AWS re:Invent

SVS403-R

Best practices for AWS Lambda and Java

Stefano Buliani
Principal BDM
Amazon Web Services

AWS re:Invent

aws

Agenda

- Should we go serverless with Java?
- Cold start challenges: A story of gradual improvements and a day of my life I'll never have back
- Key tips
- Where to next?

Should we go serverless with Java?

"Customers have been struggling with the cold start of Java functions in AWS Lambda. But there are smart people trying to make things better."

Tim Bray

Distinguished Engineer, AWS

@timbray

AWS re:Invent

aws

AWS Lambda **cold starts** for Java 8 functions **can be** slow

However

- Let’s not kid ourselves, **Java is here to stay**
  - 41% of Stack Overflow 2019 developer survey respondents work with Java regularly
  - 1st language in TIOBE index for 2019 with 16.661% rating

- We need to make it work well in AWS Lambda
- When I say “we,” I mean **all of us**. Let me tell you why…

Our story begins…

Like all good stories do

With a **high-severity ticket** waking me up at **3am**

Like all good stories do

- Production API launched with a Lambda/Java backend
- p99 over 24 seconds and p100 **over 30 seconds**
- API Gateway was timing out

First, a Band-aid

- Increasing the amount of **memory** allocated to the Lambda function helps: No more timeouts (256Mb -> 512Mb)
  - AWS Lambda allocates CPU cycles to a function based on the amount of memory configured
  - More memory = Higher CPU

- However, this also increases costs—not really what we want to happen
- Let’s **dive deeper**

Our objectives

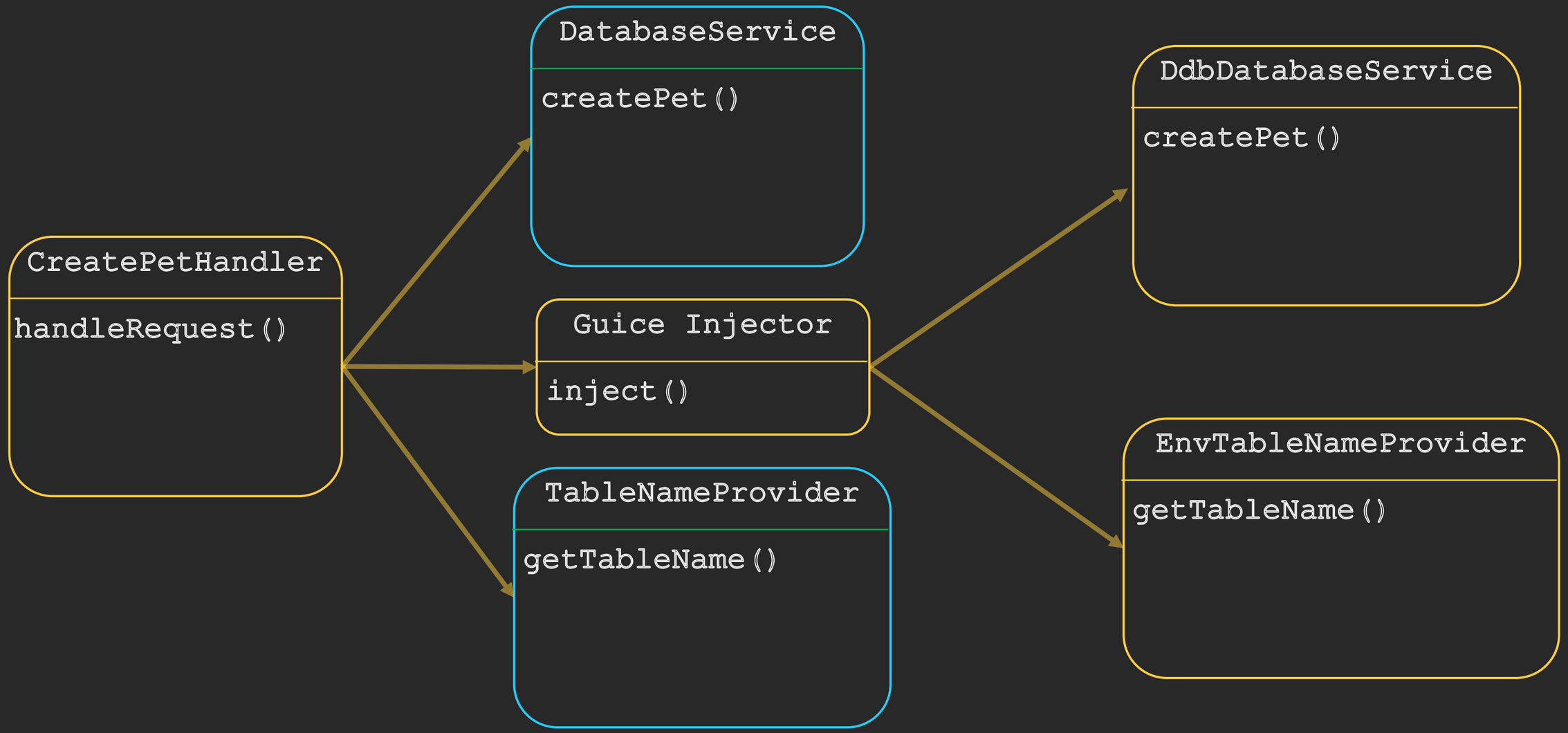
- We need to make sure that there are **no timeouts** (duration under 30 seconds)

- Getting below **10 seconds** is a bonus

- What's the right, **long-term solution**?

Diving deeper

First, let's try to replicate the issue



Establish a performance baseline

Gatling: `constantUsersPerSec(50) over(60 seconds)`

STATISTICS

Executions

	Total	OK	KO
	848	848	0
Mean req/s	4.685	4.685	-

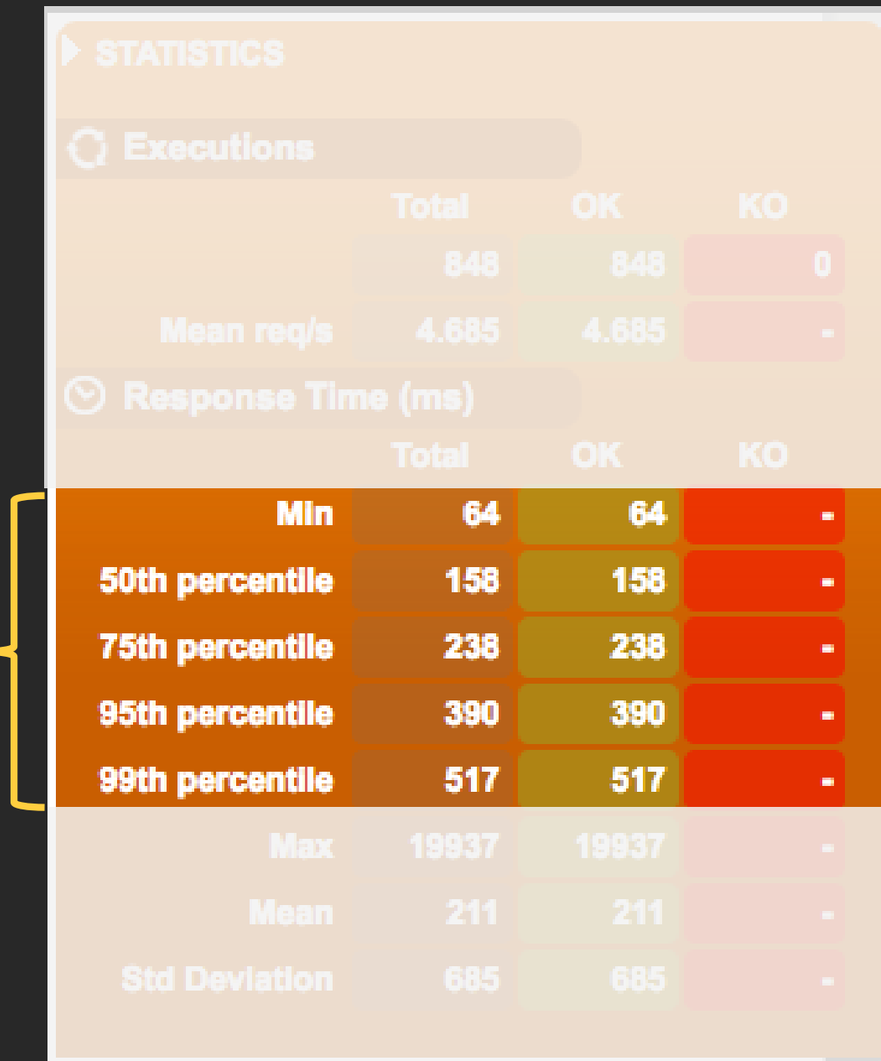
Response Time (ms)

