AWS Lambda function performance

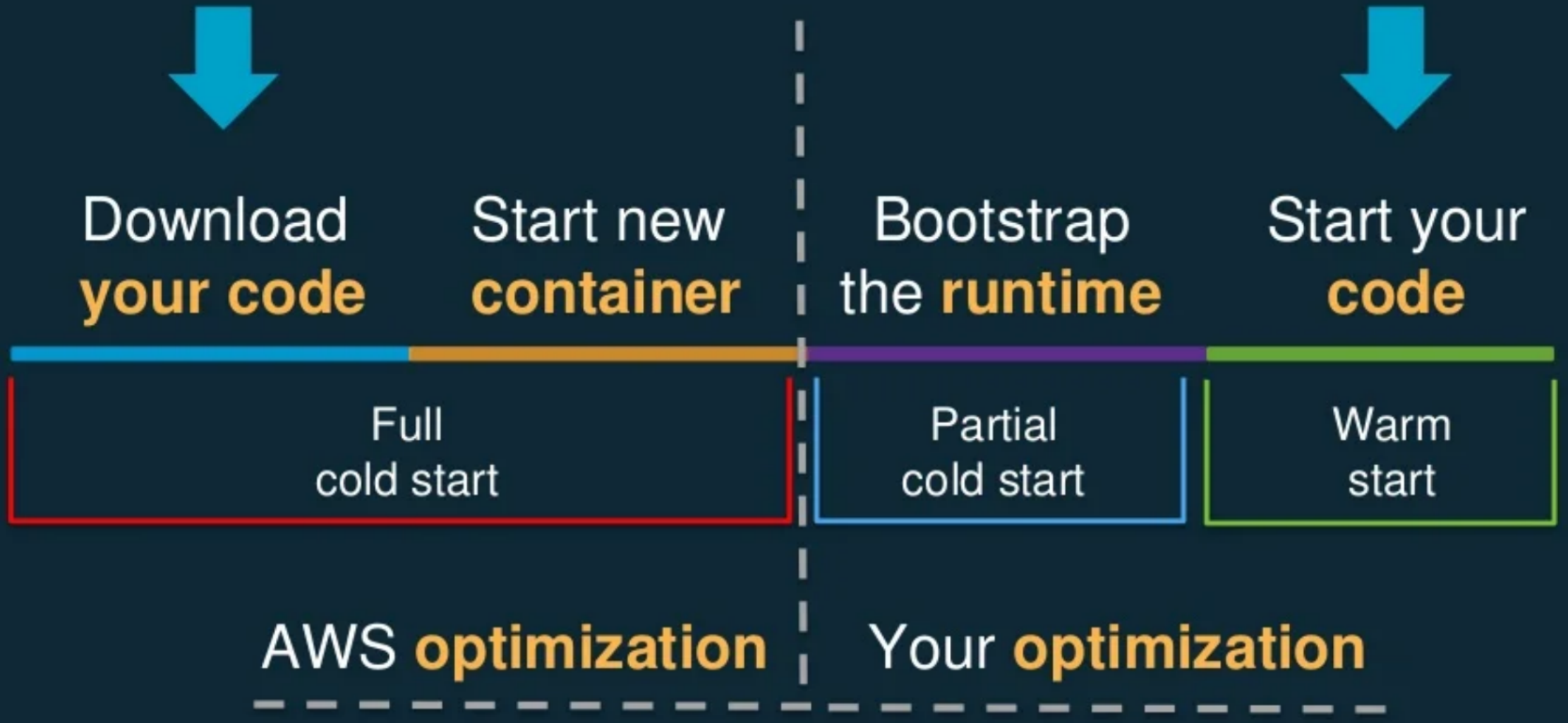
Radek Švanda

radek.svanda@aurora.io

Topics

- Lambda function lifecycle
- General recommendations
- Active tracing
- Application overview
- Kotlin
- Java
- GraalVM

The function lifecycle



aws

Easy wins

- Remove unnecessary dependencies

  Smaller package = faster upload & faster cold start

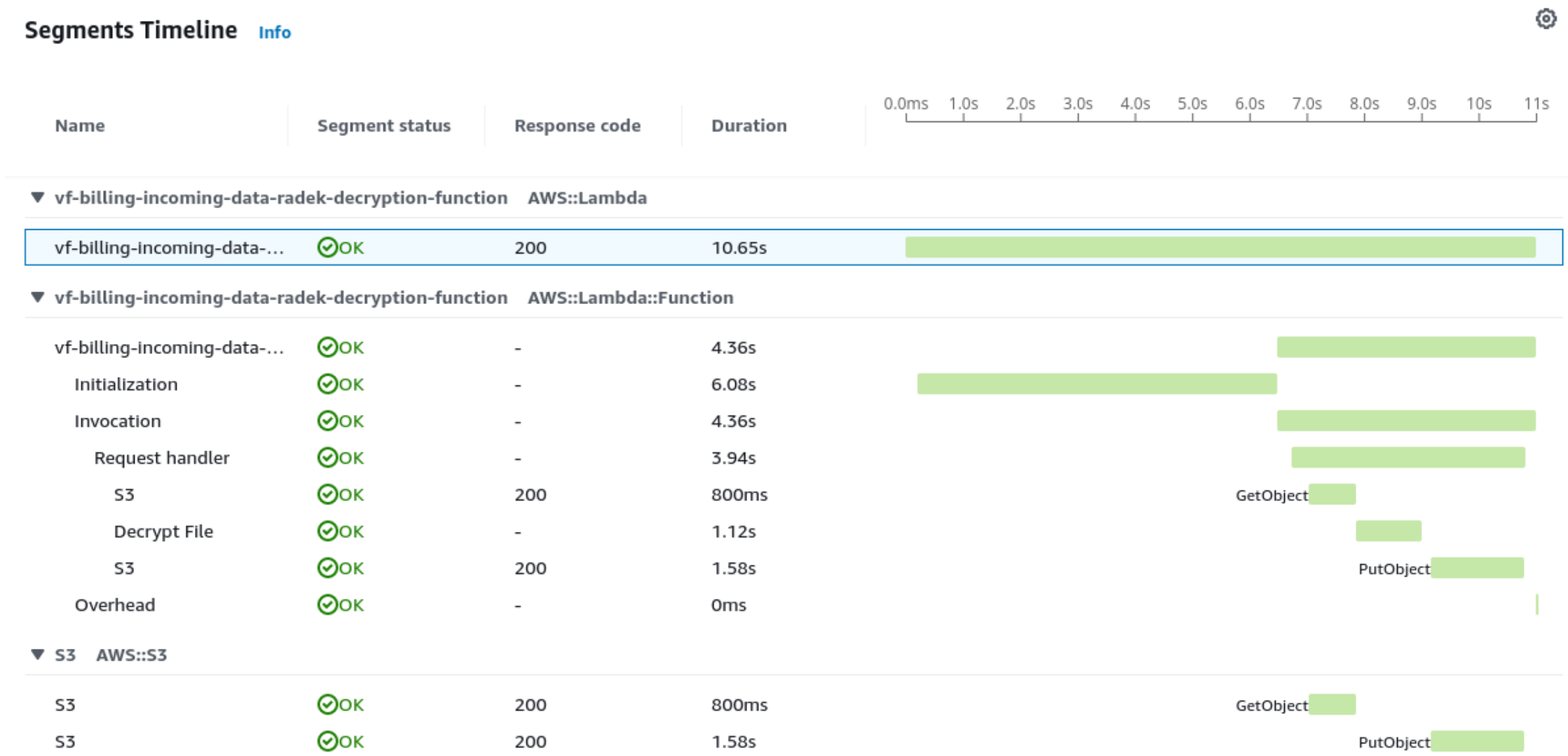
  ```
  mvn dependency:tree
  ```

- JDK JIT compiler C1 only - almost 100% speed gain instantly

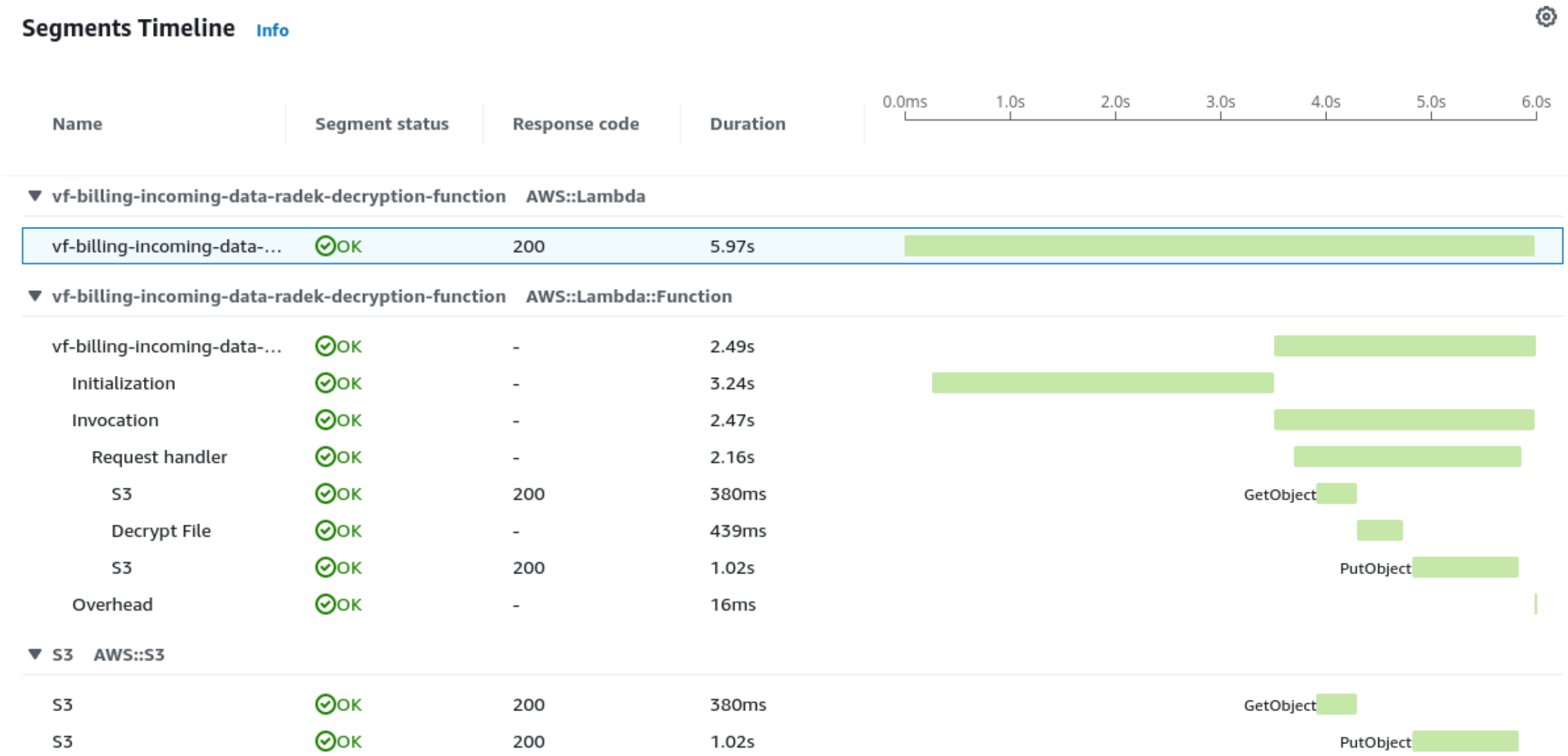
  ```
  Environment
    Variables:
      JAVA_TOOL_OPTIONS: "-XX:+TieredCompilation -XX:TieredStopAtLevel=1"
  ```

