

High Performance Serverless Java on AWS



Contact



Vadym Kazulkin

ip.labs GmbH Bonn, Germany



Co-Organizer of the Java User Group Bonn



v.kazulkin@gmail.com



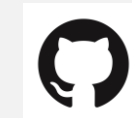
<https://www.linkedin.com/in/vadymkazulkin>



[@VKazulkin](https://twitter.com/VKazulkin)



<https://dev.to/vkazulkin>



<https://github.com/Vadym79/>



<https://de.slideshare.net/VadymKazulkin/>

About ip.labs



Java Popularity



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#

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 (since April 2023)
- Corretto Java 21(since November 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.

AWS People behind Amazon Corretto



James Gosling · 2.

Software Engineer



Amazon Web Services



Volker Simonis · 1.

Principal Software Engineer at Amazon Web Services (AWS) / OpenJDK Reviewer



Amazon Web Services (AWS)



Universität Tübingen



Aleksey Shipilëv · 2.

Principal Engineer at AWS



Amazon Web Services (AWS)



Roman Kennke · 1.

Principal Engineer, AWS



Amazon

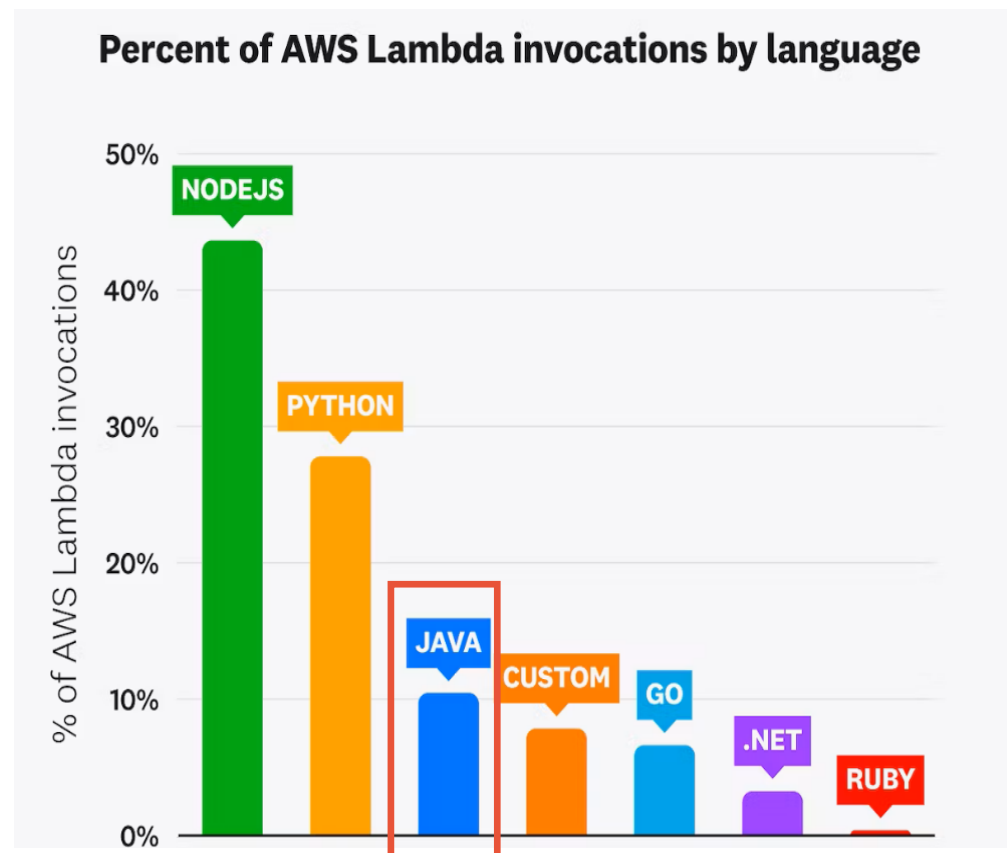
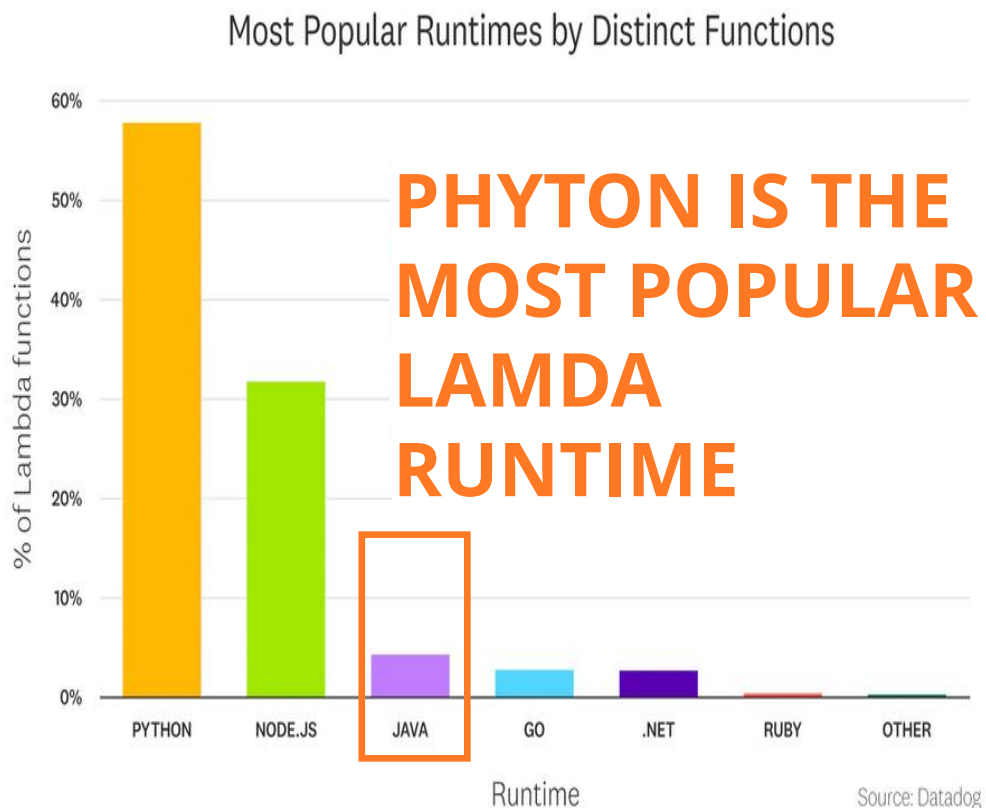


Heinrich Heine Universität Düsseldorf

**Java is a very
fast and
mature
programming
language...**

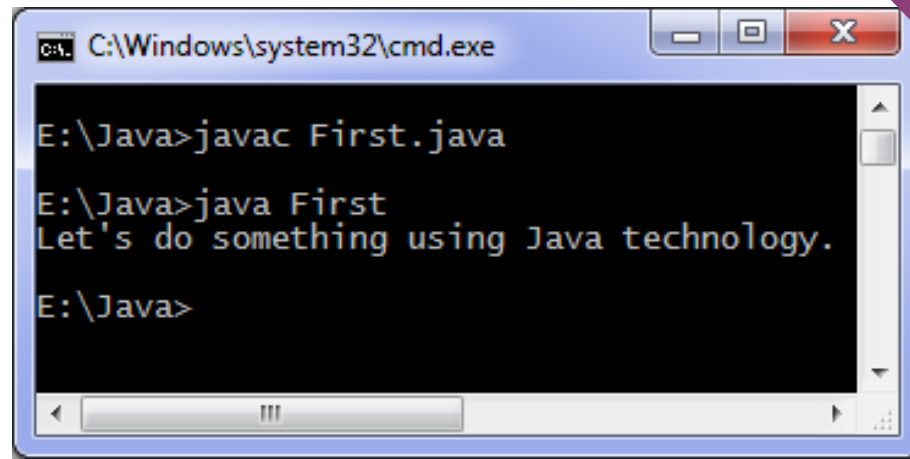
**... but
serverless
adoption of
Java looks like
this!**

Percent of AWS Lambda Invocations by Language 2021 vs 2023



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

But what are the challenges ?

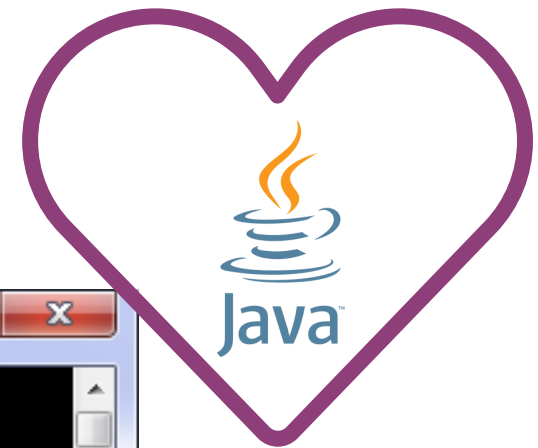


```
C:\Windows\system32\cmd.exe

E:\Java>javac First.java

E:\Java>java First
Let's do something using Java technology.

E:\Java>
```