	Total	OK	KO
Min	64	64	-
50th percentile	158	158	-
75th percentile	238	238	-
95th percentile	390	390	-
99th percentile	517	517	-
Max	19937	19937	-
Mean	211	211	-
Std Deviation	685	685	-

Good news first

Gatling: `constantUsersPerSec(50) over(60 seconds)`

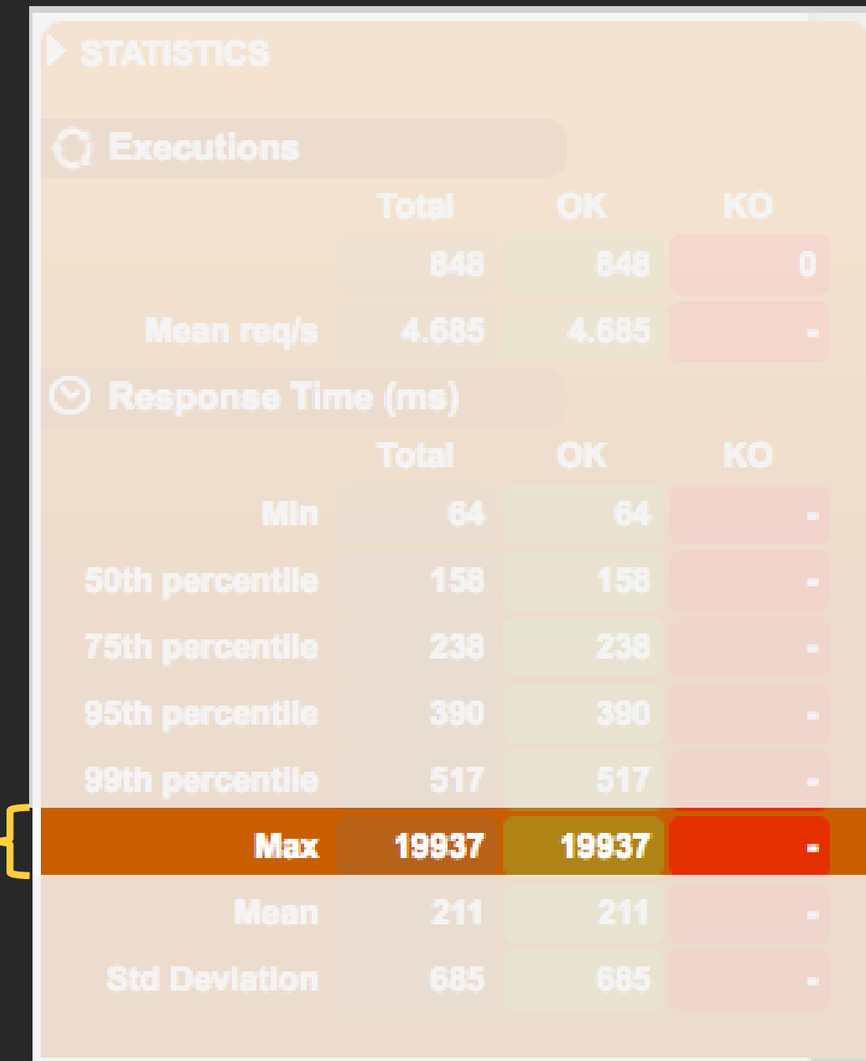
- The response time between min and p99 is **consistent**
- The standard deviation, excluding the cold start, is only **σ101**



Bad news next

Gatling: `constantUsersPerSec(50) over(60 seconds)`

- We had only one cold start, but it was **extremely cold!**



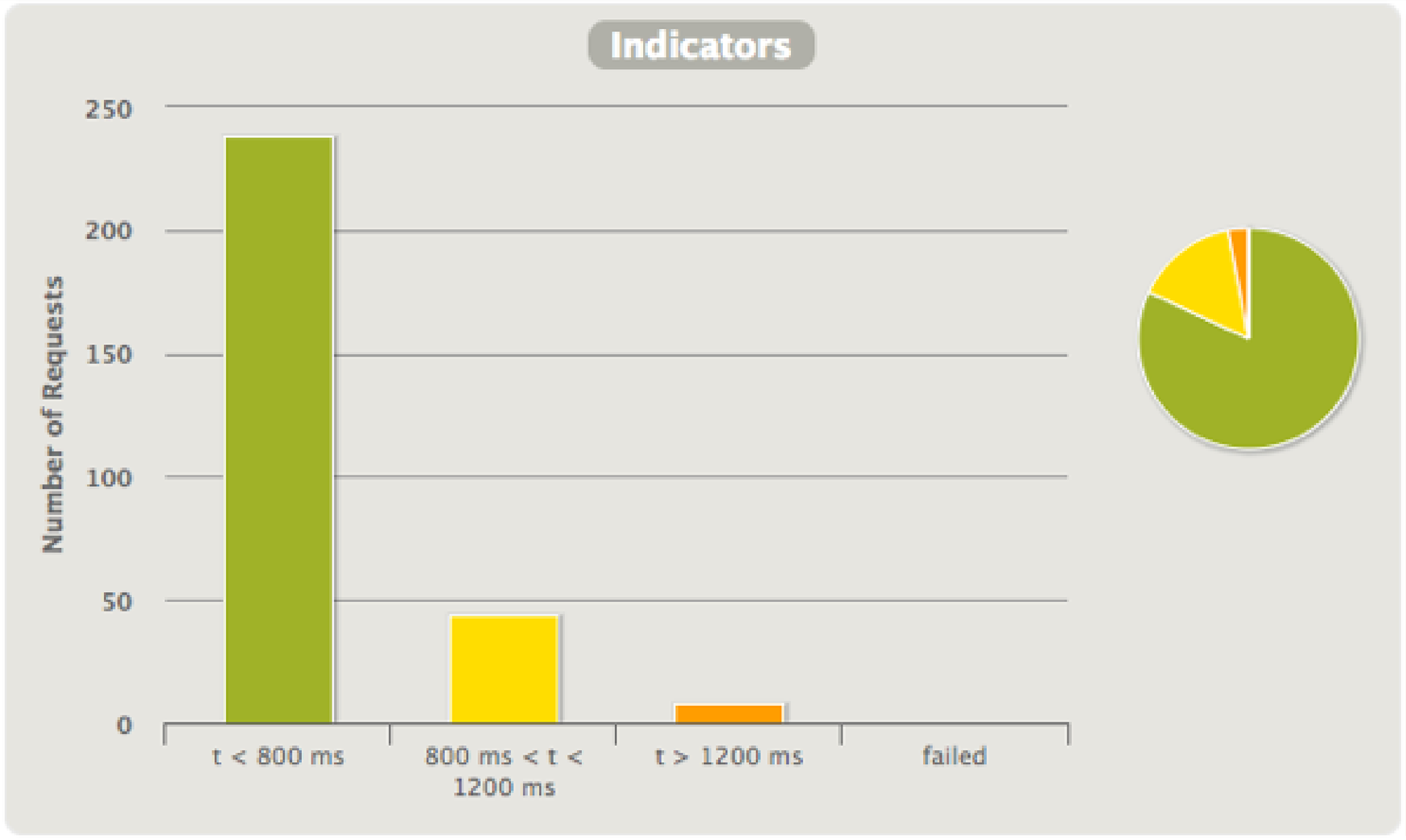
Where do we go from here?

