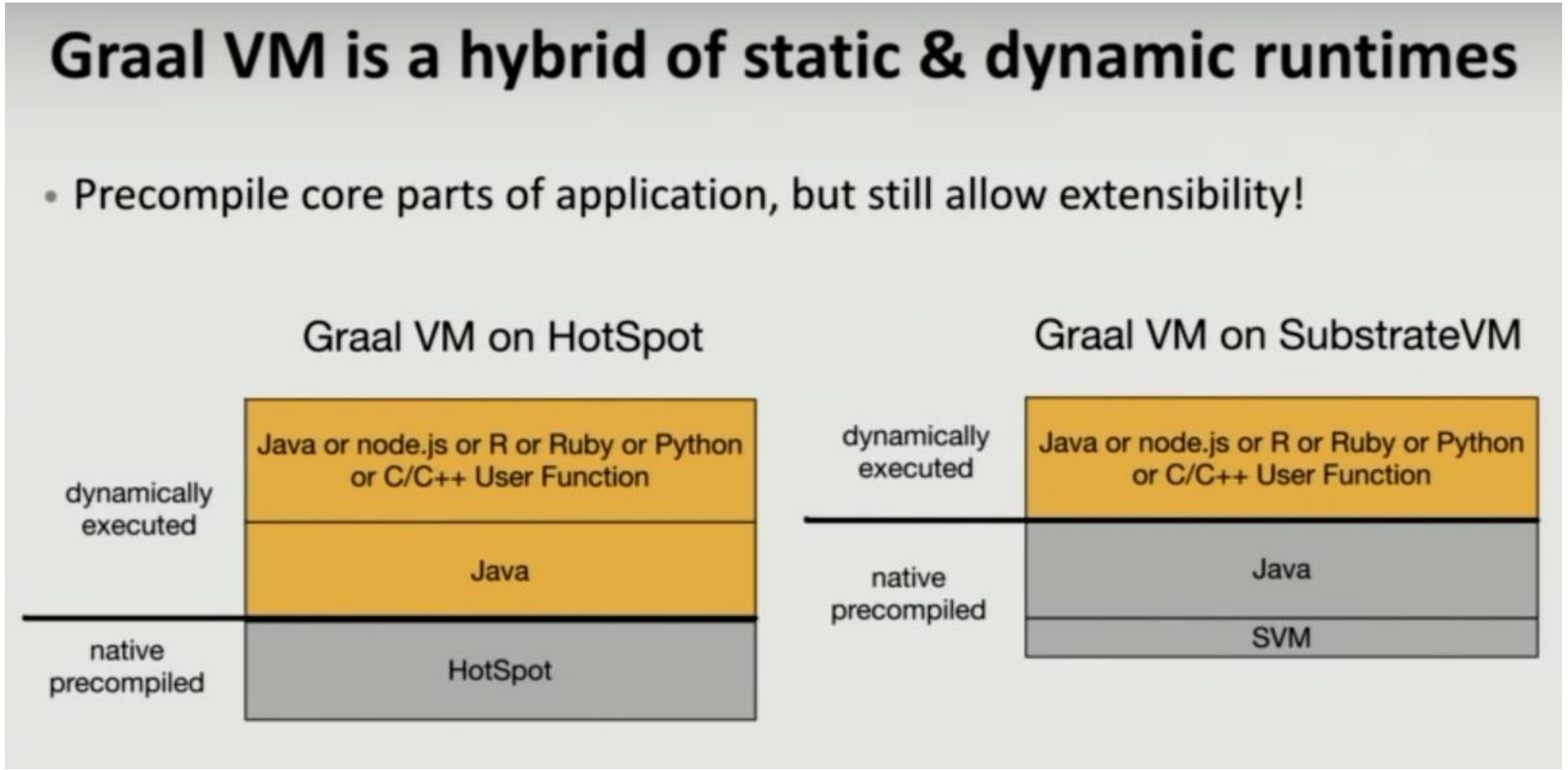
Sources: Practical Partial Evaluation for High-Performance Dynamic Language Runtimes http://chrisseaton.com/rubytruffle/pldi17-truffle/pldi17-truffle.pdf
„The LLVM Compiler Infrastructure“ https://llvm.org/

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 and SubstrateVM



Source: Oleg Selajev, Oracle : “Run Code in Any Language Anywhere with GraalVM” https://www.youtube.com/watch?v=JoDOo4FyYMU

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.