Serverless with Java Challenges

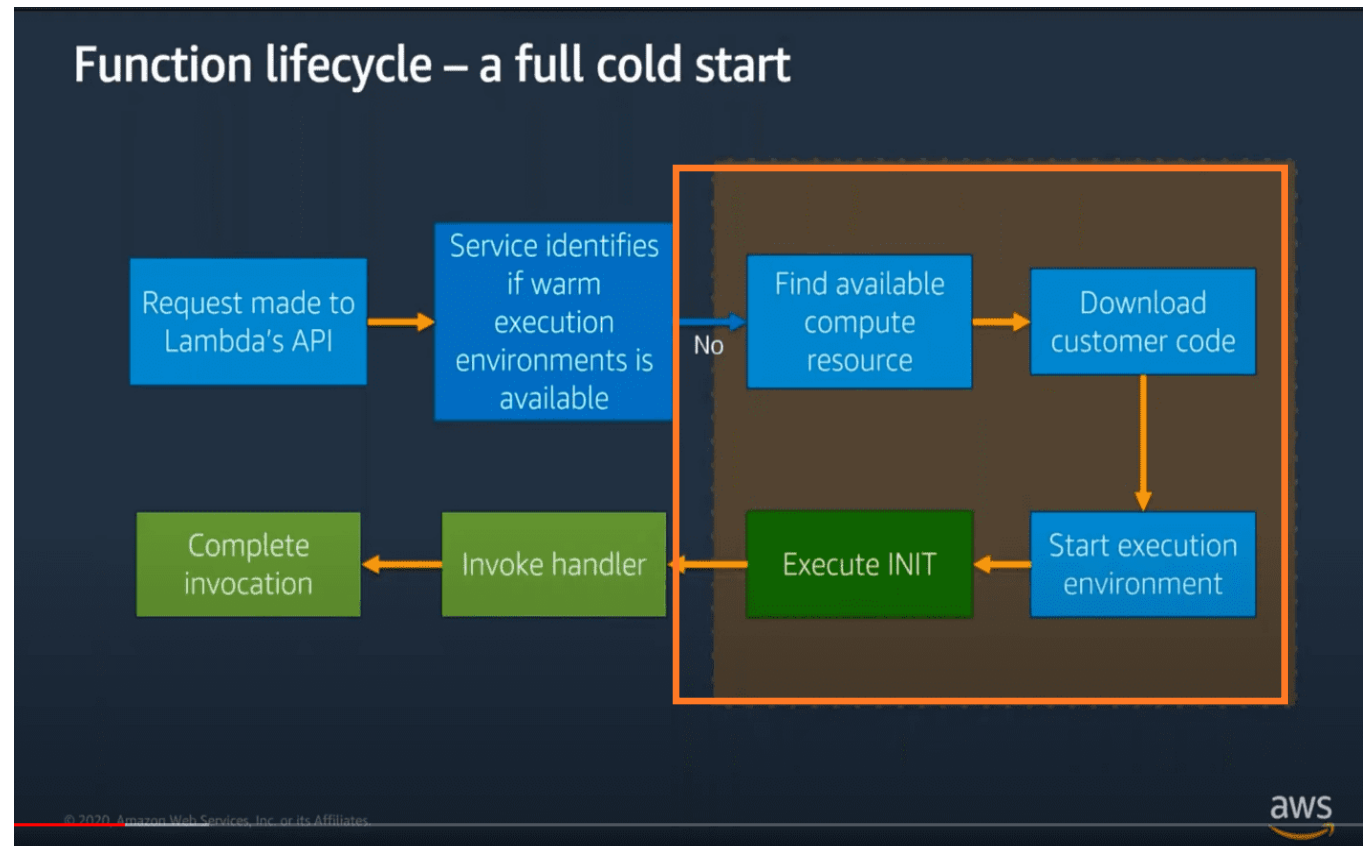
- “cold start” times (latencies)
- memory footprint (high cost in AWS)



Challenge No. 1

A Big **Cold-Start**

Lambda function lifecycle – a full cold start

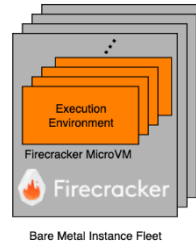


Sources: Ajay Nair „Become a Serverless Black Belt” <https://www.youtube.com/watch?v=oQFORsso2go>

Tomasz Łakomy "Notes from Optimizing Lambda Performance for Your Serverless Applications" <https://tlakomy.com/optimizing-lambda-performance-for-serverless-applications>

Lambda function lifecycle

- Start Firecracker VM (execution environment)
- AWS Lambda starts the Java runtime
- Java runtime loads and initializes Lambda function code (Lambda handler Java class)
 - Class loading
 - Static initializer block of the handler class is executed (i.e. AWS service client creation)
 - Runtime dependency injection
 - Just-in-Time (JIT) compilation
 - Init-phase has **full CPU access up to 10 seconds for free** *for the managed execution environments*
- Lambda invokes the handler method



Function lifecycle – worker host



Basic settings

Description

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

128 MB

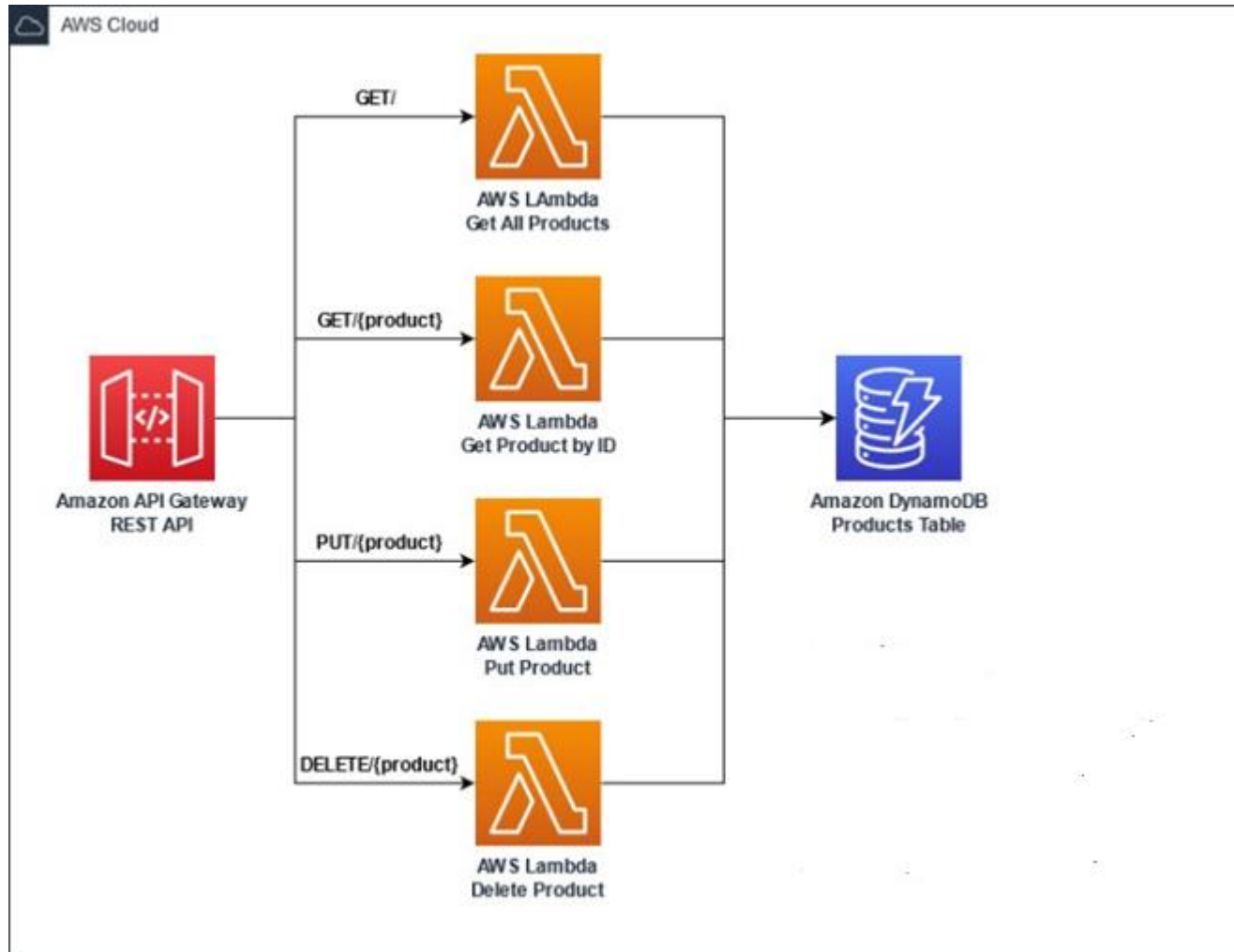
Timeout Info
0 min 3 sec

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>



Demo Application



- Lambda has 1024 MB memory setting
 - Lambda uses x86 architecture
 - Default (Apache) Http Client for communication with DynamoDB
 - 18 MB artifact size, , all dependencies in the POM file
 - Java compilation option
XX:+TieredCompilation
XX:TieredStopAtLevel=1
 - Info about the experiments:
 - Approx. 1 hour duration
 - Approx. **first*** 100 cold starts
 - Approx. first 100.000 warm starts
- *after Lambda function being re-deployed



Cold and warm starts with Java 21

Measurements in ms	p50	p75	p90	p99	p99.9	max
Amazon Corretto Java 21 cold start 	3158	3214	3270	3428	3601	3725
Amazon Corretto Java 21 warm start 	5,77	6,50	7,81	20,65	90,20	1423,63

Latency

Latency is the amount of time it takes from when a request is made by the user to the time it takes for the response to get back to that user

Latency

For our application:

Overall latency =

t(client to create and send request to the API Gateway endpoint) +

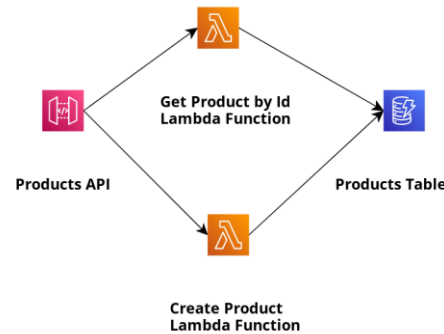
t(API Gateway to process request and invoke Lambda function) +

t(Lambda cold start time) +

t(Lambda warm start time) +

t(API Gateway to processes Lambda response and sends it back to the client) +

t(client to receive and process response from the API Gateway endpoint)



involves DNS resolve
and SSL handshake

Options To Reduce Cold Start Time

- General best practices
- AWS SnapStart
- GraalVM (Native Image)
- Provisioned concurrency



Best Practices & 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 Lambda handler method:
- `DynamoDbClient client = DynamoDbClientBuilder.builder()...build();`

Best Practices & Recommendations

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

- region, credential provider

```
DynamoDbClient client =  
DynamoDbClientBuilder.builder().region(Regions.US_WEST_2).build();
```

Best Practices & Recommendations

- **Prime** dependencies during initialization phase (when it worth doing)
 - „Fake“ the calls to pre-initialize „some other expensive stuff“ (this technique is called **Priming**)
 - In case of DynamoDB client put the following code outside of the handler method to pre-initialize the HTTP Client and Jackson Marshaller:

getProductById (int id)
method

DynamoDbClient client = DynamoDbClientBuilder.builder().region(Regions.US_WEST_2).build();

GetItemResponse getItemResponse = client.getItem(GetItemRequest.builder()

.key(Map.of("PK", AttributeValue.builder().s(id).build()))
.tableName(PRODUCT_TABLE_NAME).build());

.....



invocation forces HTTP Client and Jackson
Marshallers to pre-initialize

Word of caution



Re-measure it yourselves!

Cold and warm starts with Java 21

Measurements in ms	p50	p75	p90	p99	p99.9	max
Amazon Corretto Java 21 cold start 	3158	3214	3270	3428	3601	3725
Amazon Corretto Java 21 warm start 	5,77	6,50	7,81	20,65	90,20	1423,63

Effect of Priming

In case a priming is possible, but not done and cold start happens, the warm start/execution time takes longer

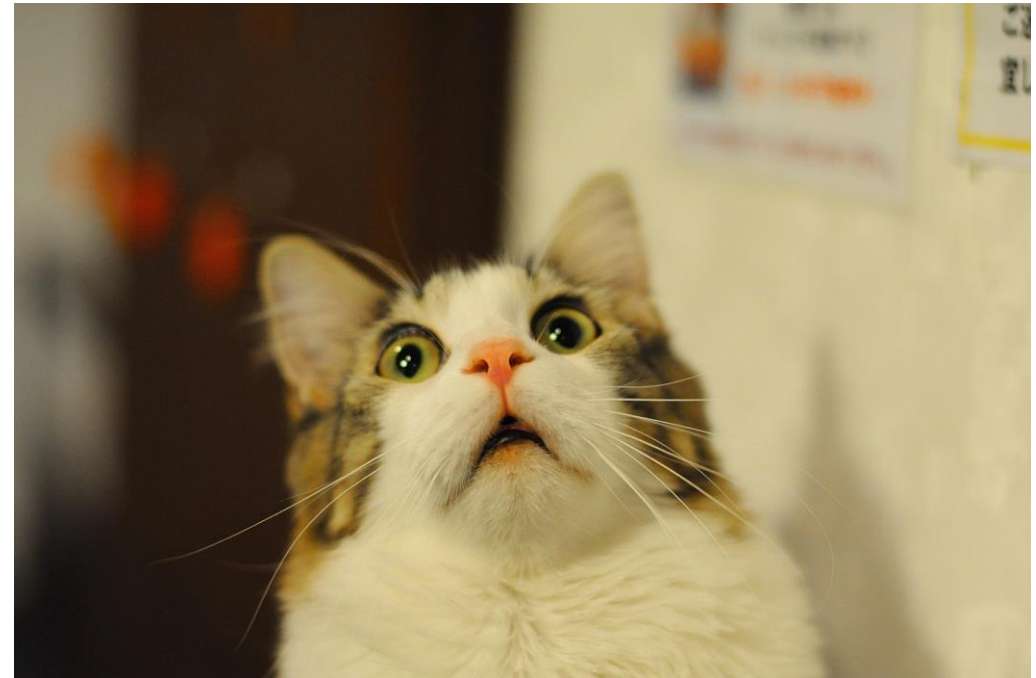


Best Practices & Recommendations

Avoid:

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

Use DI Frameworks which aren't reflection-based



AWS Lambda **SnapStart**

AWS Lambda SnapStart

AWS Lambda SnapStart \neq SnapChat



„AWS Lambda SnapStart „ series



Vadym Kazulkin

Posted on Dec 4, 2022 • Updated on Dec 5, 2023

Edit Manage Stats



13



1



1



1



1

AWS Lambda SnapStart - Part 1 Initial measuring of Java 11 Lambda cold starts

#java #serverless #aws #lambda

[AWSSnapStartWithJava \(17 Part Series\)](#)

Article series covers the why and what behind Lambda SnapStart and priming techniques including measurements for cold and warm starts with different settings for

- Java 11
- Java 17
- Java 21
- Micronaut
- Quarkus
- Spring Boot 2.7

able
serverless

AWS Lambda SnapStart

- Lambda SnapStart for Java can improve **startup performance** for latency-sensitive applications
- SnapStart is fully managed

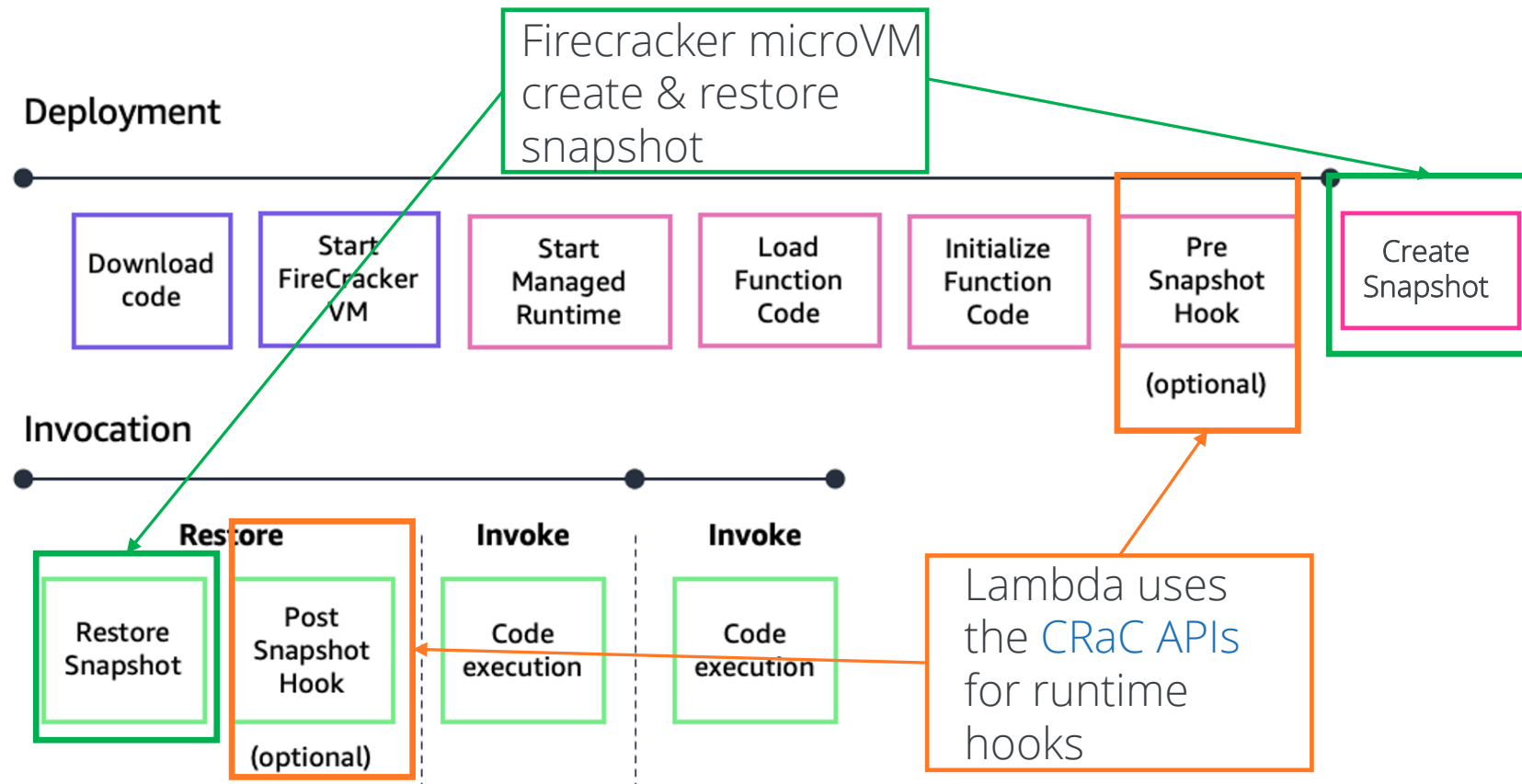
AWS SnapStart Deployment & Invocation

<div>Configuration Aliases Versions</div>		
<div>General configuration Info</div>		
Description	Memory	Ephemeral storage
-	1024 MB	512 MB
Timeout	<div>SnapStart Info PublishedVersions</div>	
0 min 30 sec		