Instrumenting our code to gather more data

```
Subsegment injectorSegment = AWSXRay.beginSubsegment("CreateInjector");
try {
    injector = Guice.createInjector(new PetsModule());
} catch (Exception e) {
    injectorSegment.addException(e);
    return null;
} finally {
    AWSXRay.endSubsegment();
}
```

And run our load test again

```
constantUsersPerSec(50) over(60 seconds)
```



STATISTICS

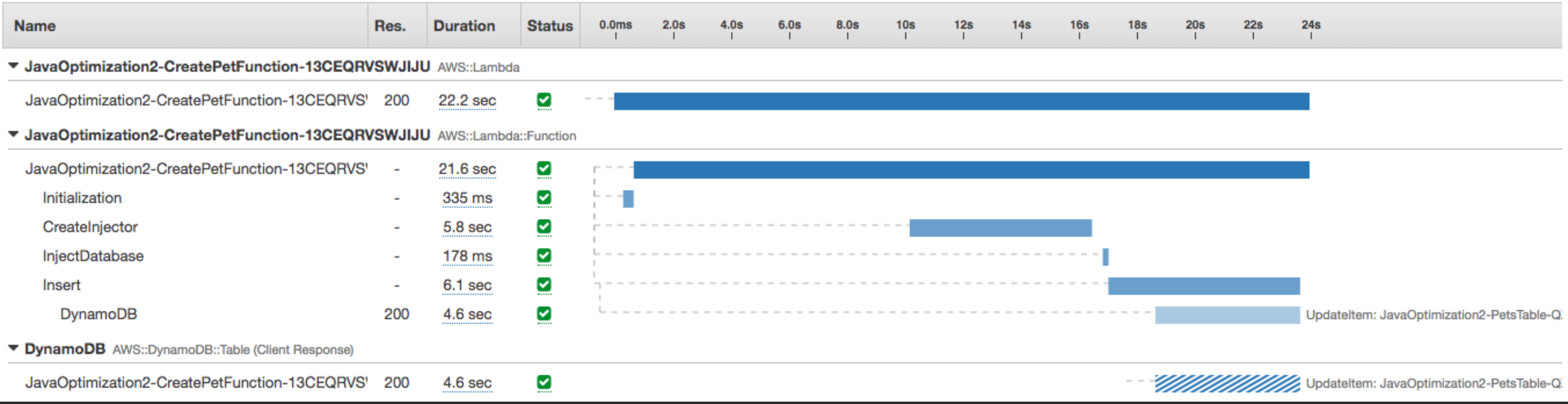
Executions

	Total	OK	KO
	290	290	0
Mean req/s	1.602	1.602	-

Response Time (ms)

	Total	OK	KO
Min	190	190	-
50th percentile	478	478	-
75th percentile	714	714	-
95th percentile	1059	1059	-
99th percentile	1308	1308	-
Max	22387	22387	-
Mean	620	620	-
Std Deviation	1308	1308	-

What can AWS X-Ray tell us?

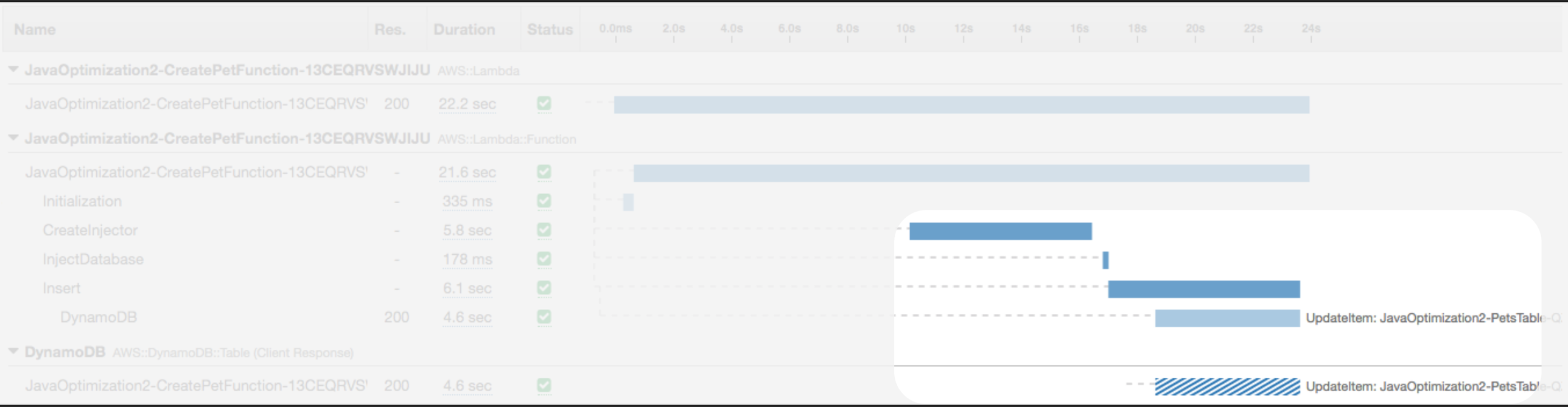


Notice anything?

What can X-Ray tell us?

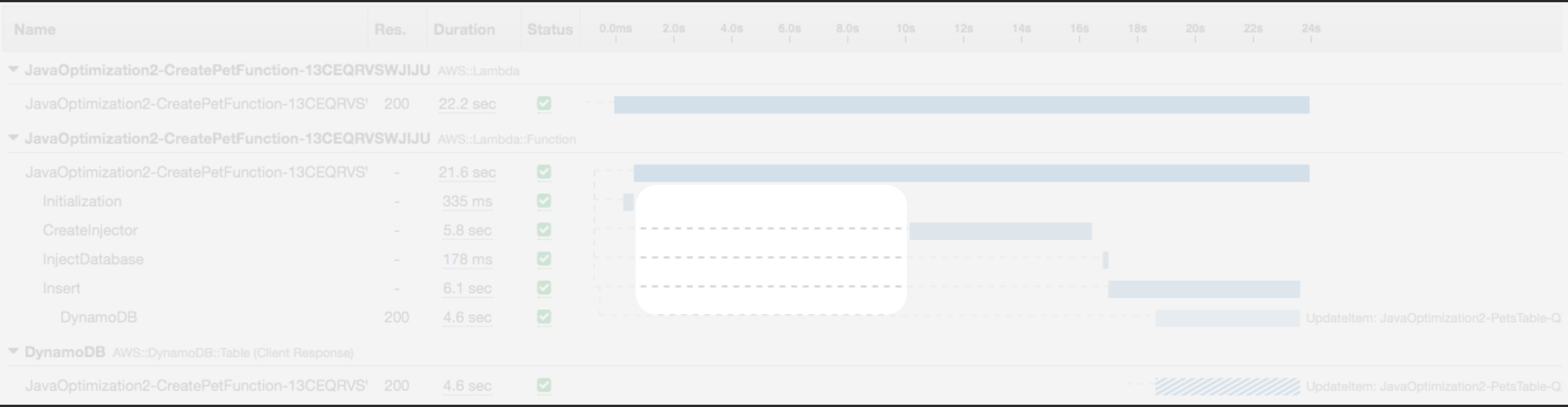
Initialization tracks the time AWS Lambda took to **start the runtime (Java virtual machine [JVM])**

What can X-Ray tell us?