And both memory and execution time are cost dimensions, when using Serverless in the cloud

GraalVM on SubstrateVM
A game changer for Java & Serverless?

Current challenges with **native executable** using **GraalVM :**

- Most Cloud Providers (AWS) doesn’t provide GraalVM as Java Runtime out of the box, only Open JDK (e.g. AWS provides Corretto)

- Some Cloud Providers (e.g. AWS) provide Custom Runtime Option

Lambda Layers & Lambda Runtime API

New for AWS Lambda – Use Any Programming Language and Share Common Components

by Danilo Poccia | on 29 NOV 2018 | in AWS Lambda, Compute, Launch, News, Serverless, Top Posts* | Permalink | Comments | Share

Voiced by Amazon Polly

I remember the excitement when AWS Lambda was announced in 2014! Four years on, customers are using Lambda functions for many different use cases. For example, iRobot is using AWS Lambda to provide compute services for their Roomba robotic vacuum cleaners, Fannie Mae to run Monte Carlo simulations for millions of mortgages, Bustle to serve billions of requests for their digital content.

Today, we are introducing two new features that are going to make serverless development even easier:

- **Lambda Layers**, a way to centrally manage code and data that is shared across multiple functions.
- **Lambda Runtime API**, a simple interface to use any programming language, or a specific language version, for developing your functions.

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

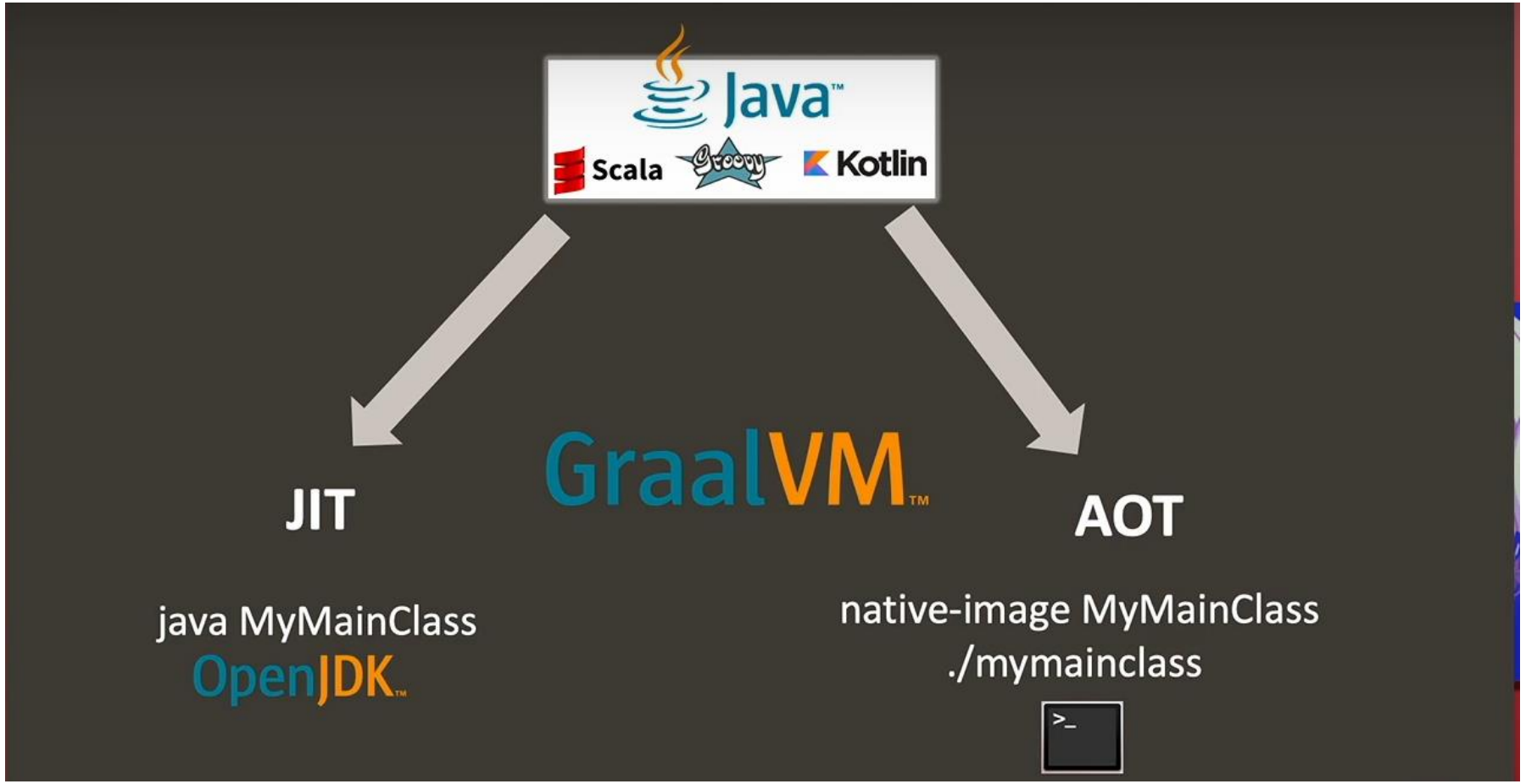
To use a custom runtime, set your function's runtime to `provided`. The runtime can be included in your function's deployment package, or in a layer.