- Initialize as much as possible during start-up (**but only what you really need**)

Kotlin cold start (without JVM params) 🔗



Kotlin cold start (with JVM params)



- Exclude AWS SDK http clients - you run in a single thread you do not need NIO

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>url-connection-client</artifactId>
</dependency>
<groupId>software.amazon.awssdk</groupId>
  <artifactId>s3</artifactId>
  <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>
```

- Fully configure AWS SDK clients

```
S3Client.builder()
  .region(Region.of(System.getenv("AWS_REGION")))
  .httpClient(UrlConnectionHttpClient.builder().build()).build()
```

- Use smart configuration defaults

```
S3Client.builder().defaultsMode(DefaultsMode.IN_REGION).build()
```

- Ditch sophisticated logging

```
override fun handleRequest(input: SQSEvent, context: Context) {
  context.logger.log("hello there")
```

Falls back to

```
public void log(String message) { System.out.print(message) }
```

- Remove cold starts with provisioned concurrency

```
Properties:
  ProvisionedConcurrencyConfig:
    ProvisionedConcurrentExecutions: 20
```

  May get too expensive & slows down deployment

  → not a good solution for development environment

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

- Keep lambdas warm yourself

  Better for development environment

```
Properties:
  Events:
    Warmer:
      Type: Schedule
      Properties:
        Schedule: rate(5 minutes)
        Description: Lambda calling at scheduled time
        Input: '{ "warmer":true,"concurrency":1 }'
```

```
class GatewayRequest( var warmer: Boolean = false, var concurrency: Int = 1 ) : APIGatewayProxyRequestEvent()

private val client: LambdaClient = LambdaClient.builder().build()

override fun handleRequest(input: GatewayRequest, context: Context): APIGatewayProxyResponseEvent? {
  if (input.warmer) {
    val request = InvokeRequest.builder()
      .functionName(context.functionName)
      .payload("{ \"warmer\":true,\"concurrency\":${input.concurrency - 1} }").build()
    client.invoke(request)
  } else {
    ...
  }
```

Tracing the function with XRay

- Opentelemetry (preferred, flexible) vs. X-Ray SDK (tight integration)
- Turn on the tracing

```
MyFunction:
  Type: AWS::Serverless::Function
  Properties:
    Tracing: Active
```

- Send tracing data (use BOM `aws-xray-recorder-sdk-bom` for versioning)

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-xray-recorder-sdk-aws-sdk-v2</artifactId>
</dependency>
```

Tracing Subsegments

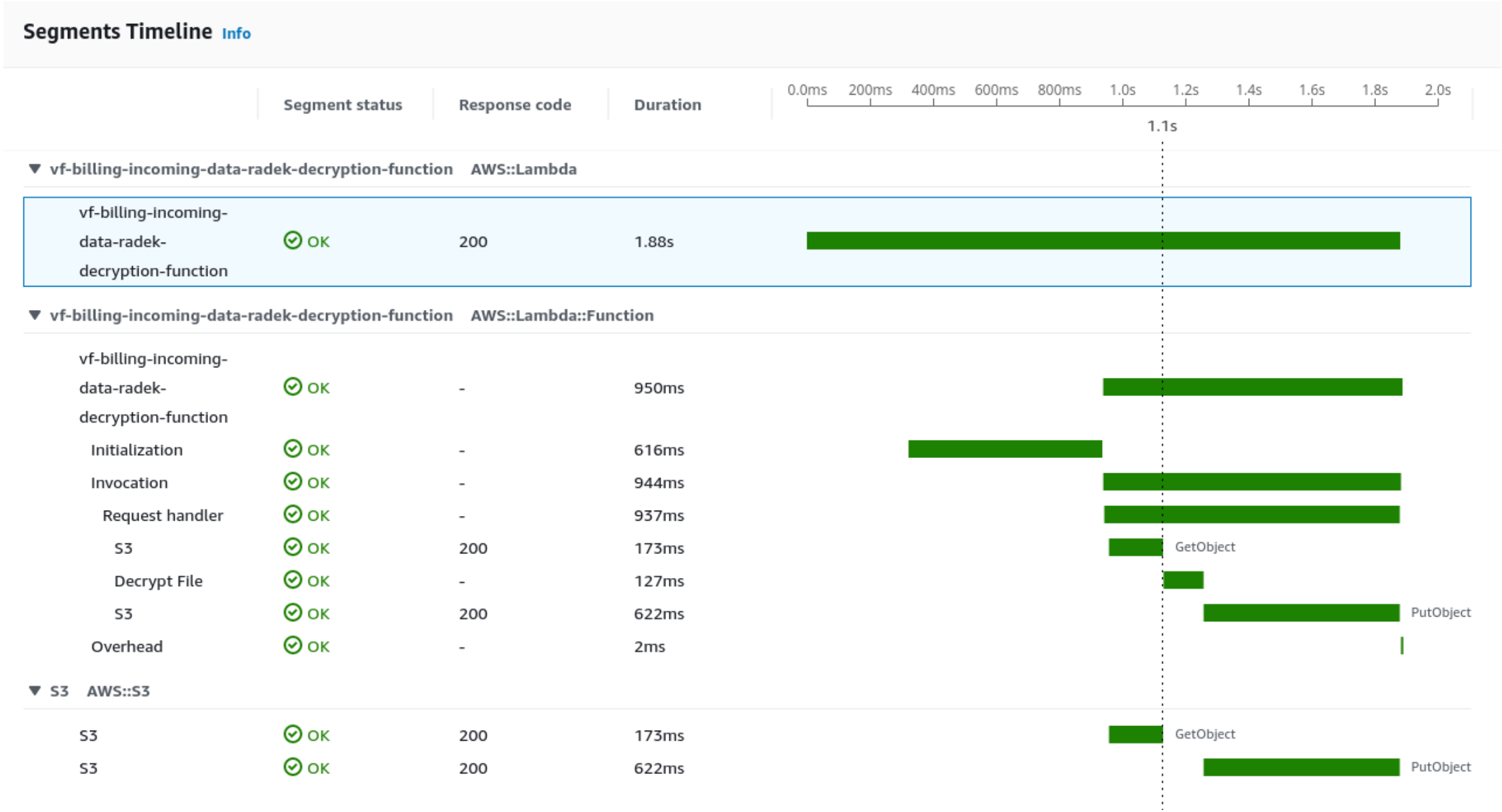
- Anywhere in your code

```
override fun handleRequest(input: SQSEvent, context: Context) {
  AWSXRay.beginSubsegment("Request handler")
  try { ... }
  finally { AWSXRay.endSubsegment() }
}
```

- AWS SDKs

```
S3Client.builder()
  .overrideConfiguration(ClientOverrideConfiguration.builder()
      .addExecutionInterceptor(TracingInterceptor()).build()
  )
```