I thought I'd spend my time in the **handler segments**

What can X-Ray tell us?



Instead, I find myself with a **7-second gap** I cannot explain

Is the JVM doing ok?

Let’s start with a **hello world**, **single-class** function

Is the JVM doing ok?

Let’s start with a **hello world**, **single-class** function

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
"Hello, World!"
```

Summary

Code SHA-256

pVNsa4s6BV4ZzJ9vH9oJugqWJ796DGzq4rP2NPQj/tM=

Request ID

75869b6f-b535-11e8-bddd-ad99505603f8

Duration

87.40 ms

Billed duration

100 ms

Resources configured

256 MB

Max memory used

40 MB

I’d say it is—**87.4ms** with an **11Kb** deployment package

Another simple change

Include the **AWS SDK for Java**: We just initialize the Amazon DynamoDB client

Another simple change

Include the **AWS SDK for Java**: We just initialize the DynamoDB client

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
"Hello, World!"
```

Summary

Code SHA-256
YYw+pyIn/lYIzbWrGbFpzaBIroFrJxcyXMGwoeh+EoU=

Request ID
047afb32-b536-11e8-afa5-57a385427a73

Duration
6372.41 ms

Billed duration
6400 ms

Resources configured
256 MB

Max memory used
62 MB

6.4 seconds and deployment package is now **7.6Mb**

My hypothesis

We are loading too many classes

My hypothesis

We are loading too many classes

- Impact of package size is not significant on cold starts
- The **JVM lazily loads classes** as they are needed—lots of I/O operations

We may be onto something

How many classes are we loading?

```
$ java -cp my.jar -verbose:class Handler | grep Loaded | wc -l
```

- Create a **main** method and start the function in local
- Verbose **logging of class loading**

We may be onto something

How many classes are we loading?

```
$ java -cp my.jar -verbose:class Handler | grep Loaded | wc -l
```

4,130

Let’s be serious—who’s doing this to me?

```
$ java -cp my.jar -verbose:class Handler | grep Loaded | clever sed | sort *
```

```
...
143 com.fasterxml.jackson
219 org.apache.http
373 com.google
507 com.amazonaws
```

* java -cp my.jar -verbose:class Handler | grep '\[Loaded' | grep '.jar\]' | sed 's/\[Loaded \([^A-Z]*\)[\$A-Za-z0-9]* from .*\]/\1/g' | sort | uniq -c | sort

Let’s be serious—who’s doing this to me?

```
$ java -cp my.jar -verbose:class Handler | grep Loaded | clever sed | sort *
```

```
...
143 com.fasterxml.jackson
219 org.apache.http
373 com.google
507 com.amazonaws
```

- Looks like the **AWS SDK** is the main offender
- Both **Jackson** and **Apache’s HTTP client** are dependencies of the SDK
- Next is **Guice**

* java -cp my.jar -verbose:class Handler | grep '\[Loaded' | grep '.jar\]' | sed 's/\[Loaded \([^A-Z]*\)[\$A-Za-z0-9]* from .*\]/\1/g' | sort | uniq -c | sort

Intermission: Understanding cold starts

The initialization step

1. AWS Lambda **starts a JVM**
2. Java runtime **loads and initializes** handler class
3. Lambda calls the **handler method**

The initialization step

1. AWS Lambda **starts a JVM**
2. Java runtime **loads and initializes** handler class
3. Lambda calls the **handler method**

Boosted host CPU access

(up to 10 seconds)

The initialization step

1. AWS Lambda **starts a JVM**
2. Java runtime **loads and initializes** handler class
3. Lambda calls the **handler method**

Throttled CPU access

Back to our story

Switch to the AWS SDK for Java 2.0

- Smaller footprint and more **modular**
- Allows us to **pick the HTTP client** we want to use
  - In our case, we are well served by Java's built-in **URLConnection**