Example function.zip

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

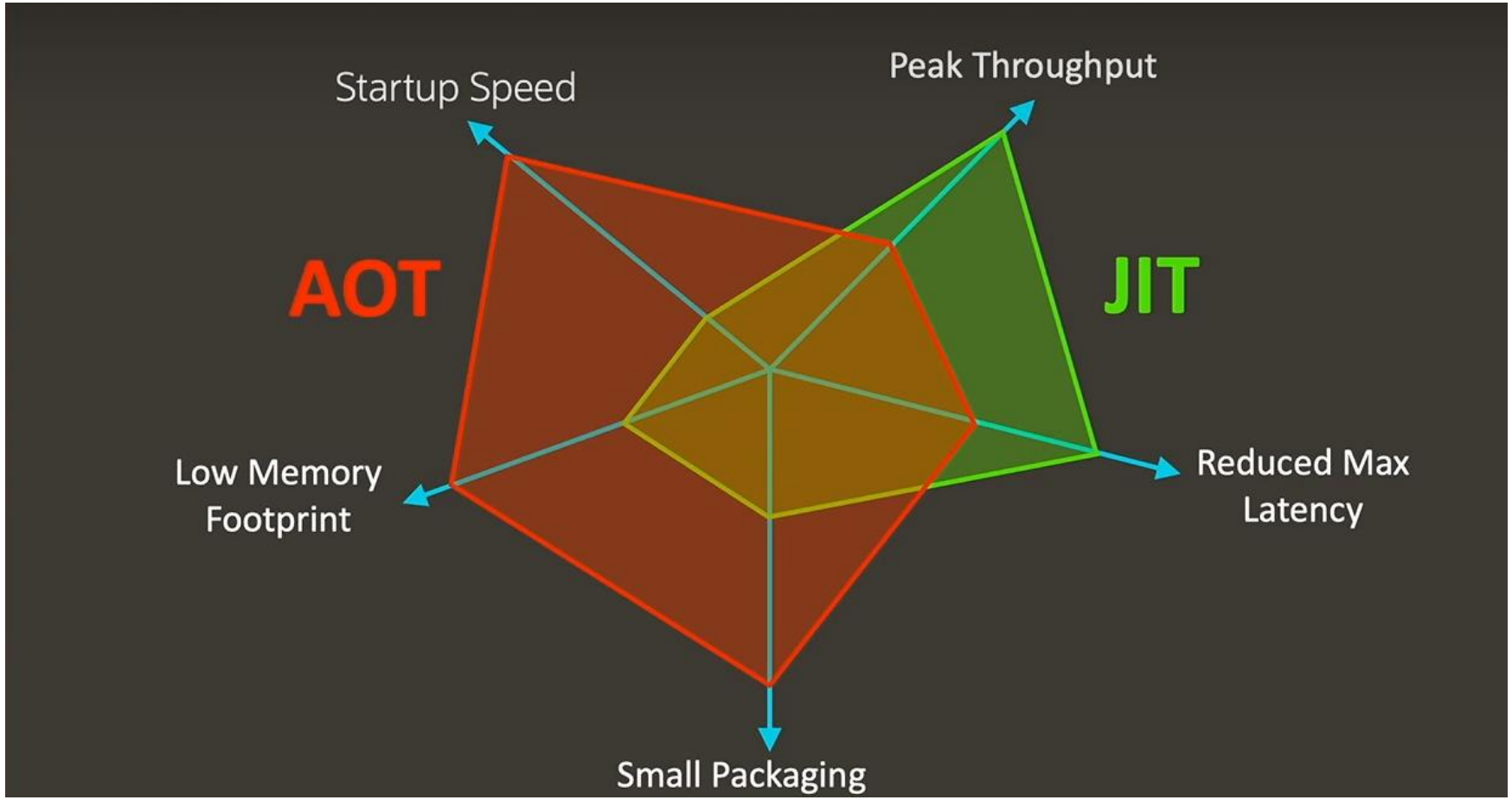
If there's a file named `bootstrap` in your deployment package, Lambda executes that file. If not, Lambda looks for a runtime in the function's layers. If the bootstrap file isn't found or isn't executable, your function returns an error upon invocation.