AWS Lambda SnapStart

- Currently available for Lambda managed Java Runtimes (Java 11, 17 and 21)
- Not available for all other Lambda runtimes:
 - Docker Container Image
 - Custom (Lambda) Runtime (a way to ship GraalVM Native Image)

AWS SnapStart Deployment & Invocation






AWS SnapStart Deployment Phase

CloudWatch > Log groups > /aws/lambda/GetProductByIdWithPureLambda > 2023/07/02/[26]674e28a9040544cb8ffebe2e94116252

Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. [Learn more about filter patterns](#)


 **Actions** ▼ **Start tailing** **Create metric filter**




Clear **1m** **30m** **1h** **12h** **Custom**  **Display** ▼ 

▶	Timestamp	Message
No older events at this moment. Retry		
▼	2023-07-02T08:24:11.190+02:00	INIT_START Runtime Version: java:11.v21 Runtime Version ARN: arn:aws:lam... INIT_START Runtime Version: java:11.v21 Runtime Version ARN: arn:aws:lambda:eu-central-1::runtime:156ab0dc268a6b4a8dedcbcf0974795cafba2ee8760fe386062fffd887b971 Copy
▼	2023-07-02T08:24:11.220+02:00	Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLeve... Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1 Copy

AWS SnapStart Invocation Phase

Log events
You can use the filter bar below to search for and match terms, phrases, or values in your log events. [Learn more about filter patterns](#)

 **Actions** ▼ **Start tailing** **Create metric filter**

 *Filter events* **Clear** **1m** **30m** **1h** **12h** **Custom**  **Local** ▼ **Display** ▼ 

▶	Timestamp	Message
No older events at this moment. Retry		
▼	2023-09-29T19:55:33.821+02:00	RESTORE_START Runtime Version: java:11.v26 Runtime Version ARN: arn:aws:lambda:eu-central-1::runtime:a25427851bf301432a3...
		RESTORE_START Runtime Version: java:11.v26 Runtime Version ARN: arn:aws:lambda:eu-central-1::runtime:a25427851bf301432a3de7958f23cffac7c9db92c85f4611028de86b2b8ff30d Copy
▼	2023-09-29T19:55:34.510+02:00	RESTORE_REPORT Restore Duration: 689.04 ms
		RESTORE_REPORT Restore Duration: 689.04 ms Copy

CRIU (Checkpoint/Restore in Userspace)

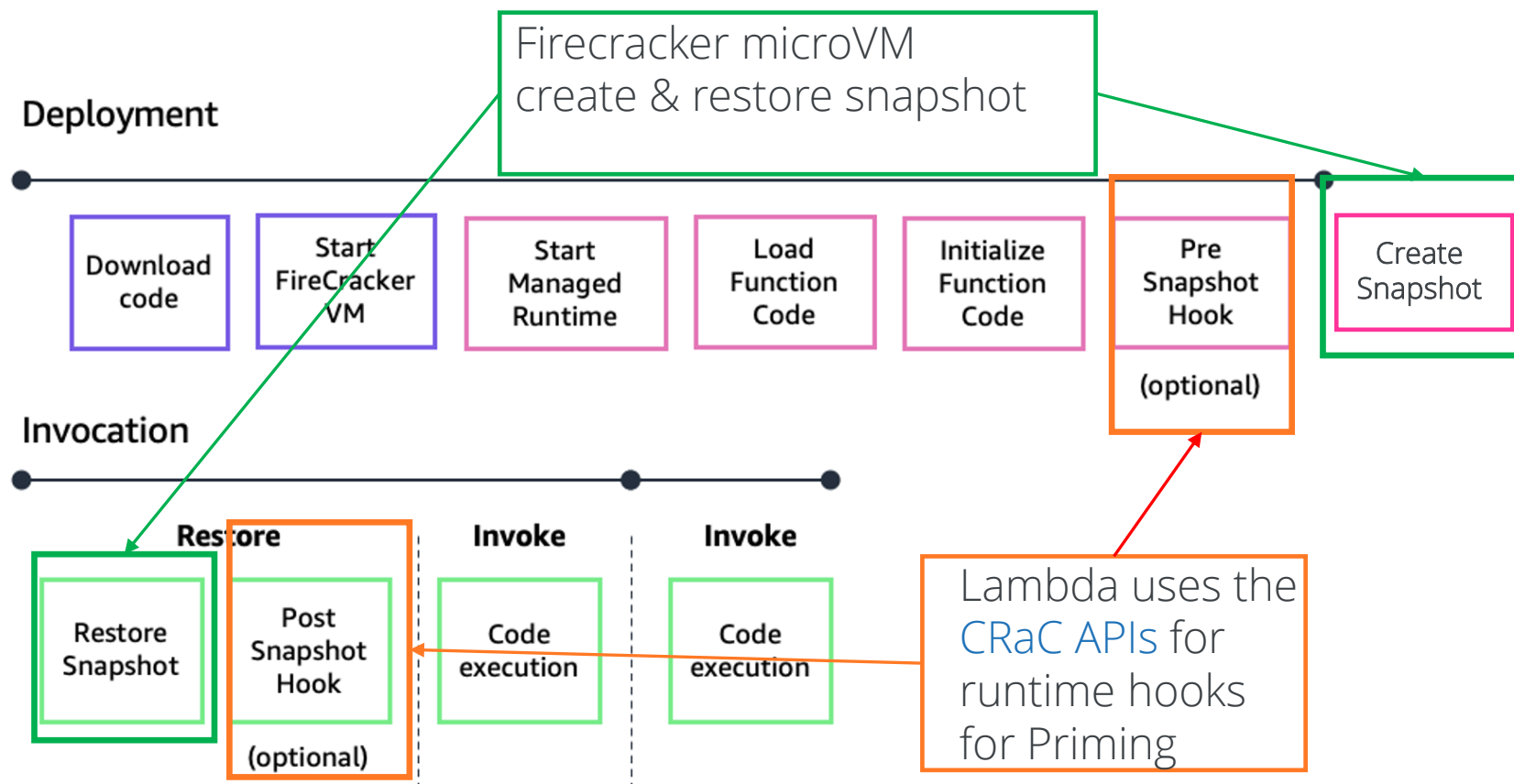
- Linux CRIU available since 2012 allows a running application to be paused and restarted at some point later in time, potentially on a different machine.
- The overall goal of the project is to support the migration of containers.
- When performing a checkpoint, essentially, the full context of the 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

Snapshot /checkpointing and restore

Firecracker microVM vs CRIU

- Comparing Firecracker and CRIU snapshotting:
 - Firecracker snapshotting saves a whole running OS
 - CRIU snapshotting saves a single process or container
- **Advantages** of the Firecracker snapshot : we don't have to care about file handles because they will be still valid after resume.
- **Drawbacks** the Firecracker snapshot : the need to reseed /dev/random and to sync the system clock.

AWS SnapStart Deployment & Invocation



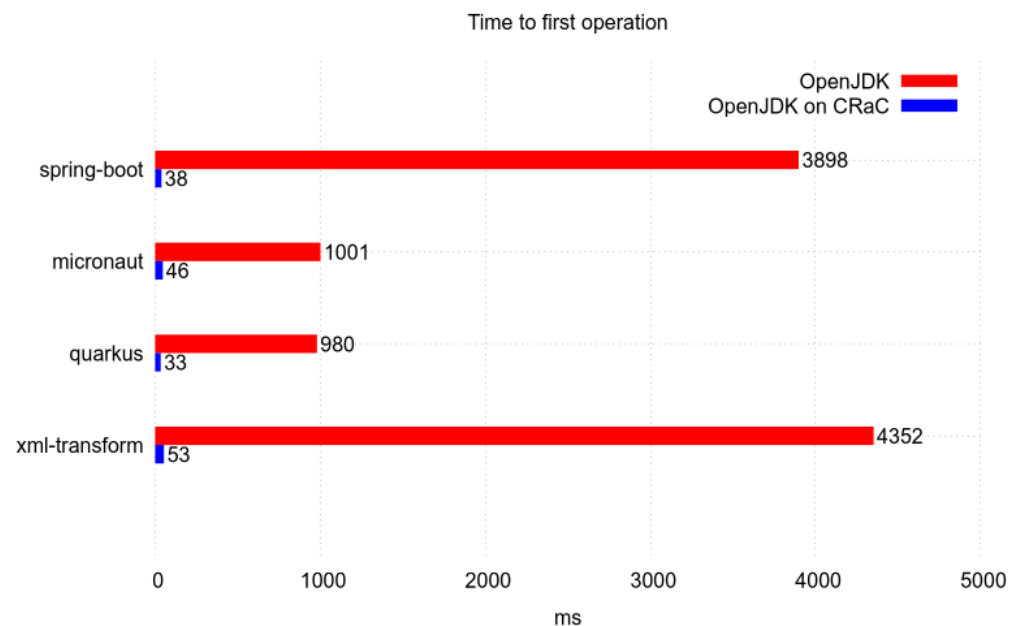
Ideas behind CRaC (Coordinated Restore at Checkpoint)

- CRaC is a JDK project that allows you to start Java programs with a shorter time to first transaction, combined with less time and resources to achieve full code speed.
- CRaC effectively takes a snapshot of the Java process (checkpoint) when it is fully warmed up, then uses that snapshot to launch any number of JVMs from this captured state.
- CRaC is based on CRIU

CRaC (Coordinated Restore at Checkpoint)