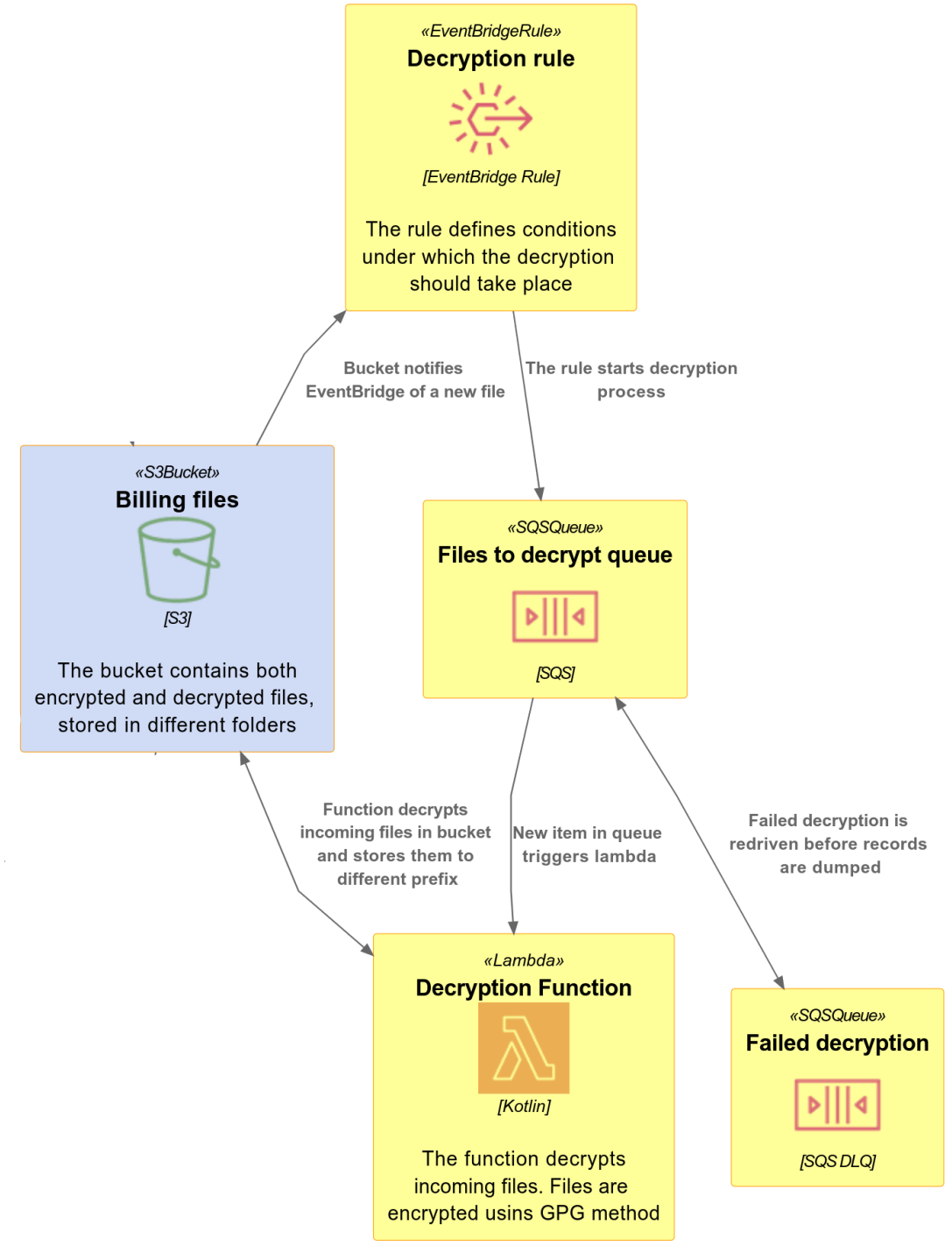
Example timeline



Application overview

Decryption function

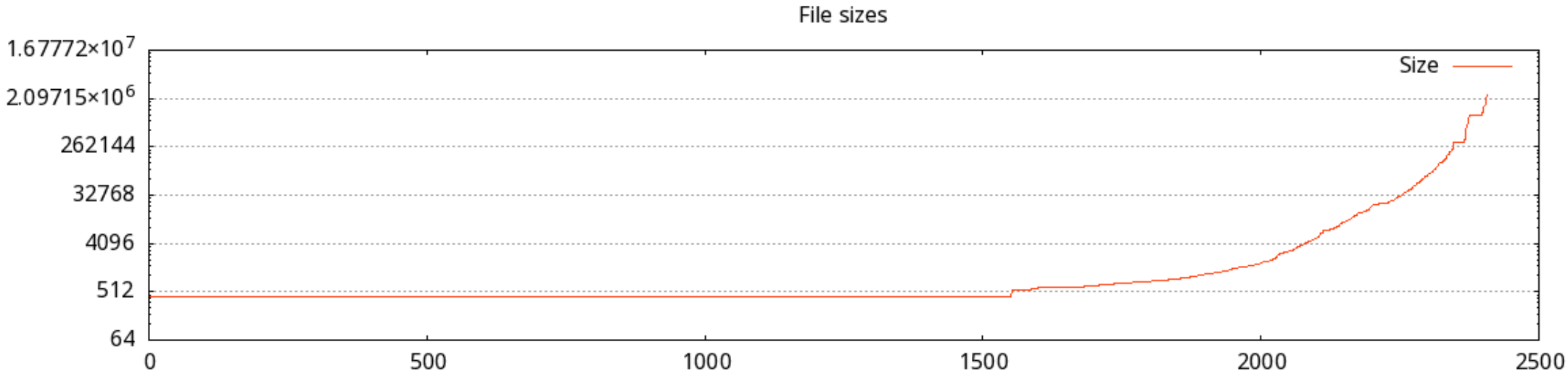
- Written in Kotlin
- Steps:
  - Downloads file from S3
  - Decrypts contents with PGP
  - Uploads decrypted file back



Performance testing

- ~2.500 PGP encrypted files
- Uploaded at once to source S3 bucket

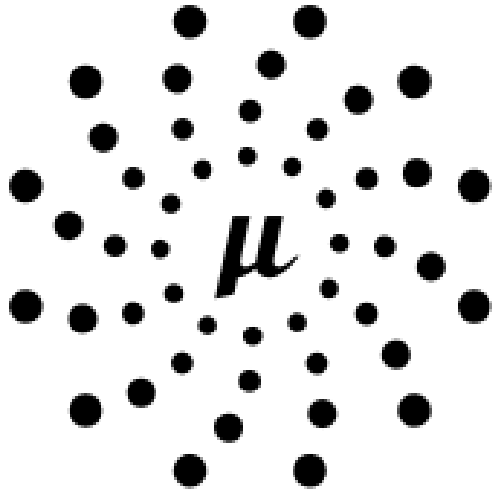
File size distribution



1st iteration: Micronaut framework

Pros:

- Out of the box Spring framework replacement
- Test tooling with mocking and spock integration
- Integration with ParamStore / SecretsManager



Problems:

- Package too large
- Slow starts
- Every. Single. One. Configuration property requested from ParamStore during startup

2nd iteration: Plain Kotlin

- Removed Micronaut
- Makefile packaged
- Most parameters configured during deployment

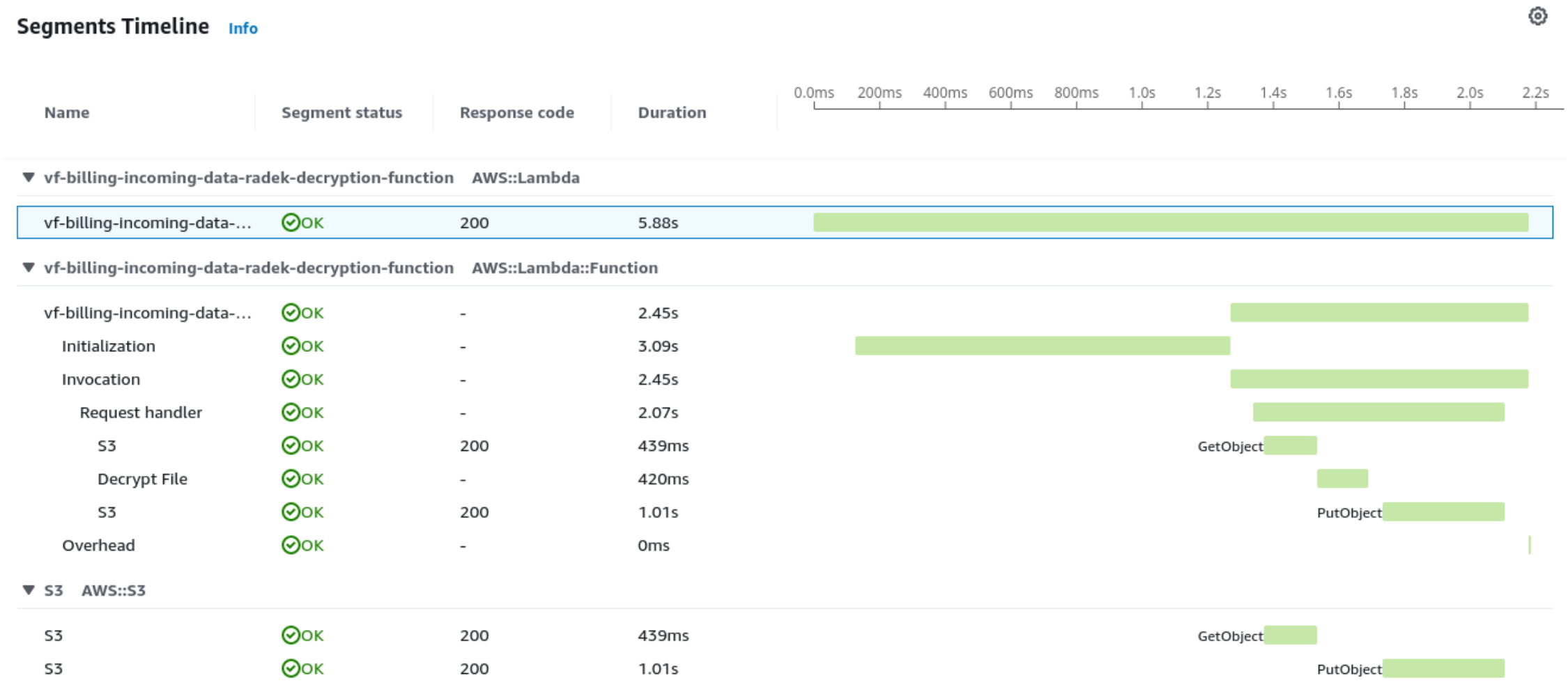
Pros:

- Smaller package, faster deploy

Cons:

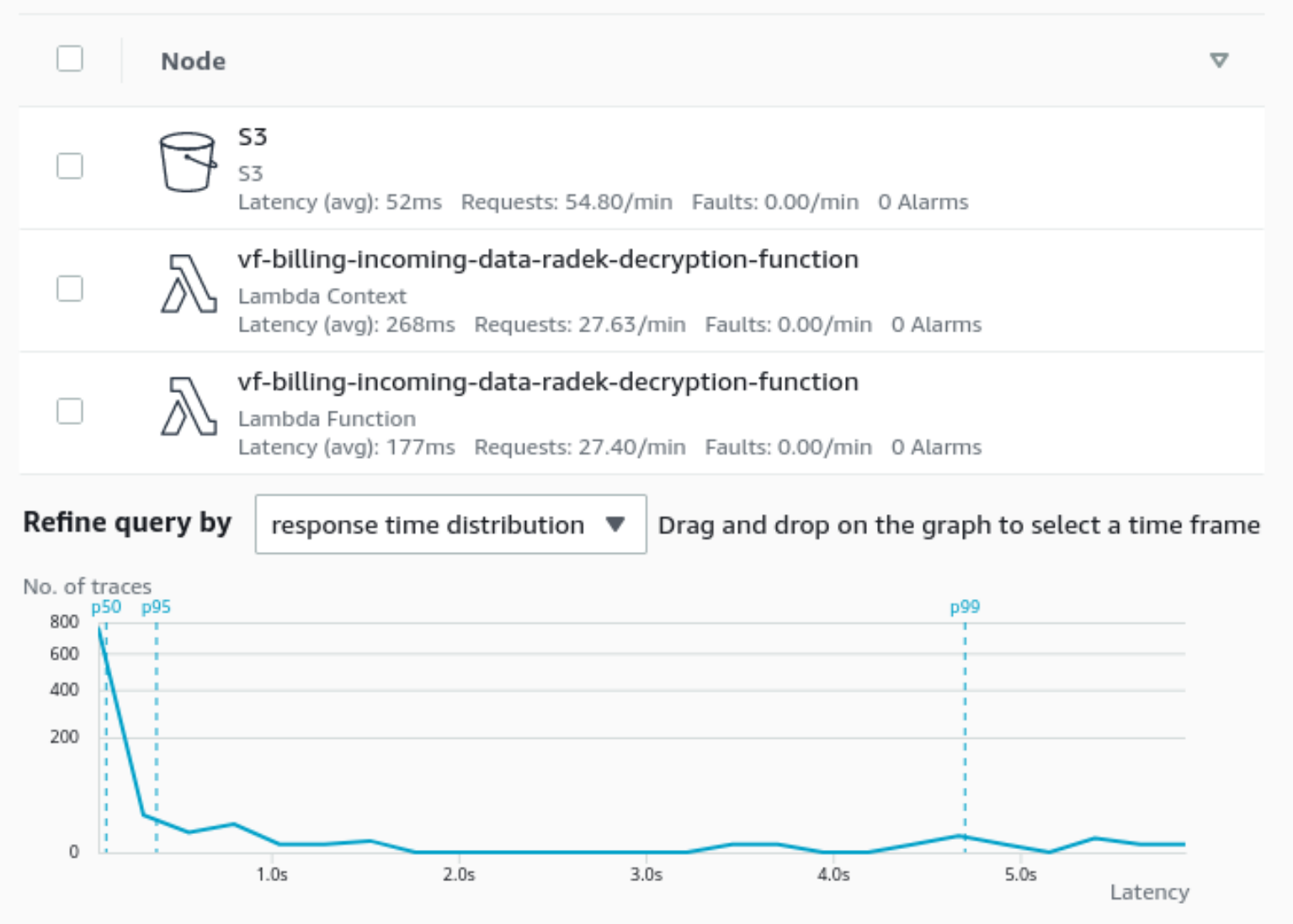
- Manual configuration handling
- Params security (visible variables in Lambda console)

Kotlin cold start 🔗



Kotlin response time distribution

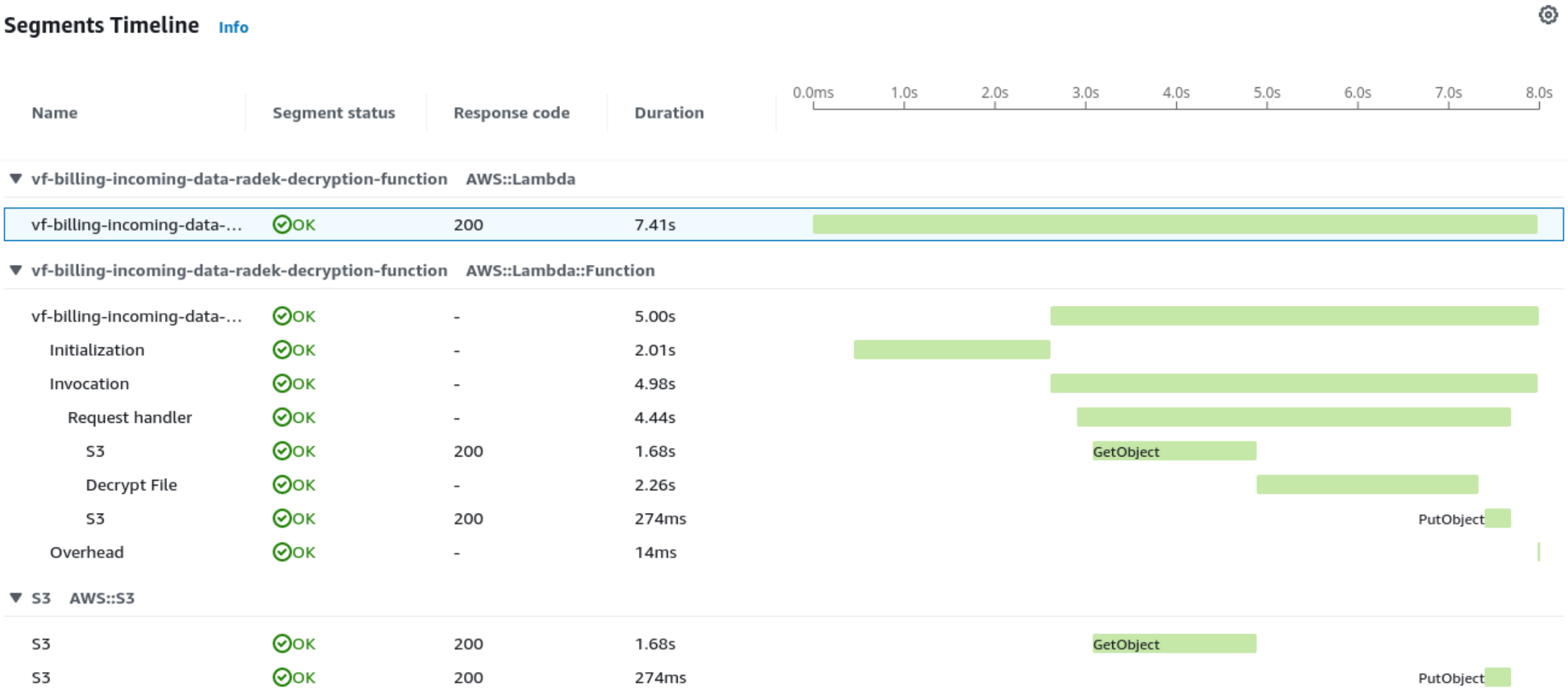
Percentile	ms
P50	120
P95	390
P99	4700
Max	5883



3rd iteration: Back to Java 11

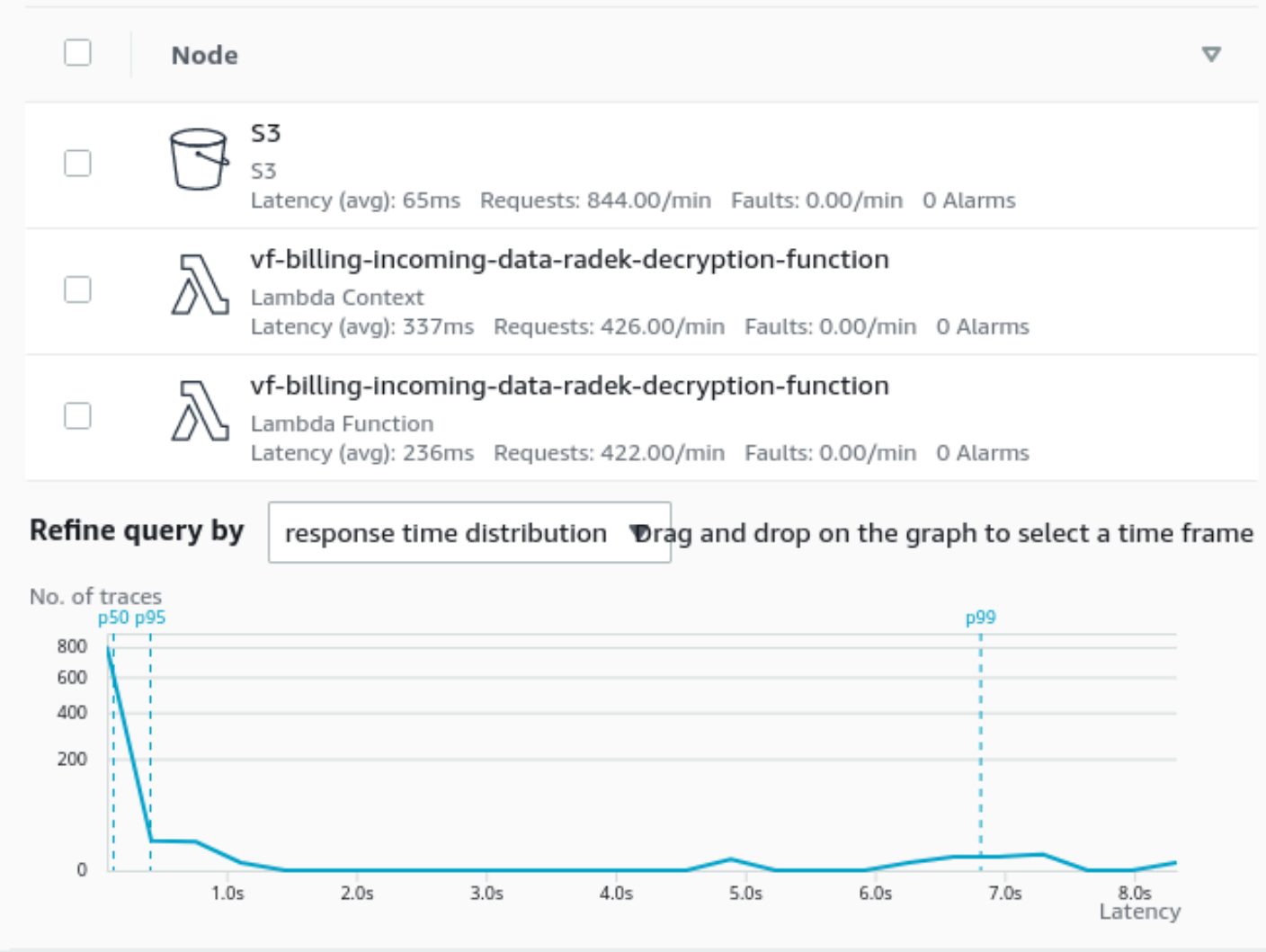
- Is a plain Java application faster in Lambda environment?
- Is Kotlin more expressive but also faster?