GraalVM Complitation Modes



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

AOT vs JIT



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

GraalVM Profile-Guided Optimizations

Profile-Guided Optimizations

GraalVM Enterprise allows to apply profile-guided optimizations (PGO) for additional performance gain and higher throughput of native images. With PGO you can collect the profiling data in advance and then feed it to the `native-image` builder, which will use this information to optimize the performance of the resulting binary.

Note: This feature is available with **GraalVM Enterprise** only.

One approach is to gather the execution profiles at one run and then use them to optimize subsequent compilation(s). In other words, you create a native image with the `--pgo-instrument` option to collect the profile information. The `--pgo-instrument` builds an instrumented native image with profile-guided optimization data collected of AOT compiled code in the *default.iprof* file, if nothing else is specified. Then you run an example program, saving the result in *default.iprof*. Finally, you create a second native image with `--pgo profile.iprof` flag that should be significantly faster. You can collect multiple profile files and add them to the image build.

Source: „https://www.graalvm.org/reference-manual/native-image/PGO"

Support of GraalVM native images in Frameworks

Spring Framework: working toward GraalVM native image support without requiring additional configuration or workaround is one of the themes of upcoming Spring Framework 5.3

Spring Boot: Ongoing work on experimental Spring Graal Native project. Probably ready for the 2.4 release

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.

Source: „GraalVM native image support“ https://github.com/spring-projects/spring-framework/wiki/GraalVM-native-image-support

Steps to deploy to AWS

- Installation prerequisites
    - Framework of your choice (Micronaut, Quarkus, Spring)
    - Java 8 or 11
    - Apache Maven or Gradle
    - AWS CLI and AWS SAM CLI (for local testing)
- Build Linux executable of your application with GraalVM native-image
- Deploy Linux executable as AWS Lambda Custom Runtime
    - Function.zip with bootstrap Linux executable

Example function.zip

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

Source: https://github.com/awslabs/aws-serverless-java-container/tree/master/samples/micronaut/pet-store

AWS Lambda Deployment of Custom Runtime with SAM

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: AWS Serverless Micronaut API - graal.spring.demo::graal-spring-demo
Globals:
  Api:
    EndpointConfiguration: REGIONAL
Resources:
  GraalVMApiService:
    Type: AWS::Serverless::Function
    Properties:
      Handler: not.used.in.provided.runtime
      Runtime: provided
      CodeUri: native-image/function.zip
      MemorySize: 512
      Policies: AWSLambdaBasicExecutionRole
      Tracing: Active
      Timeout: 15
      Events:
        GetResource:
          Type: Api
          Properties:
            Path: /{proxy+}
            Method: any
```

Source: https://github.com/awslabs/aws-serverless-java-container/tree/master/samples/micronaut/pet-store

ip.labs

Micronaut Framework

Source: https://micronaut.io/

ip.labs

AWS Lambda with Micronaut Framework

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

import javax.inject.Inject;

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

import io.micronaut.function.FunctionBean;

@FunctionBean("monthly-invoice-generator")
public class MonthlyInvoiceGeneratorFunction
implements Function<MonthlyInvoiceRequest, MonthlyInvoiceResponse> {

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

    @Inject
    private MonthlyInvoiceGeneratorService monthlyInvoiceGeneratorService;


    @Override
    public MonthlyInvoiceResponse apply(MonthlyInvoiceRequest monthlyInvoiceRequest) {

        if (LOG.isDebugEnabled()) {
            LOG.debug("request: {}", monthlyInvoiceRequest);
        }
        return this.monthlyInvoiceGeneratorService.generateInvoice(monthlyInvoiceRequest);
    }

}
```