CRaC (Coordinated Restore at Checkpoint) Project

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



AWS SnapStart Enabled with 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);
    }
}
```

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


if a refer to Priming in this talk, I mean concretely
DynamoDB request
invocation priming






Lambda SnapStart Priming Guide



 **snapstart-priming-guide** Public Watch 2

main

 1 Branch 0 Tags Add file Code

 **marksailes** Full code example c9eda19 · 2 weeks ago 21 Commits

 examples/emf	Full code example	2 weeks ago
 images	diagrams to show the different phases between on-demand ...	last month
 .gitignore	Full code example	2 weeks ago
 LICENSE	Initial commit	now
 README.md	Full code example	2 weeks ago

 **README**  MIT license edit menu

Lambda SnapStart Priming Guide

Goal

This guide aims to explain techniques for priming Java applications. It assumes a base understanding of AWS Lambda, Lambda SnapStart, and CRaC.

guide aims to explain techniques for priming Java applications.

It assumes a base understanding of AWS Lambda, Lambda SnapStart, and CRaC.

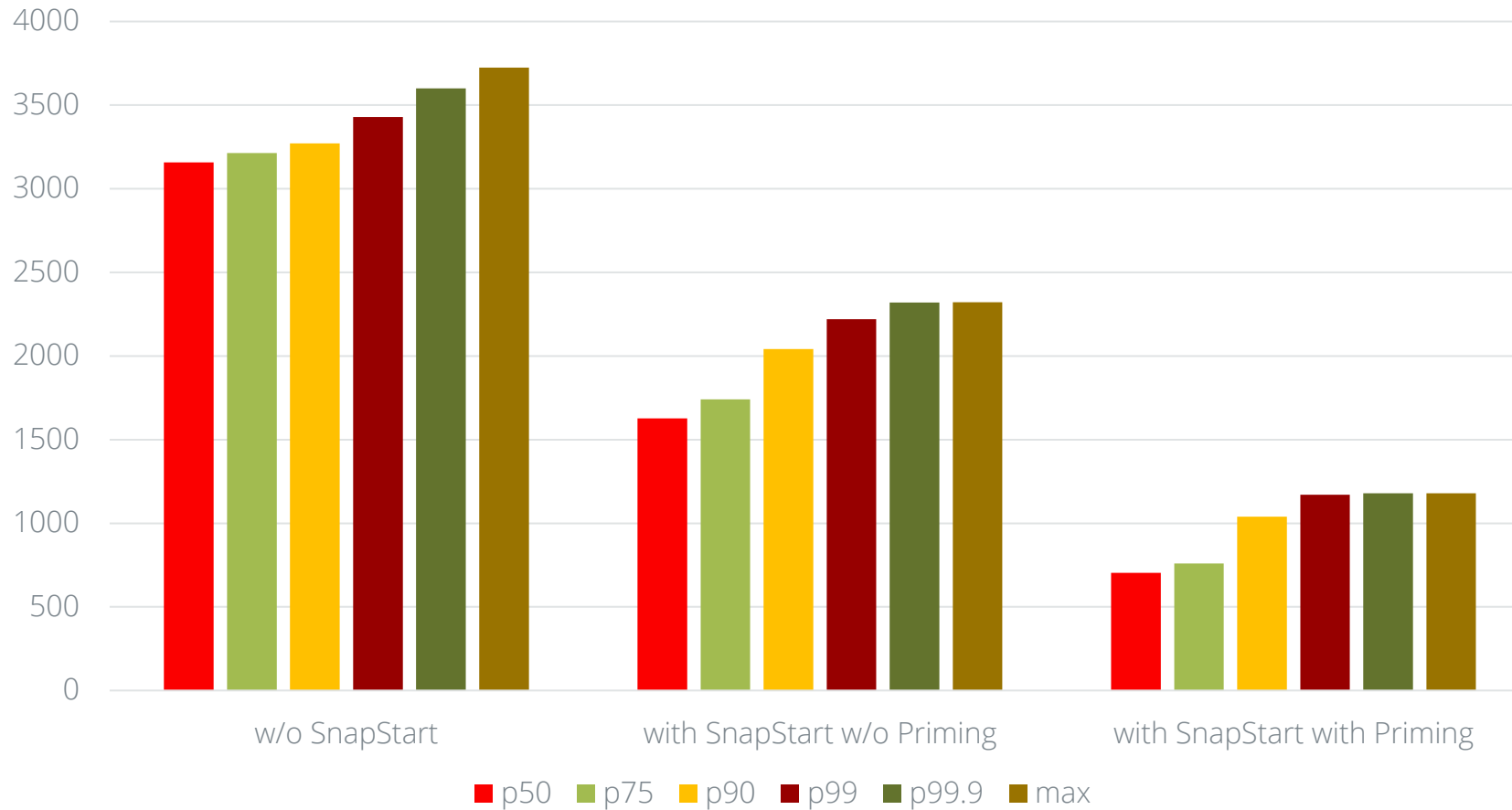
AWS SnapStart Challenges & Limitations

- SnapStart supports the Java 11, 17 and 21 (Corretto) managed runtime only
- Deployment with SnapStart enabled takes more than 2-2,5 minutes additionally
- Snapshot is deleted from cache if Lambda function is not invoked for 14 days
- SnapStart currently does not support :
 - Provisioned concurrency
 - ~~arm64 architecture (supports only x86).~~ Supports arm64 architecture since 18 July 2024
 - Amazon Elastic File System (Amazon EFS)
 - Ephemeral storage greater than 512 MB



ms

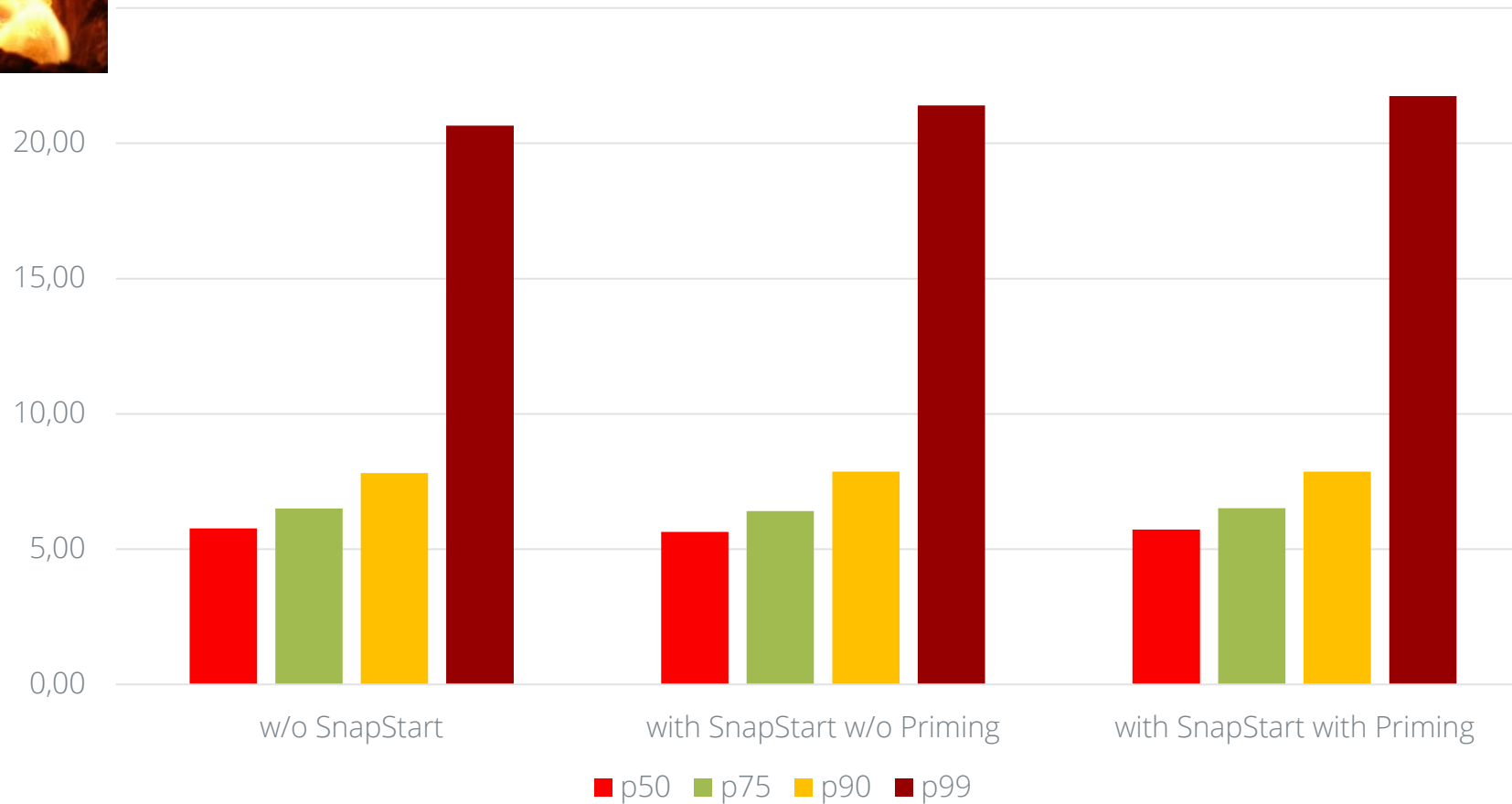
Cold starts of Lambda function with Java 21 runtime with
1024 MB memory setting, Apache Http Client, compilation