Switch to the AWS SDK for Java 2.0

6.4 seconds to 4.7 seconds: **26% improvement**

Package size is slightly larger: **8.9Mb**

We can do better

Excluding Apache and Netty clients via Maven, package size: **4.7Mb**

4.7 seconds to 4 seconds: Another **15%** improvement

Let's make the same changes to our sample application

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>dynamodb</artifactId>
  <version>2.9.10</version>
  <exclusions>
    <exclusion>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>apache-client</artifactId>
    </exclusion>
    <exclusion>
      <groupId>software.amazon.awssdk</groupId>
      <artifactId>netty-nio-client</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>url-connection-client</artifactId>
  <version>2.0.10</version>
</dependency>
```

1. Include **AWS SDK for Java 2.0**

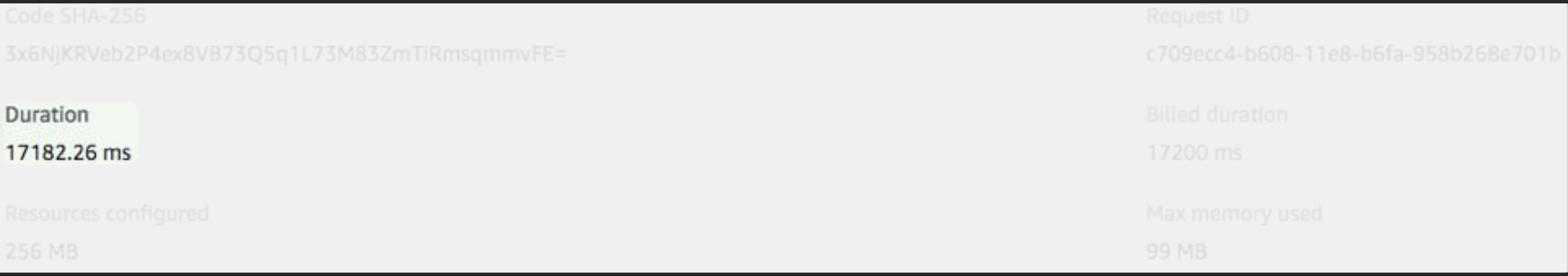
2. **Exclude** unnecessary deps

3. Include **URLConnection** client

Test our sample application again

Code SHA-256	Request ID
3x6NjKRVeb2P4ex8VB73Q5q1L73M83ZmTiRmsqmmvFE=	c709ecc4-b608-11e8-b6fa-958b268e701b
Duration	Billed duration
17182.26 ms	17200 ms
Resources configured	Max memory used
256 MB	99 MB

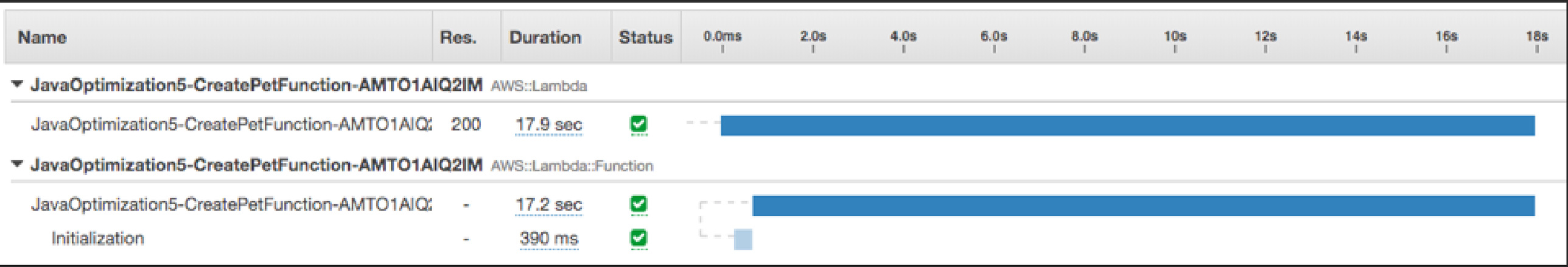
Test our sample application again



We are making progress

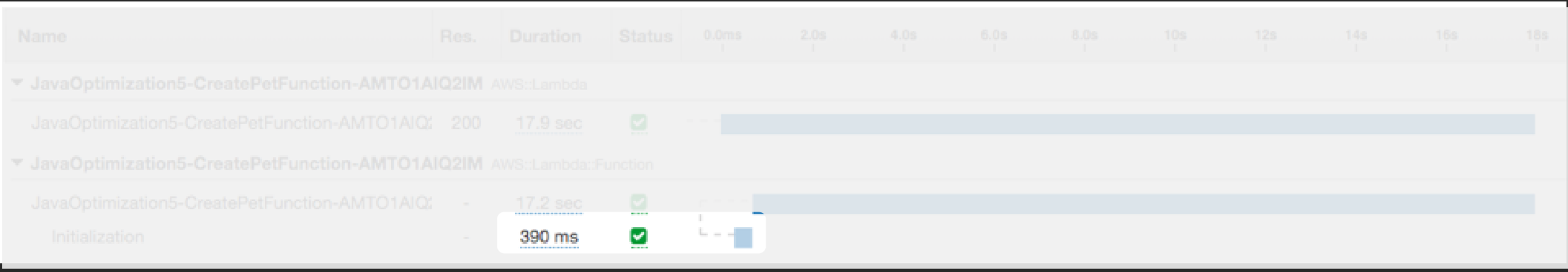
23.6 seconds to 17.2: **27% improvement**

Back to the X-Ray data



More importantly, the **gap** between initialization and handler call is gone!

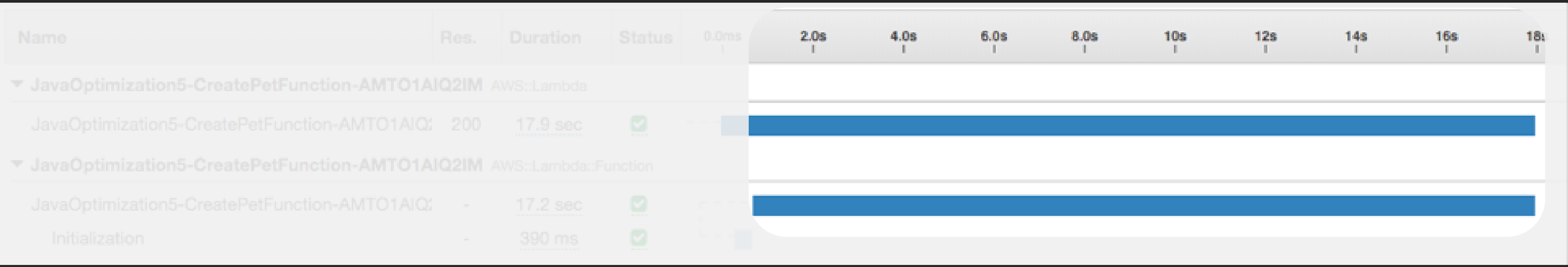
Back to the X-Ray data



Now that we have started making progress, let’s take advantage of the **boosted CPU access during the initialization**

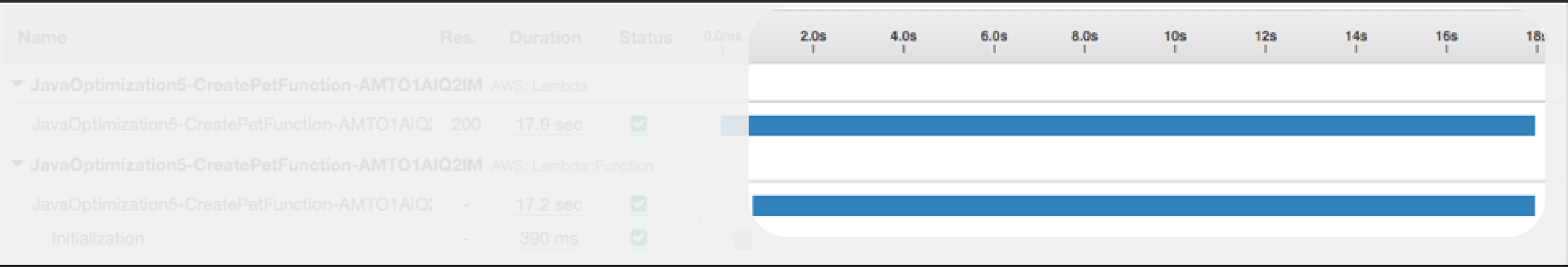
Make the most of the init time

Back to the X-Ray data



Now that all of the time is spent in the handler, we use **Guice** to inject the **DynamoDB client**

So what next?



Let’s trade some of this handler time for initialization time by **pre-configuring components**

Front-load all necessary classes

```
public class CreatePetHandler implements RequestHandler<Pet, Pet> {
    private static Injector injector = Guice.createInjector(new PetsModule());
    private static DatabaseService db = injector.getInstance(DatabaseService.class);

    @Override
    public Pet handleRequest(Pet input, Context context) {

        if (input.getName() == null && input.getBreed() == null) {
            System.out.println("Invalid pet data");
            return null;
        }

        Pet newPet = db.createPet(input);
            if (newPet == null) {
                System.out.println("Could not create pet in database");
            }
           return newPet;

    }
}
```

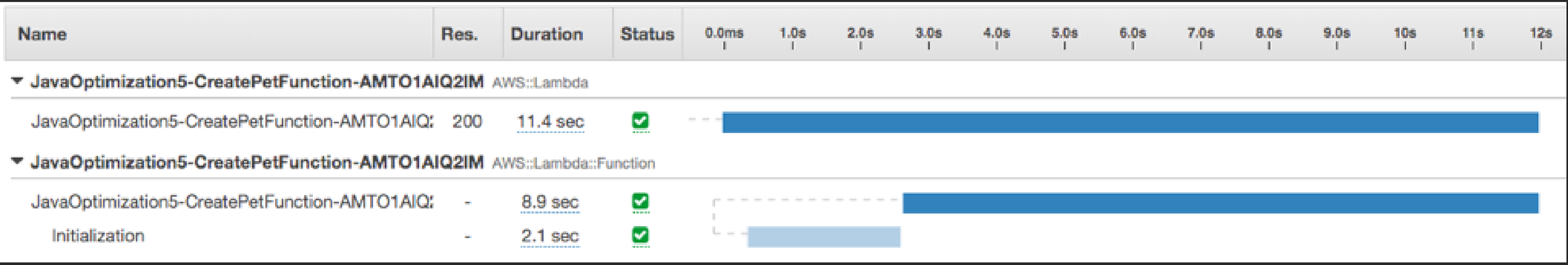
Use **static class members**

Set all configuration parameters in advance