Testing AWS Lambda with Micronaut Framework

```java
import io.micronaut.function.client.FunctionClient;

@FunctionClient
public interface MonthlyInvoiceGeneratorClient {

    @Named("monthly-invoice-generator")
    Single<MonthlyInvoiceResponse> apply(@Body MonthlyInvoiceRequest request);

}
```

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import javax.inject.Inject;

import org.junit.jupiter.api.Test;

import io.micronaut.test.annotation.MicronautTest;

@MicronautTest
public class MonthlyInvoiceGeneratorFunctionTest {

    @Inject
    MonthlyInvoiceGeneratorClient client;

    @Test
    public void testMonthlyInvoiceFunction() throws Exception {
        MonthlyInvoiceRequest monthlyInvoiceRequest = new MonthlyInvoiceRequest(9,2012, 123556);
        assertEquals(50.5, client.apply(monthlyInvoiceRequest).blockingGet().getInvoiceAmount());
    }
}
```

Micronaut Additional Features

- Custom Validators
- AWS API Gateway integration
- Spring annotation processor available

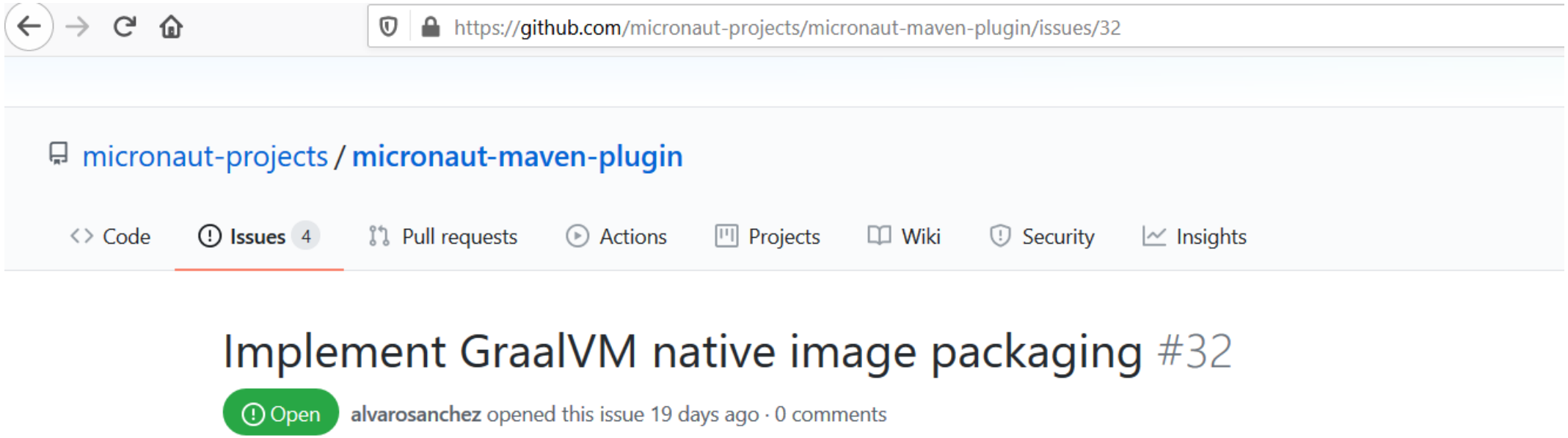
```
dependencies {
    annotationProcessor platform("io.micronaut:micronaut-bom:$micronautVersion")
    annotationProcessor "io.micronaut:micronaut-inject-java"
    annotationProcessor "io.micronaut:micronaut-validation"
    annotationProcessor "io.micronaut:micronaut-graal"
    implementation platform("io.micronaut:micronaut-bom:$micronautVersion")
    implementation "io.micronaut:micronaut-inject"
    implementation "io.micronaut:micronaut-validation"
    implementation "io.micronaut:micronaut-runtime"
    implementation "io.micronaut:micronaut-function-aws"
```