-XX:+TieredCompilation -XX:TieredStopAtLevel=1





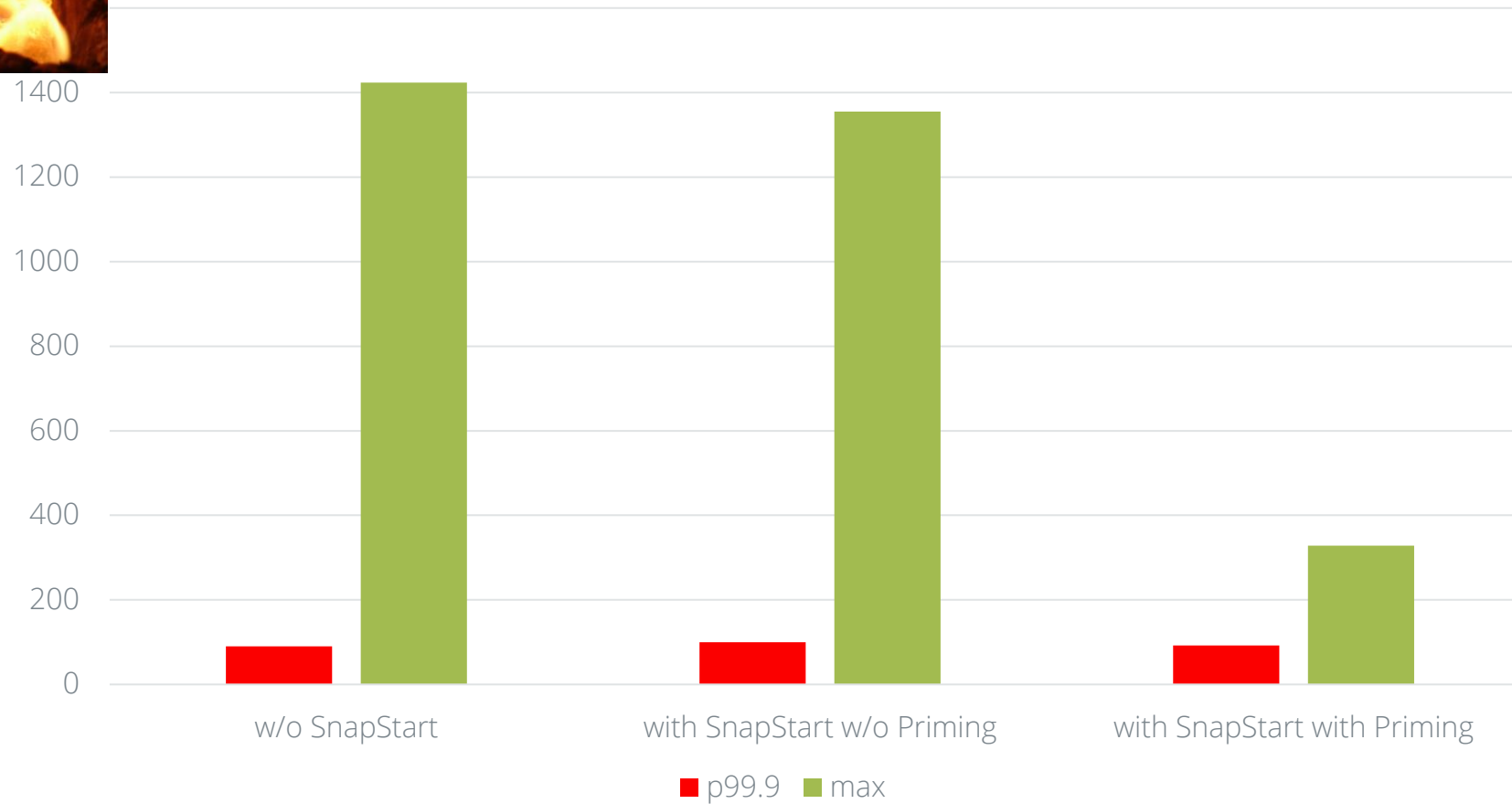
ms

Warm starts of Lambda function with Java 21 runtime with 1024 MB memory setting, Apache Http Client compilation - XX:+TieredCompilation -XX:TieredStopAtLevel=1

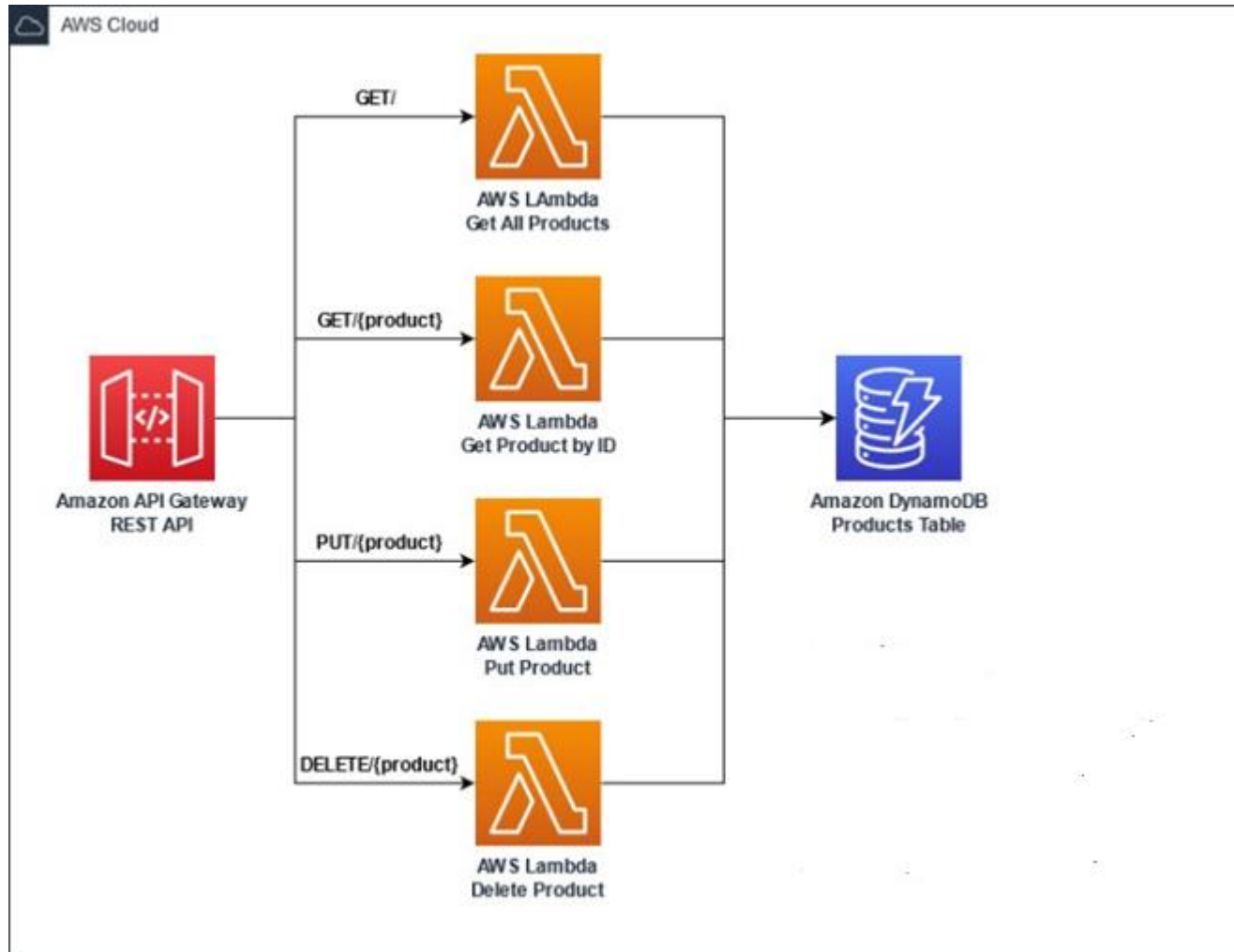


**ms**

Warm starts of Lambda function with Java 21 runtime with 1024 MB memory setting, Apache Http Client compilation - XX:+TieredCompilation -XX:TieredStopAtLevel=1



Demo Application



- Lambda has 1024 MB memory setting
 - Lambda uses x86 architecture
 - Default (Apache) Http Client for communication with DynamoDB
 - 18 MB artifact size, , all dependencies in the POM file
 - Java compilation option
XX:+TieredCompilation
XX:TieredStopAtLevel=1
 - Info about the experiments:
 - Approx. 1 hour duration
 - Approx. **first*** 100 cold starts
 - Approx. first 100.000 warm starts