```
SdkHttpClient httpClient = UrlConnectionHttpClient.builder().build();
client = DynamoDbClient.builder()
                       .httpClient(httpClient)
                       .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
                       .region(Region.US_WEST_2)
                       .endpointOverride(new URI( str: "https://dynamodb.us-west-2.amazonaws.com"))
                       .overrideConfiguration(ClientOverrideConfiguration.builder().build())
                       .build();
```

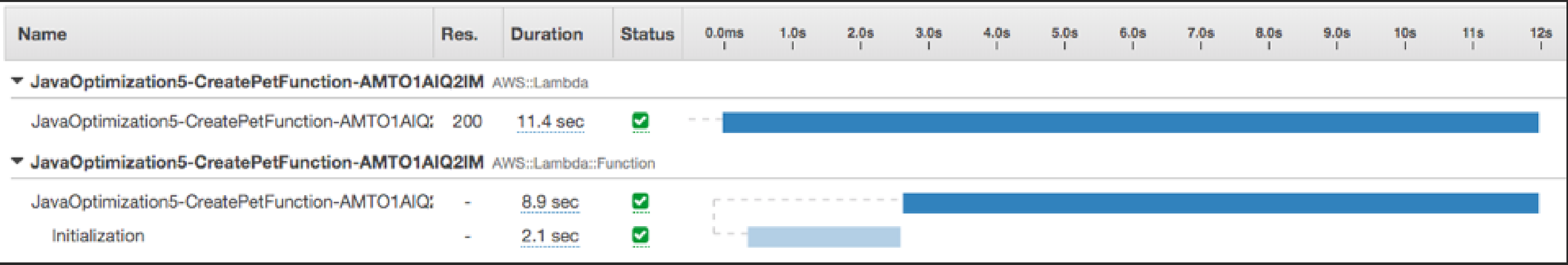
Discovering **endpoints**, **regions**, **clients**, and **credential providers** takes time; you should already know these values

Have we made a difference?



18 seconds to 12: **33% improvement**

Have we made a difference?



All **Guice** reflection occurs while we have access to more of the host CPU

We traded **8.3 seconds** of handler execution time for **1.71 seconds** of initialization time. **Steal of a deal**!

State of the union

With **three simple changes**, our cold start has gone from ~23s to ~12s, a **47% improvement**!

- Switch to the **AWS SDK for Java 2.0**
- **Front-load classes** during initialization
- Provide all known values to **avoid auto-discovery**

Intermission: Why static fields?

When the JVM loads a class

- **Preparation**
  - *Preparation* involves creating the static fields for a class or interface and initializing such fields to their default values (§2.3, §2.4). This does not require the execution of any Java Virtual Machine code; explicit initializers for static fields are executed as part of initialization (§5.5), not preparation.
- **Linking, classes, interfaces, fields, and call sites resolution as well as access control**
- **Initialization**
  - *Initialization* of a class or interface consists of executing its class or interface initialization method (§2.9).

The JVM initialization step

- Initialization blocks run in the same **order** in which they appear in the program

- Static initialization blocks run when a class is **loaded for the first time**

```
class TestInit {
  private static boolean isTrue;
  static {
    isTrue = true;
  }
}
```

- Instance initialization blocks are executed whenever the **class is initialized** and before constructors are invoked

```
class TestInit {
  private boolean isTrue;
  {
    isTrue = true;
  }
}
```

Who wants to see some bytecode?

With static initializers

```
public class StreamLambdaHandler {
  public StreamLambdaHandler();
    Code:
      0: aload_0
      1: invokespecial #1 // Method
java/lang/Object."<init>":()V
      4: return

static {};
    Code:
      0: iconst_1
      1: anewarray     #13 // class
java/lang/Class
      4: dup
      5: iconst_0
      6: ldc           #14 // class
PetStoreSpringAppConfig
      8: aastore
      9: invokestatic  #15 // Method
SpringLambdaContainerHandler.getAwsProxyHandler
:([Ljava/lang/Class;)Lcom/amazonaws/serverless/
proxy/spring/SpringLambdaContainerHandler;
       ...
```

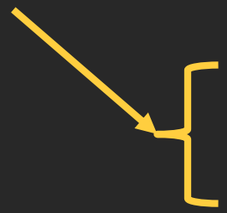
With constructor

```
public class StreamLambdaHandler {
  public StreamLambdaHandler();
    Code:
      0: aload_0
      1: invokespecial #1 // Method
java/lang/Object."<init>":()V
      4: invokestatic  #2 // Method
java/time/Instant.now:()
      7: invokevirtual #3 // Method
java/time/Instant.toEpochMilli:()J
      10: lstore_1
      11: getstatic     #4 // Field
java/lang/System.out:Ljava/io/PrintStream;
      14: new           #5 // class
java/lang/StringBuilder
      17: dup
      18: invokespecial #6 // Method
java/lang/StringBuilder."<init>":()V
      21: ldc           #7 // String startCall:
      ...
```

All this is to say what?

We start executing our code before:

1. Instance initializers
2. Constructors



```
public class StreamLambdaHandler {
  public StreamLambdaHandler();
    Code:
      0: aload_0
      1: invokespecial #1 // Method
java/lang/Object."<init>":()V
      4: invokestatic  #2 // Method
java/time/Instant.now:()
      7: invokevirtual #3 // Method
java/time/Instant.toEpochMilli:()J
      10: lstore_1
      11: getstatic     #4 // Field
java/lang/System.out:Ljava/io/PrintStream;
      14: new           #5 // class
java/lang/StringBuilder
      17: dup
      18: invokespecial #6 // Method
java/lang/StringBuilder."<init>":()V
      21: ldc           #7 // String startCall:
      ...
```

Also

Including the class constructor means the code does not fit on the slide

Back to our story

State of the union

With three simple changes, our cold start has gone from ~23s to ~12s, a 47% improvement!

- 47% improvement is great, but **12 seconds is not ideal**

Let’s profile the application

Where are we spending the time?

	Time (ms)
Guice.createInjector(java.lang.Iterable)	1,745
InjectorImpl.getInstance(java.lang.Class)	992
DefaultDynamoDbClient.putItem(PutItemRequest)	927

Let’s profile the application

Where are we spending the time?

	Time (ms)
Guice.createInjector(java.lang.Iterable)	1,745
InjectorImpl.getInstance(java.lang.Class)	992
DefaultDynamoDbClient.putItem(PutItemRequest)	927

Could be the **TLS handshake** for the first request; let’s ignore it for now

Let’s profile the application

Where are we spending the time?

	Time (ms)
Guice.createInjector(java.lang.Iterable)	1,745
InjectorImpl.getInstance(java.lang.Class)	992
DefaultDynamoDbClient.putItem(PutItemRequest)	927

Guice seems to be slow—especially when **discovering constructors**

My hypothesis

Reflection's what's done it

My hypothesis

Reflection's what's done it

- *Because reflection involves types that are dynamically resolved, certain Java virtual machine **optimizations cannot be performed**. Consequently, reflective operations have slower performance than their non-reflective counterparts and should be avoided in sections of code that are called frequently in performance-sensitive applications.*

- Further, classpath scanning in a **CPU and IO-constrained** environment is not a great idea.

Eliminating reflection

Switch from Guice to Dagger 2

Dagger 2 is Google's fork of Square's DI Dagger framework

- Generates injection code at **compile time**
- Uses the standard **@Inject** annotations
- No reflection FTW!

In practice this means

```
<dependency>
  <groupId>com.google.dagger</groupId>
  <artifactId>dagger</artifactId>
  <version>2.17</version>
</dependency>
```

The dagger package includes the classes we need at **runtime**

```
<dependency>
  <groupId>com.google.dagger</groupId>
  <artifactId>dagger-compiler</artifactId>
  <version>2.17</version>
  <scope>provided</scope>
</dependency>
```

The compiler package includes the **annotation processors** that are executed at build time

In practice this means