```
dependencies {
    annotationProcessor platform("io.micronaut:micronaut-bom:$micronautVersion")
    annotationProcessor "io.micronaut:micronaut-graal"
    annotationProcessor "io.micronaut:micronaut-inject-java"
    annotationProcessor "io.micronaut:micronaut-validation"
    annotationProcessor "io.micronaut.spring:micronaut-spring-boot"
    annotationProcessor "io.micronaut.spring:micronaut-spring-boot-annotation"
    annotationProcessor "io.micronaut.spring:micronaut-spring-web-annotation"
    implementation platform("io.micronaut:micronaut-bom:$micronautVersion")
    implementation "io.micronaut:micronaut-http-client"
    implementation "io.micronaut:micronaut-inject"
    implementation "io.micronaut:micronaut-validation"
    implementation "io.micronaut:micronaut-runtime"
```

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8

Build GraalVM Native Image with Micronaut Framework

```
/usr/lib/graalvm/bin/native-image -H:+TraceClassInitialization --initialize-at-build-time=reactor.core.publisher.Mono
--initialize-at-build-time=reactor.core.publisher.Flux --no-fallback --no-server -cp myapplication.jar
```



Quarkus

Source: https://quarkus.io/

AWS Lambda with Quarkus Framework

```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);
    }

}
```

Testing AWS Lambda with Quarkus Framework

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

import io.quarkus.amazon.lambda.test.LambdaClient;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class MonthlyInvoiceGeneratorFunctionTest {

    @Test
    public void testMonthlyInvoiceFunction() throws Exception {
        MonthlyInvoiceRequest monthlyInvoiceRequest = new MonthlyInvoiceRequest(9,2012, 123556);
        MonthlyInvoiceResponse monthlyInvoiceResponse =
                LambdaClient.invoke(MonthlyInvoiceResponse.class, monthlyInvoiceRequest);
        assertEquals(50.5, monthlyInvoiceResponse.getInvoiceAmount());
    }
}
```

Build GraalVM Native Image with Quarkus Framework

mvn –Pnative package

```
<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>
```

Quarkus Additional Features

- Website for creating the App
- AWS API Gateway integration
- Funqy

Source: Stefano Buliani : "Best practices for AWS Lambda and Java„ https://www.youtube.com/watch?v=ddg1u5HLwg8

Quarkus Fanqy

QUARKUS - FUNQY

Quarkus Funqy is part of Quarkus's serverless strategy and aims to provide a portable Java API to write functions deployable to various FaaS environments like AWS Lambda, Azure Functions, Knative, and Knative Events (Cloud Events). It is also usable as a standalone service.

Because Funqy is an abstraction that spans multiple different cloud/function providers and protocols it has to be a very simple API and thus, might not have all the features you are used to in other remoting abstractions. A nice side effect though is that Funqy is as optimized and as small as possible. This means that because Funqy sacrifices a little bit on flexibility, you'll get a framework that has little to no overhead.

Source: https://quarkus.io/guides/funqy

Quarkus-Fanqy AWS Serverless Support

- AWS Lambda
- AWS API Gateway

Spring (Boot) Framework

Source: https://spring.io/

ip.labs

Spring GraalVM Native Project

This project goal is to incubate support for building Spring Boot applications as GraalVM native images.

Watch The Path Towards Spring Boot Native Applications SpringOne 2020 talk recording for more details:

It is mainly composed of:

- `spring-graalvm-native-feature` : this module is a GraalVM feature. It is a kind of plugin for the native-image compilation process (which creates the native-image from the built class files). The feature participates in the compilation lifecycle, being invoked at different compilation stages to offer extra information about the application to aid in the image construction.

AWS Lambda with Spring Framework using Spring Graal Native and Spring Cloud Functions

```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);
    }
}
```

Bean Registration with Spring Framework using Spring Graal Native and Spring Cloud Functions

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.function.context.FunctionRegistration;
import org.springframework.cloud.function.context.FunctionType;
import org.springframework.cloud.function.context.FunctionalSpringApplication;
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.support.GenericApplicationContext;

@SpringBootApplication(proxyBeanMethods = false)
public class DemoApplication implements ApplicationContextInitializer<GenericApplicationContext> {

    public static void main(String[] args) {
        FunctionalSpringApplication.run(DemoApplication.class, args);
    }