Java cold start



Java response time distribution 🔗

Percentile	ms	Kotlin
P50	130	120
P95	413	390
P99	6800	4700
Max	8328	5883



4th iteration: GraalVM native-image

- an ELF linux binary build from Java bytecode. See https://www.graalvm.org/22.3/reference-manual/native-image/

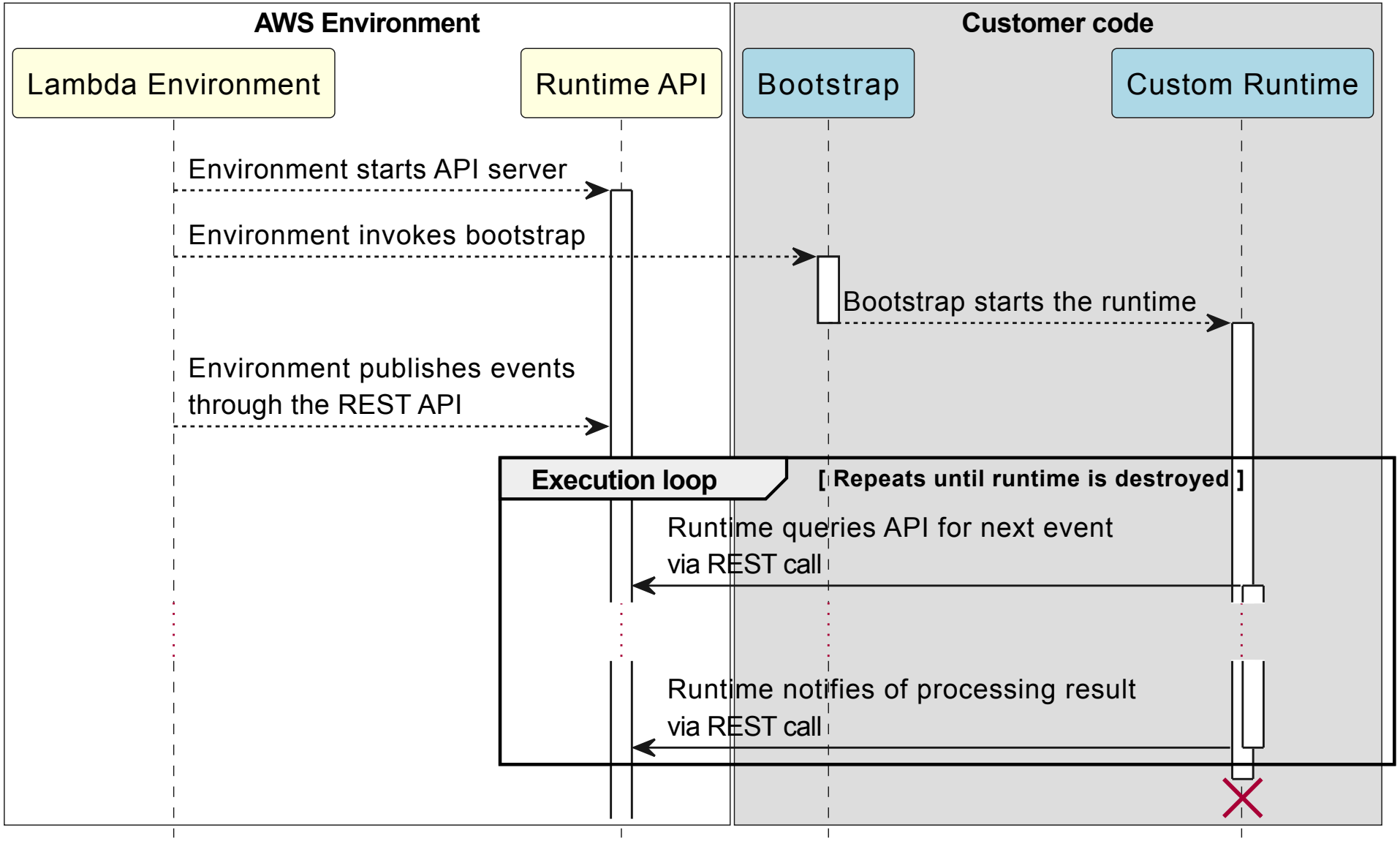
Pros:

- A fast binary
- Small package to upload
  80 MB binary package → 20 MB zipped upload

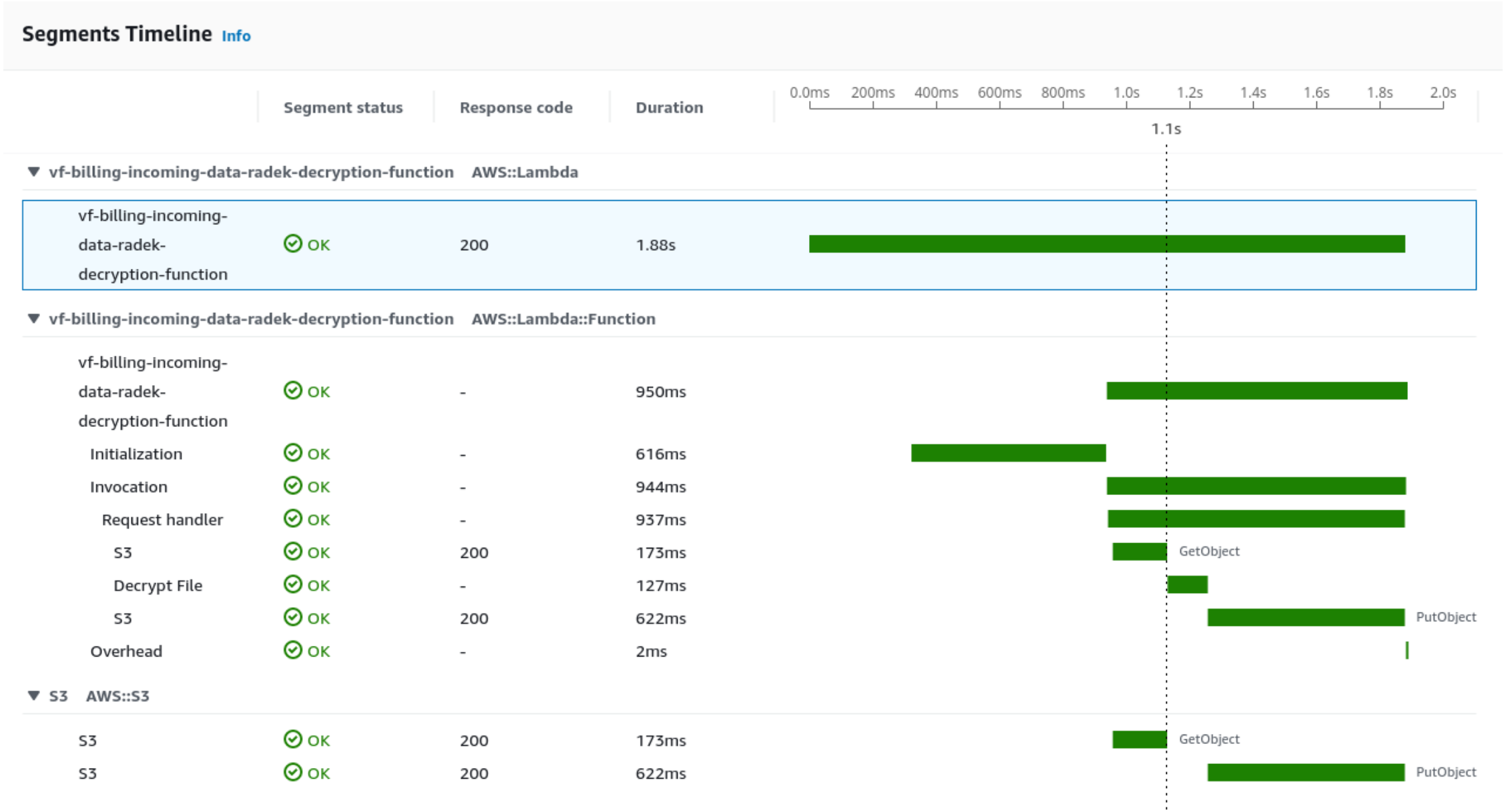
Cons:

- Everything has to be baked in during compile time
  No reflection during runtime (DI frameworks, logging, security ...)
- Sloooow build times (2-3 minutes, 12 CPU cores on 100%, peak mem 5-6GB)