```
@Inject
public DdbDatabaseService(final TableNameProvider tableNameProvider) {...}
```

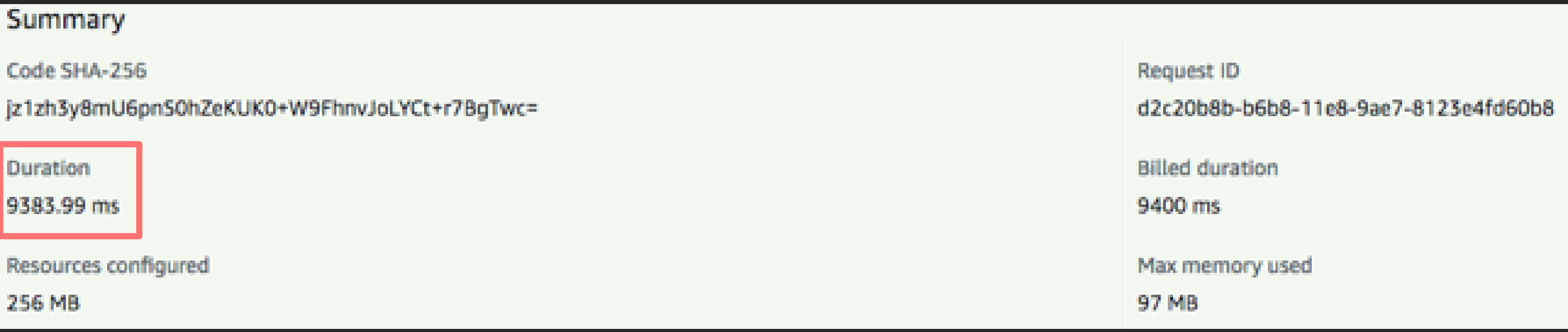
Use **@Inject** annotations

```
public class CreatePetHandler implements RequestHandler<Pet, Pet> {
    private static DatabaseService db = DaggerDatabaseService.create();

    @Override
    public Pet handleRequest(Pet input, Context context) {
```

Compile once to **generate** the Dagger object

My first test with Dagger 2



Looks like we shaved off only **~3 seconds** (22%); not what I expected

Looking at the logs tells me

- The cold start initialization process and dependency injection took only **400ms**—this was the desired effect
- The DynamoDB PutItem call took **8.9 seconds**—not what I expected at all

Looking at the logs tells me

- The cold start initialization process and dependency injection only took **400ms**—this was the desired effect
- The DynamoDB PutItem call took **8.9 seconds**—not what I expected at all
  - The AWS SDK for Java 2.0 uses **Jackson's high-level** APIs to marshal and unmarshal errors and data
  - The marshallers are **initialized lazily**, and Jackson relies heavily on reflection to understand models
  - Fortunately, Jackson **caches** all of the "discovered" fields so that the SDK introspects objects only once

How can we address this?

- Lazily loading marshallers happens once the **CPU is throttled**

- We can force the AWS SDK to **exercise its marshallers at initialization**

I will pay for my sins another time

- Using a static initializer, we can pre-warm the SDK by making a call we know will fail

```
static {
    Pet invalidPet = new Pet();
    invalidPet.setName("invalid");
    invalidPet.setBreed("invalid");
    db.primePet(invalidPet);
}
```

Let’s test again without reflection

Code SHA-256	Request ID
jz1zh3y8mU6pnS0hZeKUK0+W9FhnvJoLYCt+r7BgTwc=	30b80314-b6bd-11e8-88ef-b7582cdf4a5b
Duration	Billed duration
1049.50 ms	1100 ms
Resources configured	Max memory used
256 MB	102 MB

Reflection really is evil

Code SHA-256

jz1zh3y8mU6pnS0hZeKUK0+W9FhnvJoLYCt+r7BgTwc=

Duration

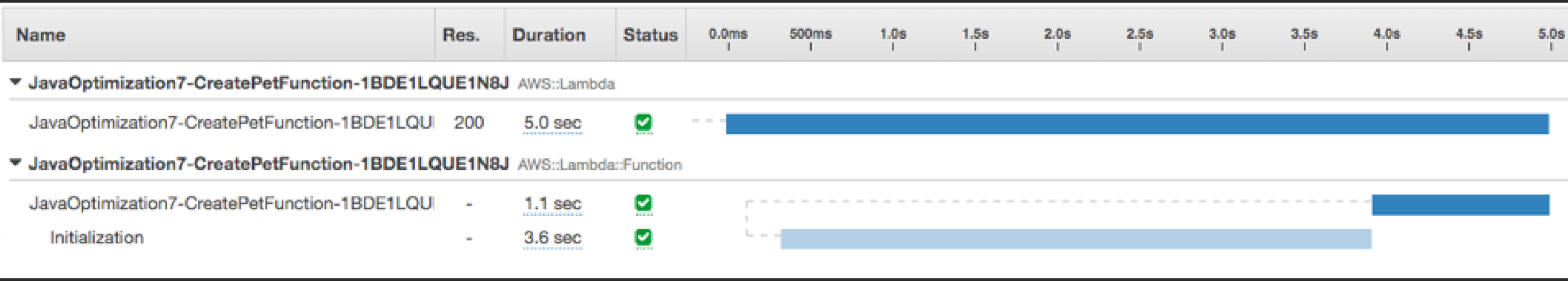
1049.50 ms

Resources configured

256 MB

9.3 to 1 second: **89% improvement**

Not so fast, hotshot



The real number is **5 seconds**—the init time increased by **~700ms**

- In reality we've gone from 12 to 5 seconds: **58% improvement**
- Handler time has gone from 9.3 to 1 second: **88% improvement**

Intermission: What's wrong with reflection?

Reflection is slower because…

1. Class loading by name prevents the JVM from pre-loading or performing any **optimization**
2. Method lookup/invocations means going through Java Native Interface (**JNI**) or using a generated `MethodAccessor`—the former is slow, and the latter consumes memory