    @Override
    public void initialize(GenericApplicationContext context) {

        context.registerBean("monthlyInvoiceGenerator", FunctionRegistration.class,
                () ->  new FunctionRegistration<> (new MonthlyInvoiceGeneratorFunction()).
                type(FunctionType.from(MonthlyInvoiceRequest.class).
                to(MonthlyInvoiceResponse.class)));
    }

}
```

Build GraalVM Native Image with Spring Framework

mvn –Pnative package

```
<profile>
    <id>native</id>
    <build>
        <plugins>
        <plugin>
            <groupId>org.graalvm.nativeimage</groupId>
            <artifactId>native-image-maven-plugin</artifactId>
            <version>20.2.0</version>
            <configuration>
                <mainClass>com.example.demo.DemoApplication</mainClass>
                <buildArgs>-J-Xmx4G -H:+TraceClassInitialization -H:+ReportExceptionStackTraces
                -Dspring.native.remove-unused-autoconfig=true -Dspring.native.remove-yaml-support=true
                </buildArgs>
                <additionalBuildArgs>--delay-class-initialization-to-runtime=sun.nio.ch.WindowsAsynchronousFileChannelImpl
                </additionalBuildArgs>
                <imageName>${project.artifactId}</imageName>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>native-image</goal>
                    </goals>
                    <phase>package</phase>
                </execution>
            </executions>
        </plugin>
        </plugins>
    </build>
</profile>
```

Framework Comparison

- Project Initializer
- Programming Model
- Database Support
- Test Support
- Native Image
  - Native image size
  - Startup time
  - Heap size

Source: „Battle Of The Microservice Frameworks: Micronaut versus Quarkus edition! by Michel Schudel“ https://www.youtube.com/watch?v=hnEXOqcNXPs
„Micronaut 2.0 vs Quarkus 1.3.1 vs Spring Boot 2.3 Performance on JDK 14“ https://www.youtube.com/watch?v=rJFgdFIs_k8

Conclusion

- GraalVM and Frameworks are really powerful with a lot of potential
- But in combination with Native Image currently not without challenges
  - AWS Lambda Custom runtime requires Linux executable only
  - Windows and Mac developers may only build Linux executable via Docker
  - plenty of *'No instances of … are allowed in the image heap'* and other errors when building a native image
  - Lots of experimentation with additional build arguments like “initialize-at-runtime” or “delay-class-initialization-to-runtime“ required
- Once again: Less (dependencies, classes) is more
  - AWS Lambda function should be small and shouldn’t have many dependencies

Source: „https://stackoverflow.com/questions/63328298/how-do-you-debug-a-no-instances-of-are-allowed-in-the-image-heap-when-buil

Try it yourselves

- Micronaut
  - https://github.com/micronaut-guides/micronaut-function-aws-lambda
- Quarkus
  - https://github.com/JosemyDuarte/quarkus-terraform-lambda-demo/tree/dynamo-terraform
- Spring Boot
  - https://github.com/spring-projects-experimental/spring-graalvm-native/tree/master/spring-graalvm-native-samples/function-aws
- Misc
  - https://github.com/awslabs/aws-serverless-java-container/tree/master/samples

Project Leyden

Call for Discussion: New Project: Leyden

mark.reinhold at oracle.com mark.reinhold at oracle.com
Mon Apr 27 16:38:55 UTC 2020

- Previous message: Type-parameterized complement to Object.equals(Object)
- Next message: Call for Discussion: New Project: Leyden
- **Messages sorted by:** [date] [thread] [subject] [author]

```
I hereby invite discussion of a new Project, Leyden, whose primary goal
will be to address the long-term pain points of Java's slow startup time,
slow time to peak performance, and large footprint.

Leyden will address these pain points by introducing a concept of _static
images_ to the Java Platform, and to the JDK.

  - A static image is a standalone program, derived from an application,
    which runs that application -- and no other.

  - A static image is a closed world: It cannot load classes from outside
    the image, nor can it spin new bytecodes at run time.

These two constraints enable build-time analyses that can remove unused
classes and identify class initializers which can be run at build time,
thereby reducing both the size of the image and its startup time.  These
constraints also enable aggressive ahead-of-time compilation, thereby
reducing the image's time to peak performance.
```

Source: https://mail.openjdk.java.net/pipermail/discuss/2020-April/005429.html

Thank You!