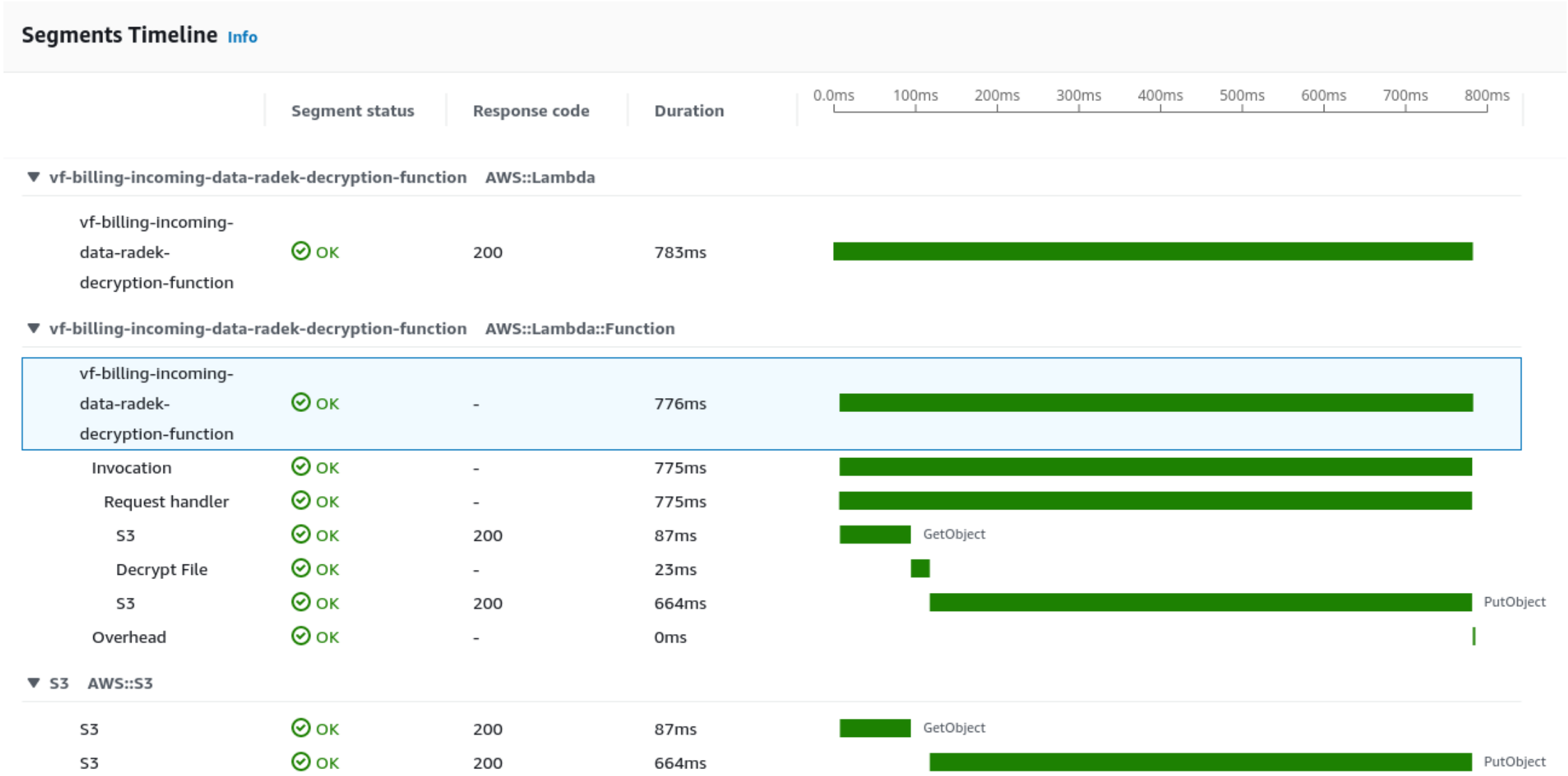
AWS Lambda custom runtimes



GraalVM cold start

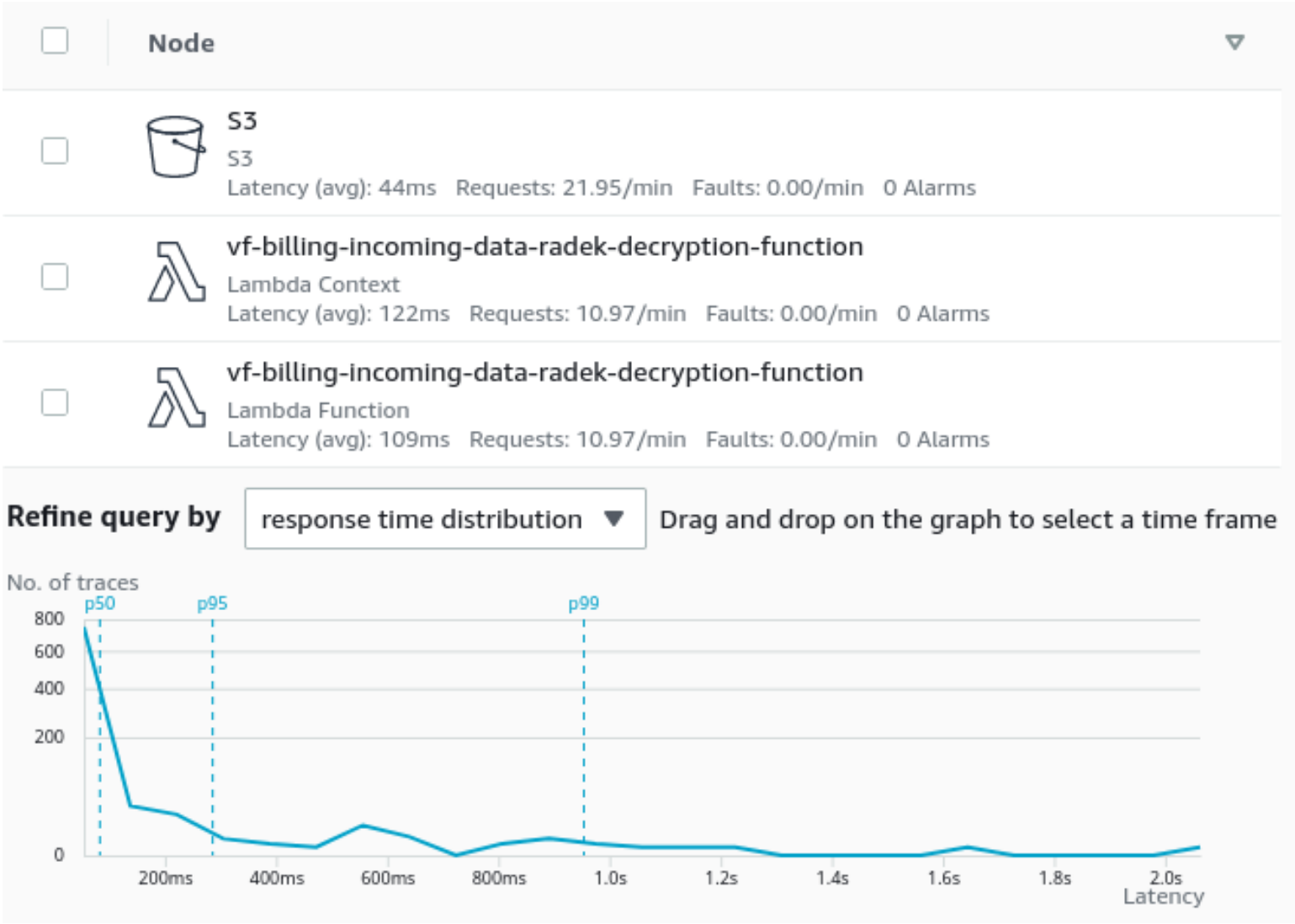


GraalVM warmed request



GraalVM response time distribution

Percentile	ms
P50	84
P95	286
P99	954
Max	2064



Let's try something different: Javascript

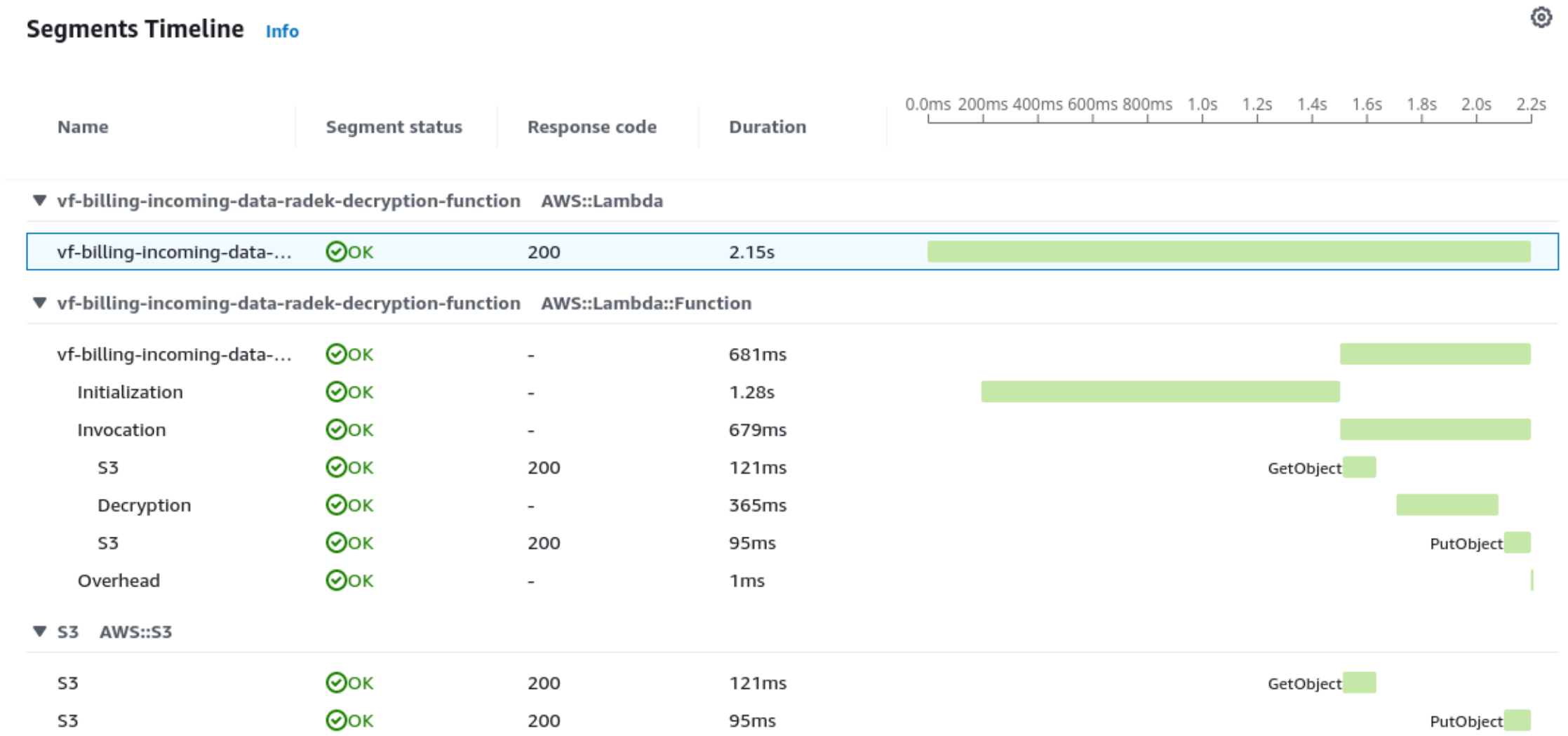
Pros:

- Everybody knows it
- No build time
- Small packages, fast uploads

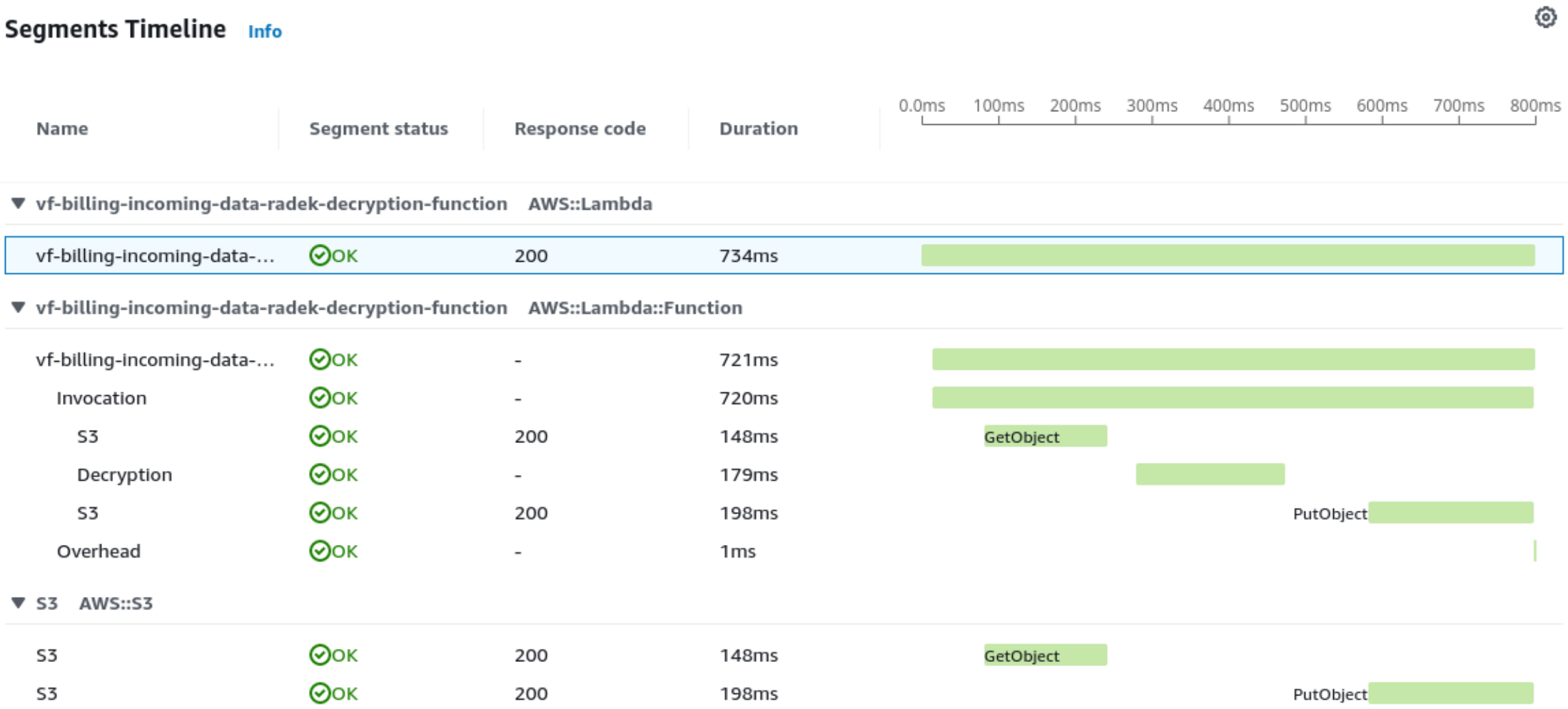
Cons:

- 3 different versions of AWS SDK
- AWS documentation not on par with Java

Node.js cold start

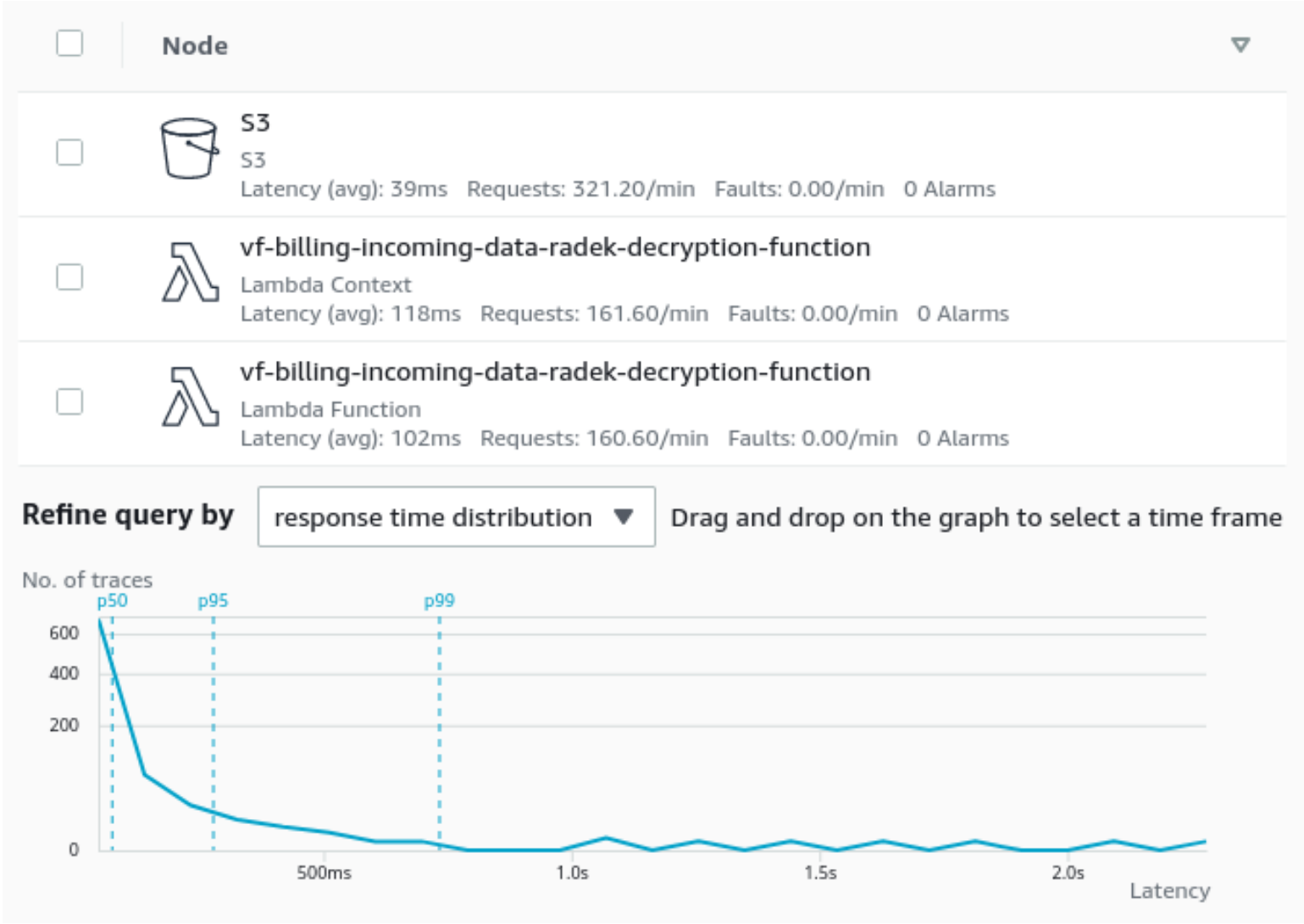


Node.js warm run



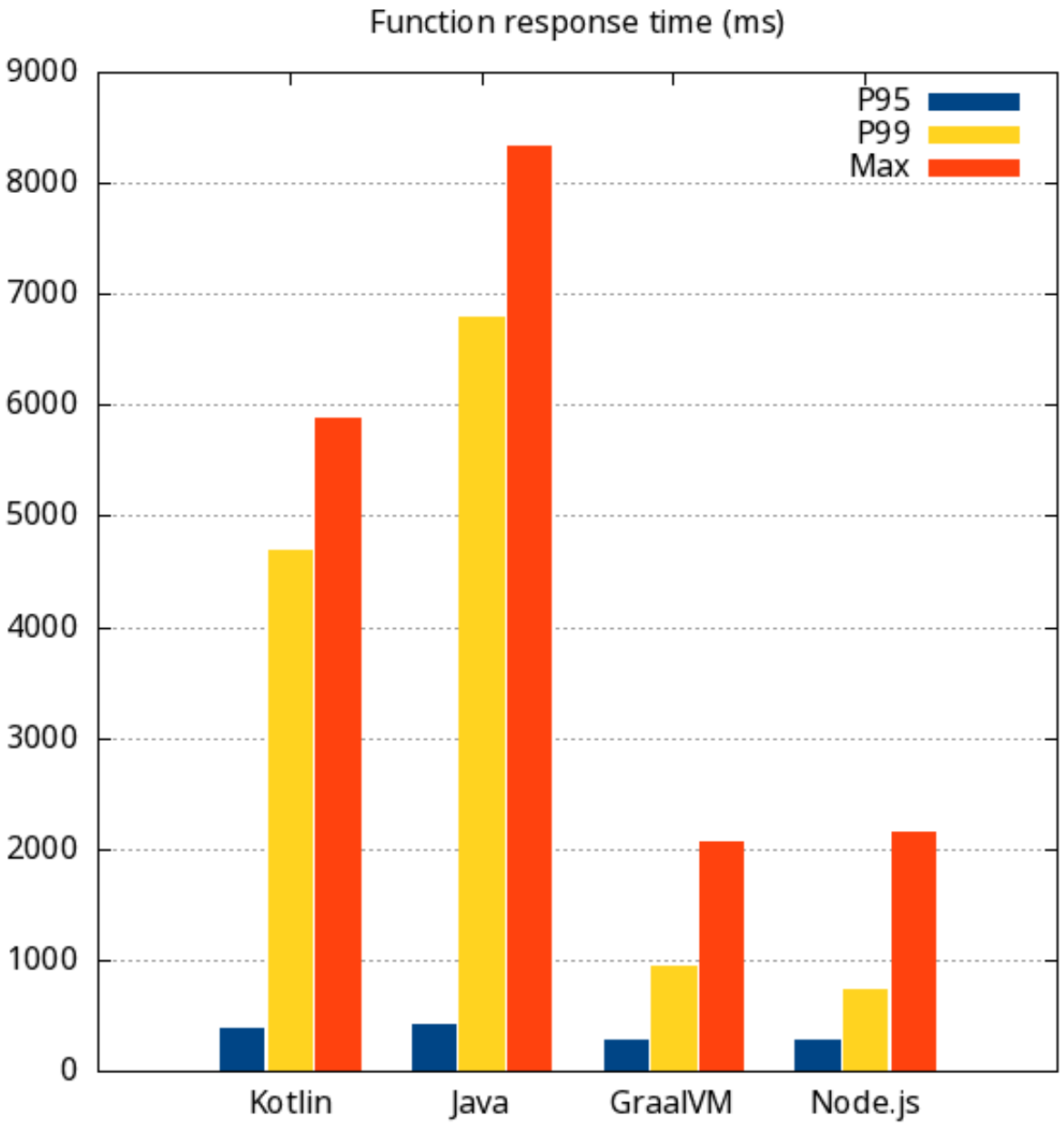
Node.js response time distribution

Percentile	ms
P50	75
P95	279
P99	734
Max	2160



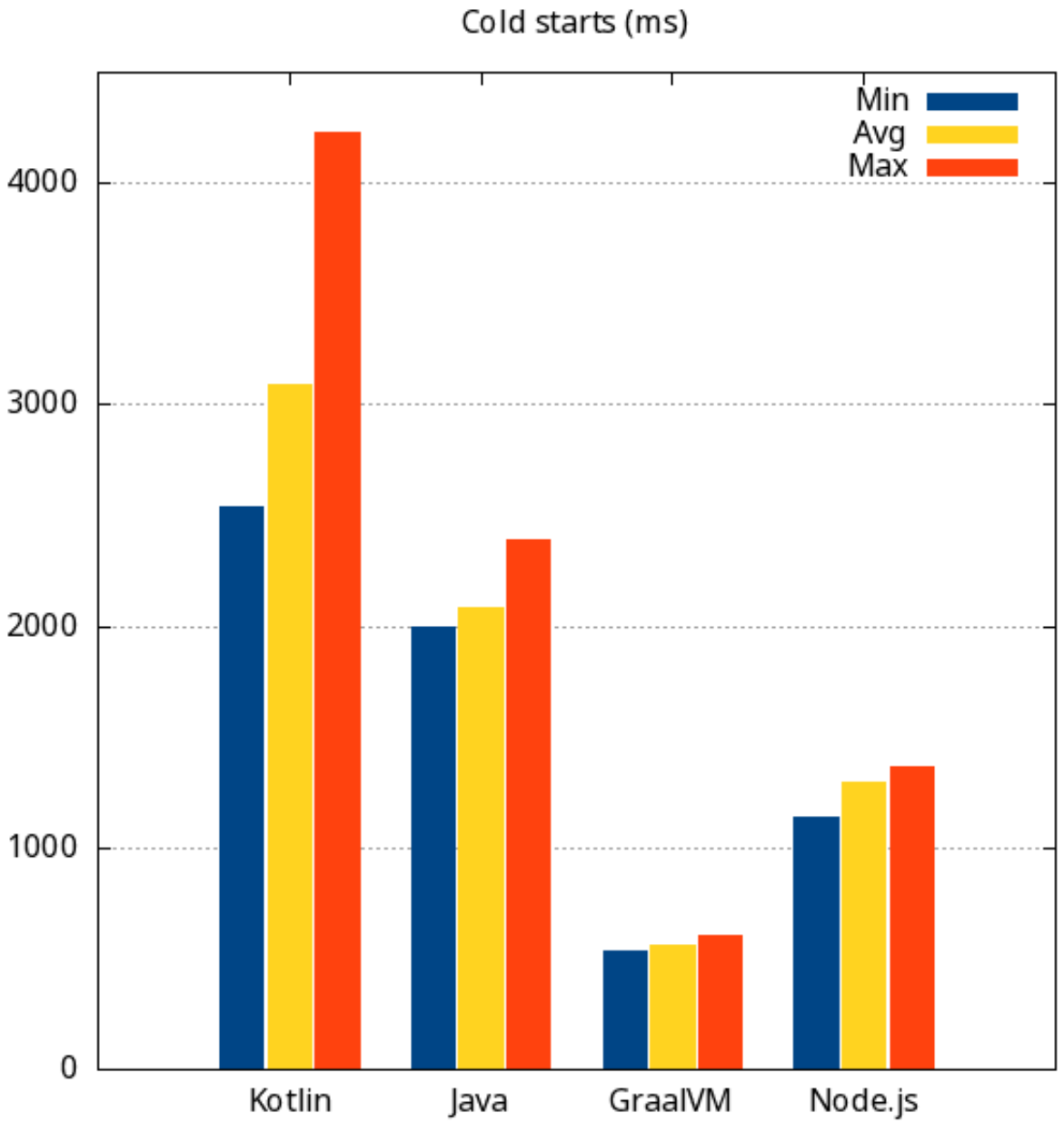
Response time summary

Lang	P95	P99	Max
Kotlin	390	4700	5883
Java	413	6800	8328
GraalVM	286	954	2065
Node.js	279	734	2160



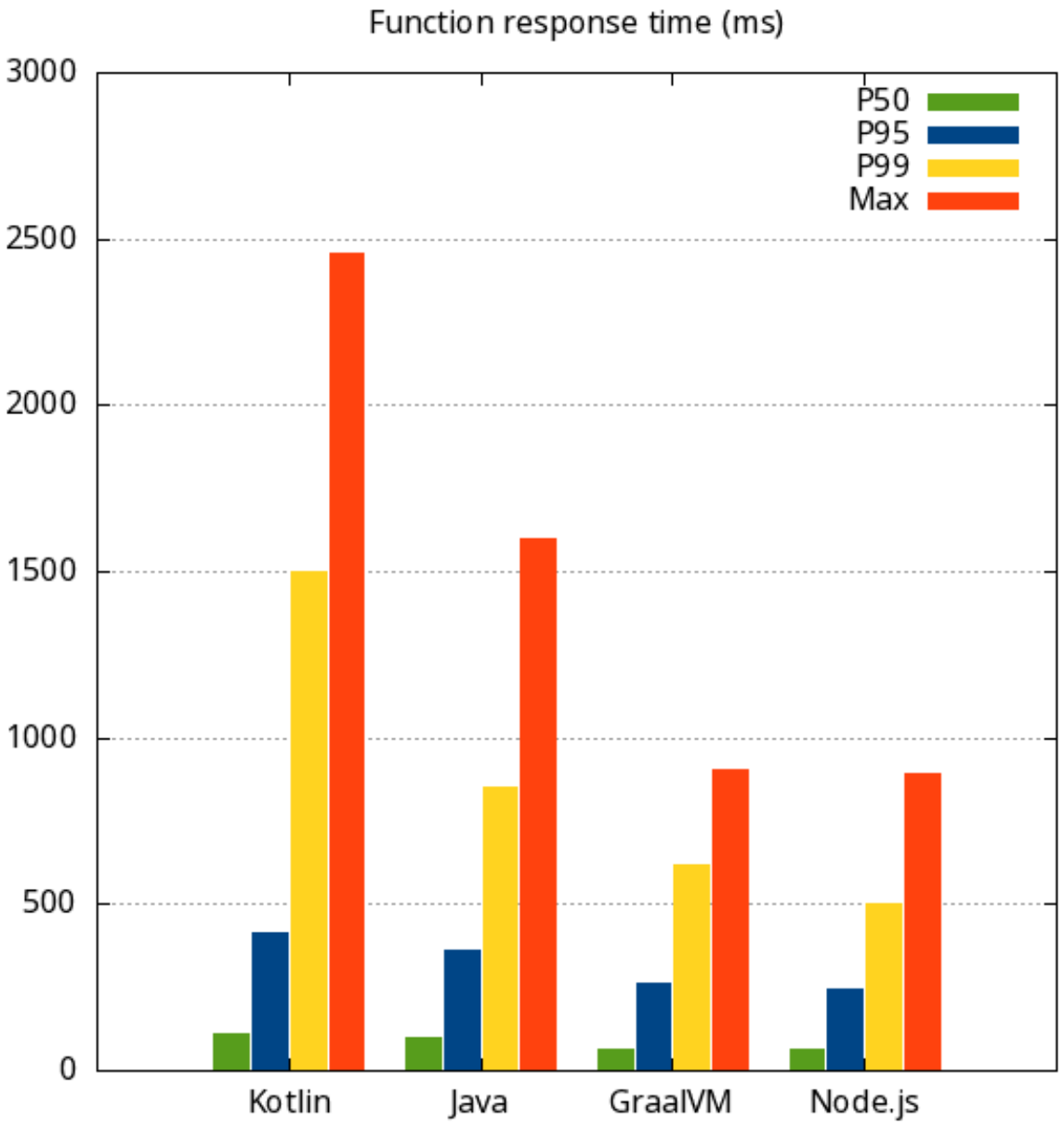
Cold starts

Language	Count
Kotlin	30
JAVA	28
Graal	17
Node	17



Results with provisioned concurrency

Lang	P50	P95	P99	Max
Kotlin	109	414	1500	2460
Java	99	360	850	1599
GraalVM	67	260	618	902
Node.js	67	247	503	894



Summary

- Keep your Lambda functions small and simple
- Consider Java if your concern is low latency
    - easier GraalVM compilation later
    - Use Java SDK 2 with targeted configuration
    - Avoid reflection
- Consider Javascript if not sure (with Node SDK v3)

Thanks for your attention

Radek Švanda

radek.svanda@aurora.io