```
Benchmark       Mode   Cnt    Score  Error         Units
=============================================================
DirectAccess    avgt   60     2.590  ± 0.014       ns/op
Reflection      avgt   60     5.275  ± 0.053       ns/op
```

JNI (Java Native Interface)

- The Method object in Java uses a `MethodAccessor` implementation to invoke a method discovered via reflection; **`NativeMethodAccessor`** is the JNI-based implementation of this
- `NativeMethodAccessor` **talks to the JVM** through JNI to retrieve class, object, and method metadata—trampolining is expensive:
  - Native methods are **never inlined**
  - Java array (stack) is **copied** both ways
  - Objects and callbacks will trigger **further JNI calls**
  - Signatures use Strings: Java Strings are objects, have length, and are encoded—accessing or creating a string may require an **O(n) copy**

GeneratedMethodAccessor

- After x invocations of a method through JNI, the JVM will **generate Java bytecode** to implement a `GeneratedMethodAccessor`
  - The value of x is 15 by default and configured on `sun.reflect.inflationThreshold`
- Generating the bytecode takes **time**
- Loading the new class takes **time**
- More classes = more **memory** consumed

Once an accessor is generated, we are all good, right?

- If reflection is used only at boot (cold start), we don’t really get to take advantage of all that work going forward—it just makes the **cold start colder**
- If we do need the accessors at runtime, we are likely consuming more memory than we ought to—in Lambda this directly translates to **cost** as you’ll have to allocate more memory

- Either way, **we lose**

Key tips

Key tips

1. Load all required classes during **initialization**
2. Try to **avoid reflection** like the plague
3. Each class is more bytecode to load, I/O access—**less is more**
4. If you need to, **prime dependencies**

Think hard

- If you are writing a builder for a class that takes **one constructor parameter**
- If you are injecting dependencies just **for the sake of DI**
- Or if you are declaring an interface that will only ever have a **single implementation**
- Please **STOP** and consider whether you really need them

Great, but it was still 5 seconds

There's **a lot we can do** to make our code faster

Decades of evolution of the Java ecosystem are not going to be re-written **in a year** (or two, or ten)

*Back to Tim's point… If we want Java to be a successful citizen of the serverless ecosystem, we need to **do our part** and adjust the way we architect our Java code*

Ready to adjust—where to next?

New approaches, new frameworks

Fortunately, there’re lots of very smart people in the Java ecosystem thinking about the **cold start problem**

It isn’t just relevant to serverless—**containers** also need to start very fast

New approaches, new frameworks

Fortunately, there're lots of very smart people in the Java ecosystem thinking about the **cold start problem**

It isn't just relevant to serverless—**containers** also need to start very fast

Quarkus

https://www.graalvm.org/

https://quarkus.io/

https://micronaut.io/

New approaches, new frameworks

Fortunately, there’re lots of very smart people in the Java ecosystem thinking about the **cold start problem**

It isn’t just relevant to serverless—**containers** also need to start very fast

https://www.graalvm.org/

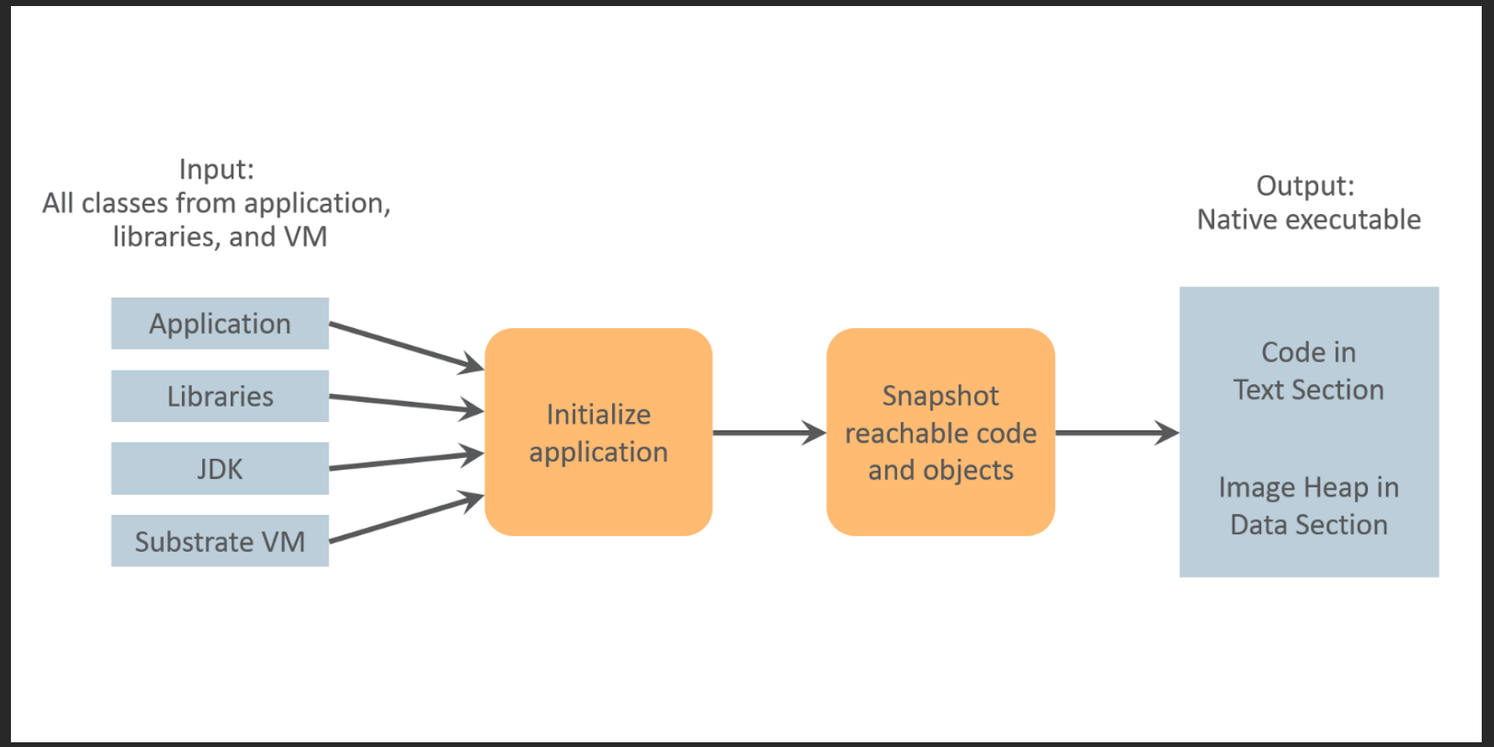
GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Groovy, Kotlin, Clojure, and LLVM-based languages such as C and C++

New approaches, new frameworks

Fortunately, there’re lots of very smart people in the Java ecosystem thinking about the **cold start problem**

GraalVM™

https://www.graalvm.org/



Compile Java programs to **native executables**

New approaches, new frameworks

Fortunately, there’re lots of very smart people in the Java ecosystem thinking about the **cold start problem**

Constantly improving with cool features such as **profile-guided optimization** (PGO)

https://www.graalvm.org/

Compile Java programs to **native executables**

New approaches, new frameworks

Fortunately, there're lots of very smart people in the Java ecosystem thinking about the **cold start problem**

It isn't just relevant to serverless—**containers** also need to start very fast

Quarkus
https://quarkus.io/

Micronaut
https://micronaut.io/

Full stack frameworks for building modern Java applications—optimized for native compilation via GraalVM & OpenJDK HotSpot

Easily create DI-enabled, full-stack applications

But I’m a Spring developer, you say

Indeed, the last thing we want to do is **rewrite** all the code we have ever written in our career

But I'm a Spring developer, you say

Indeed, the last thing we want to do is **rewrite** all the code we have ever written in our career

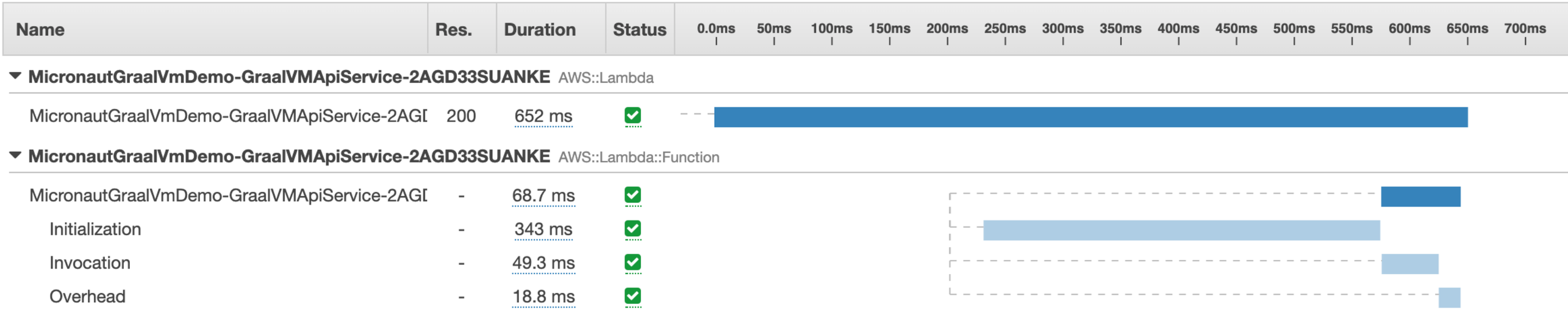
Quarkus
https://quarkus.io/

Micronaut
https://micronaut.io/

Good news: Both Quarkus and Micronaut are **compatible with some of Spring's annotations**

Let me show you a sample

This is our Pet Store sample, using Spring's annotations and compiled with Micronaut to a native executable



652ms cold start is within reason

http://bit.ly/aws-lambda-api-samples

There is hope yet

These technologies are in their **infancy**, but they are **evolving fast**

Spring is working on native support for GraalVM's native image for **SpringBoot**

Keep our key takeaways in mind and **we may still be writing Java in AWS Lambda a few years from now**

http://bit.ly/aws-lambda-api-samples

Demo

Related breakouts

SVS219-S	Serverless at scale
SVS308-R	Moving to event-driven architectures
SVS310	Securing enterprise-grade serverless apps
SVS405-R	A serverless journey: AWS Lambda under the hood
SVS407-R	Architecting and operating resilient serverless systems at scale

Thank you!

Stefano Buliani

@sapessi

Please complete the session survey in the mobile app.