- *after Lambda function being re-deployed



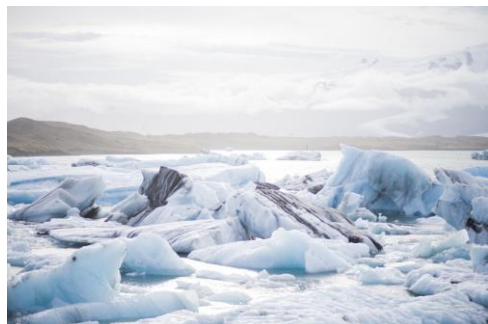
Lambda Memory Setting



ONE SECOND

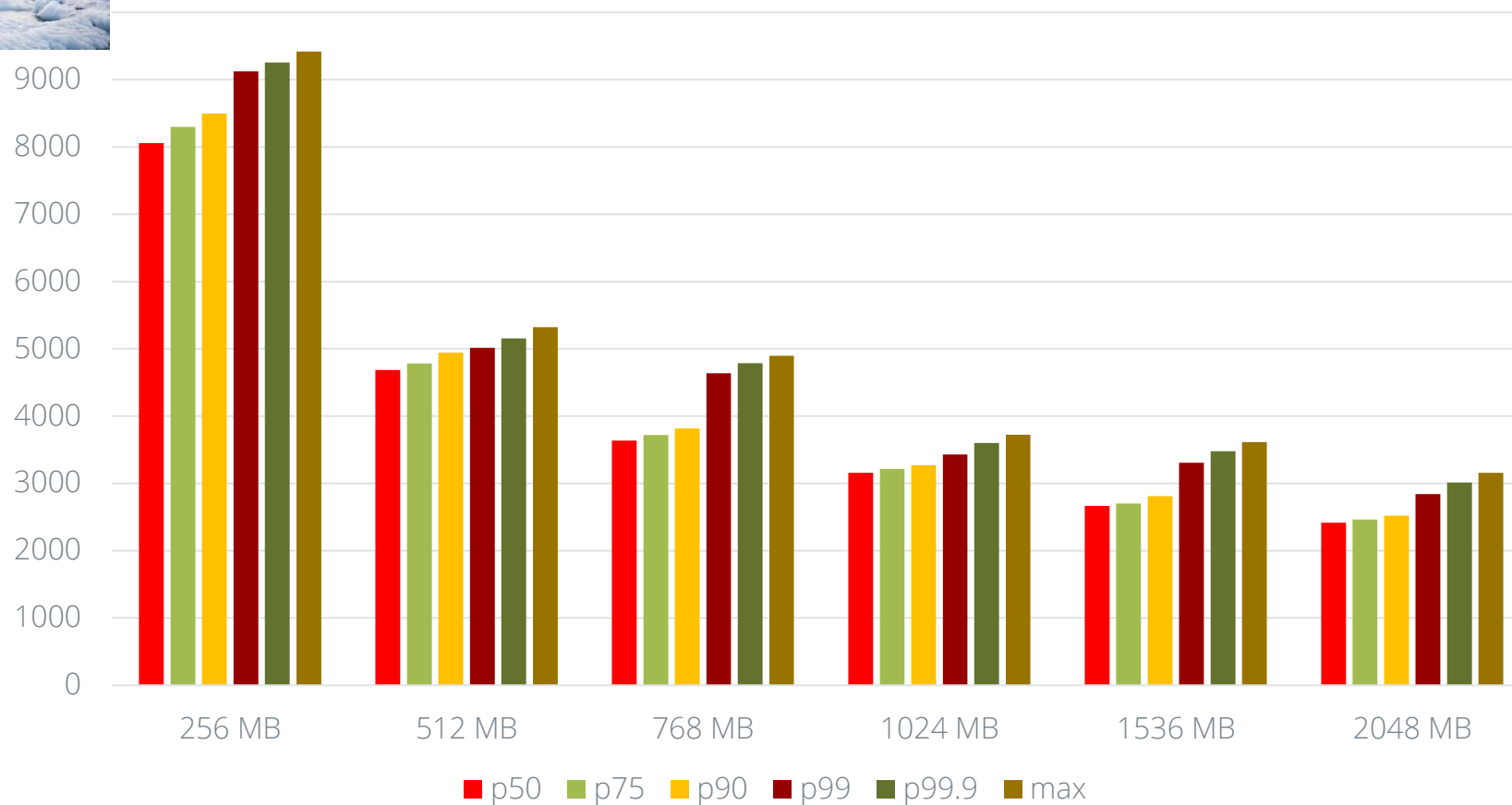


ONE GB



ms

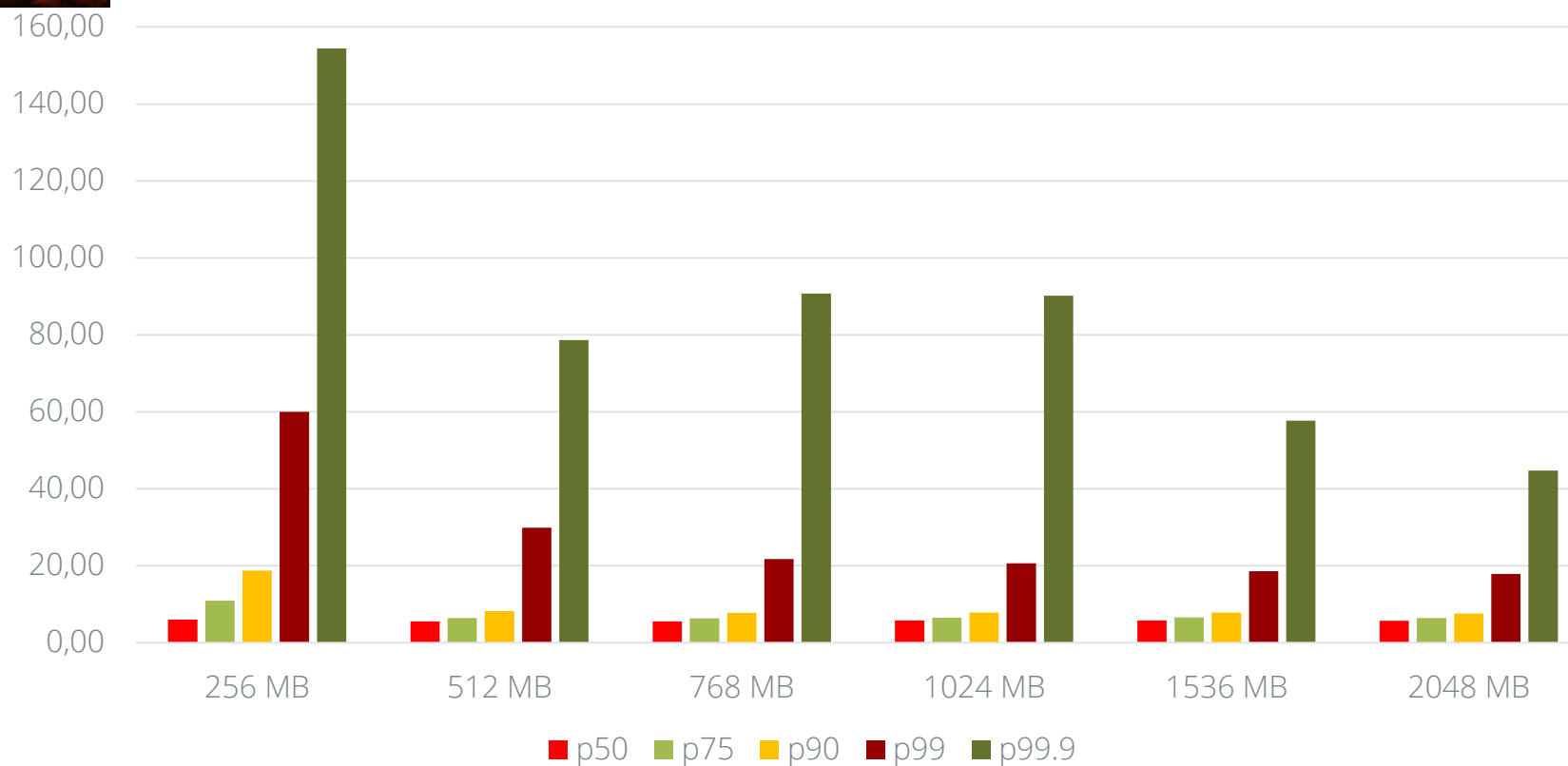
Cold starts of Lambda function with Java 21 runtime
without SnapStart for different memory settings





ms

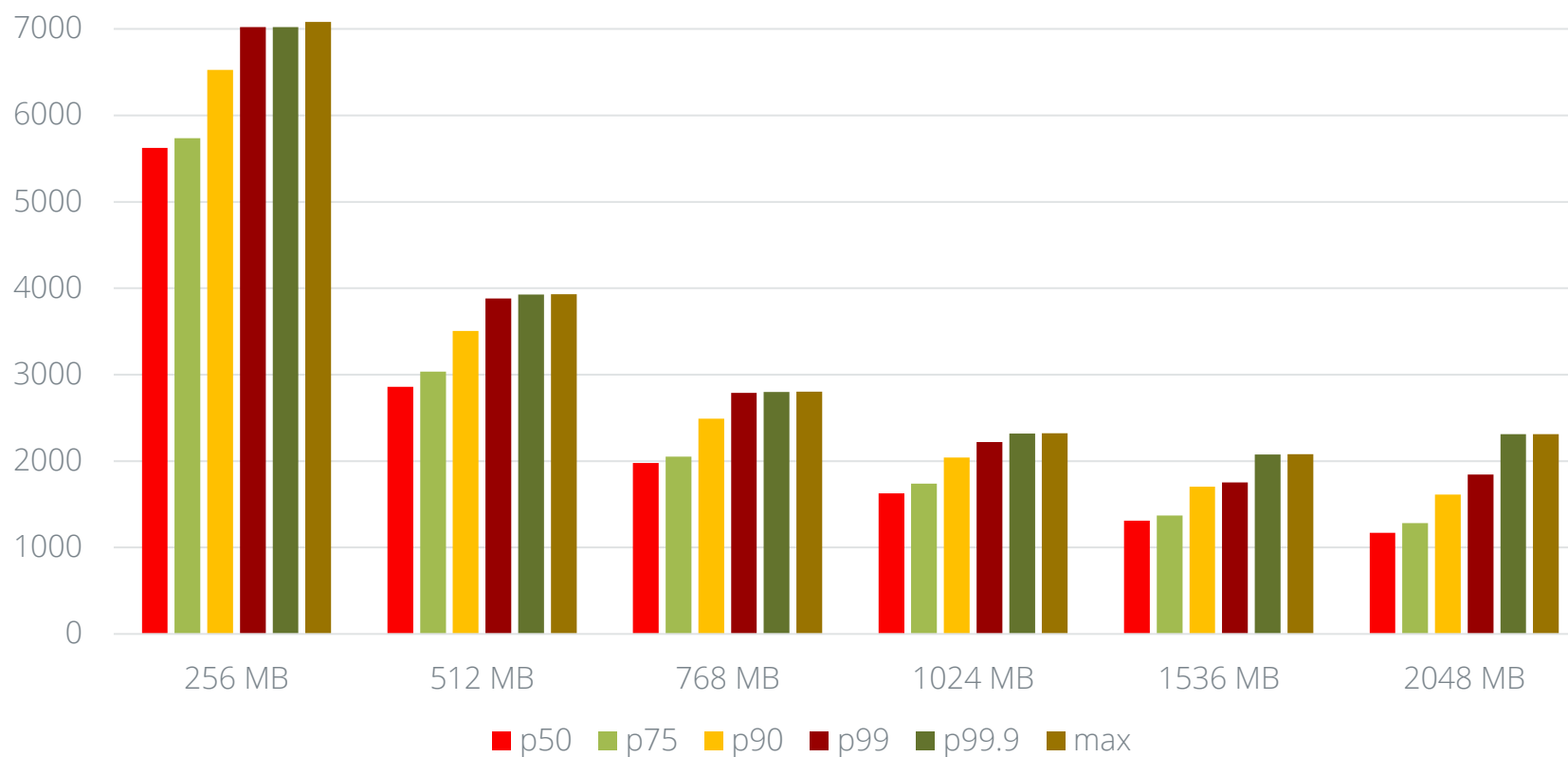
Warm starts of Lambda function with Java 21 runtime
without SnapStart for different memory settings





ms

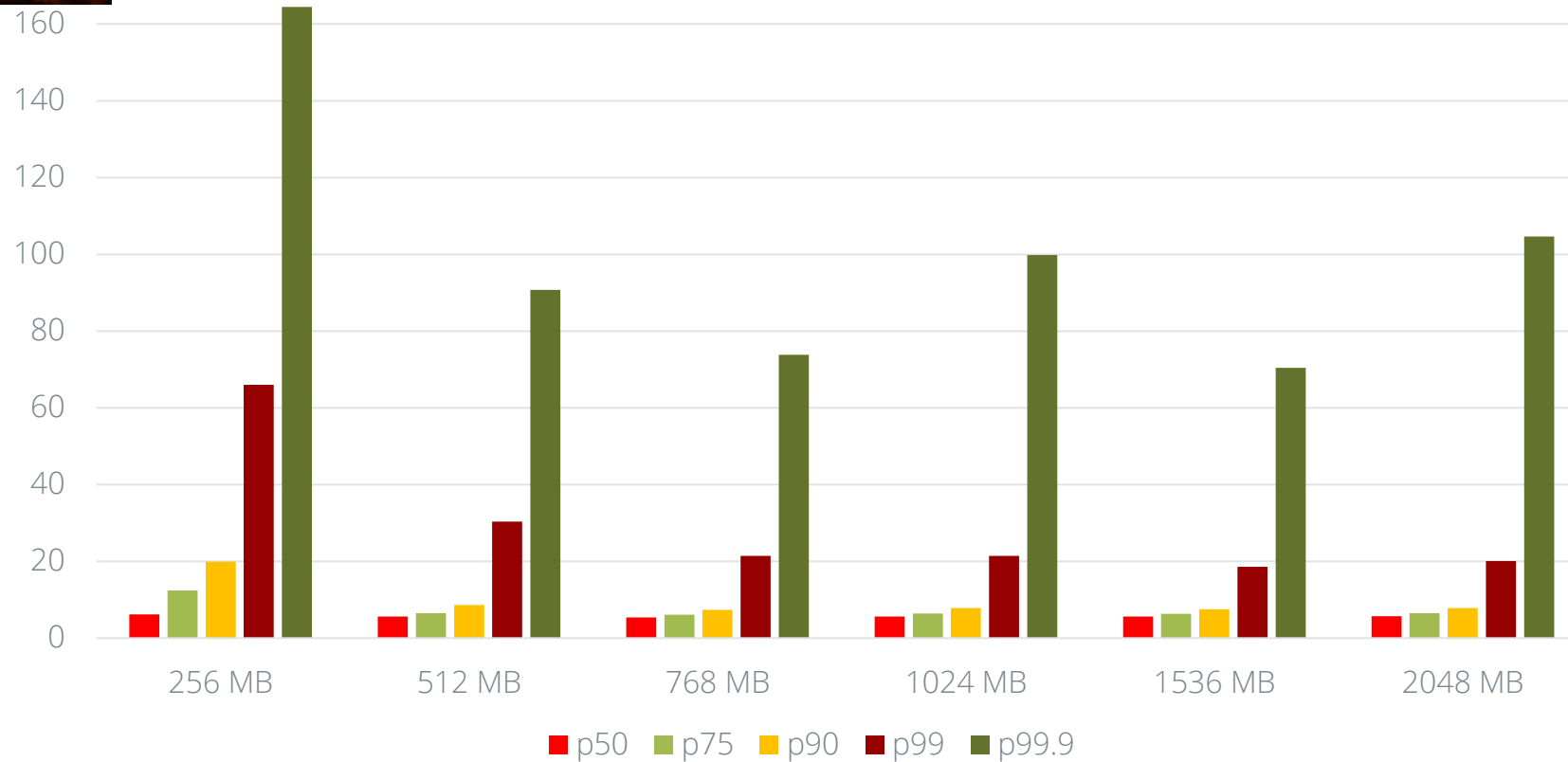
Cold starts of Lambda function with Java 21 runtime **with SnapStart without Priming** for different memory settings





ms

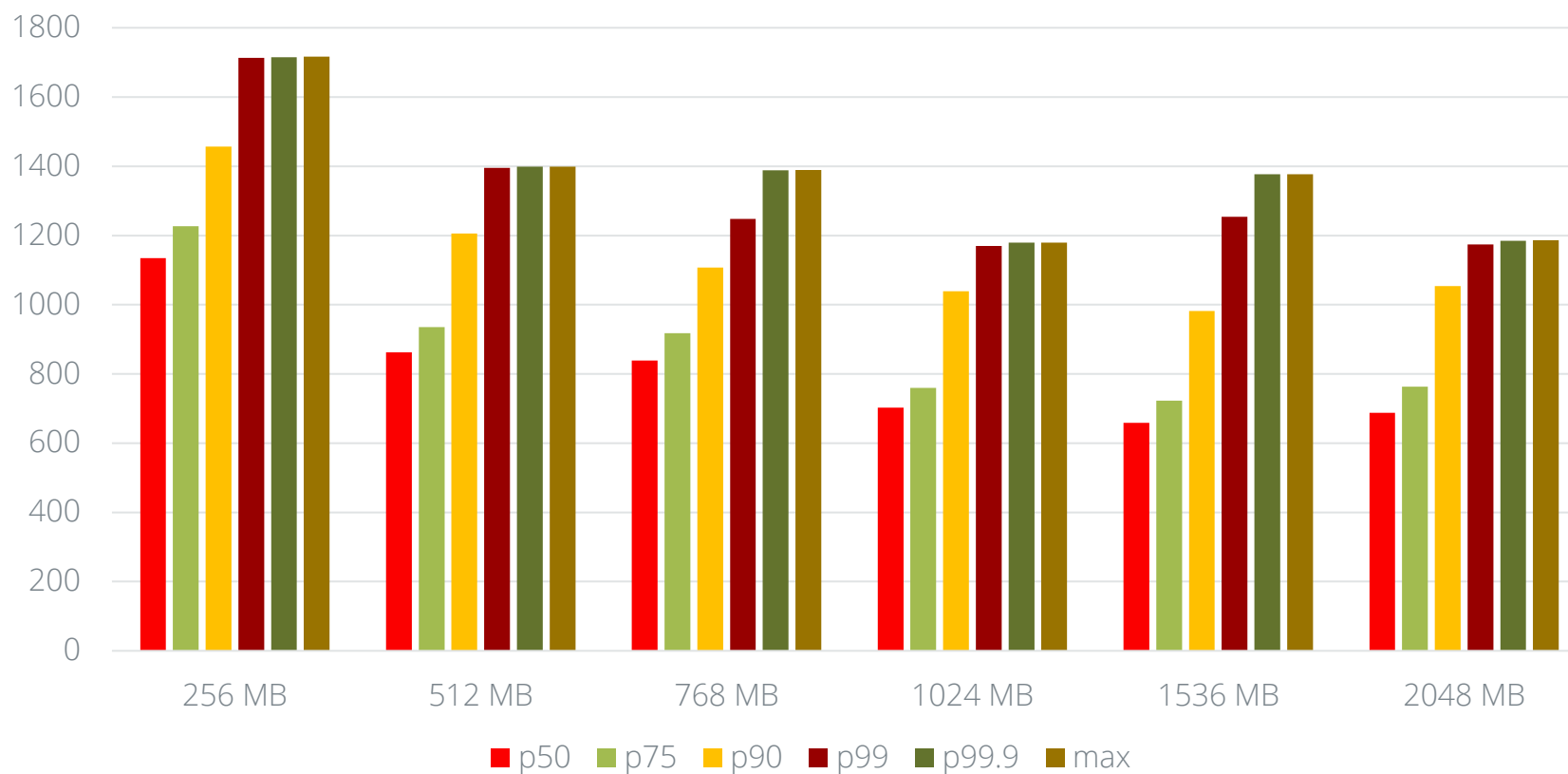
Warm starts of Lambda function with Java 21 runtime
with SnapStart without Priming for different memory settings





ms

Cold starts of Lambda function with Java 21 runtime **with SnapStart with Priming** for different memory settings



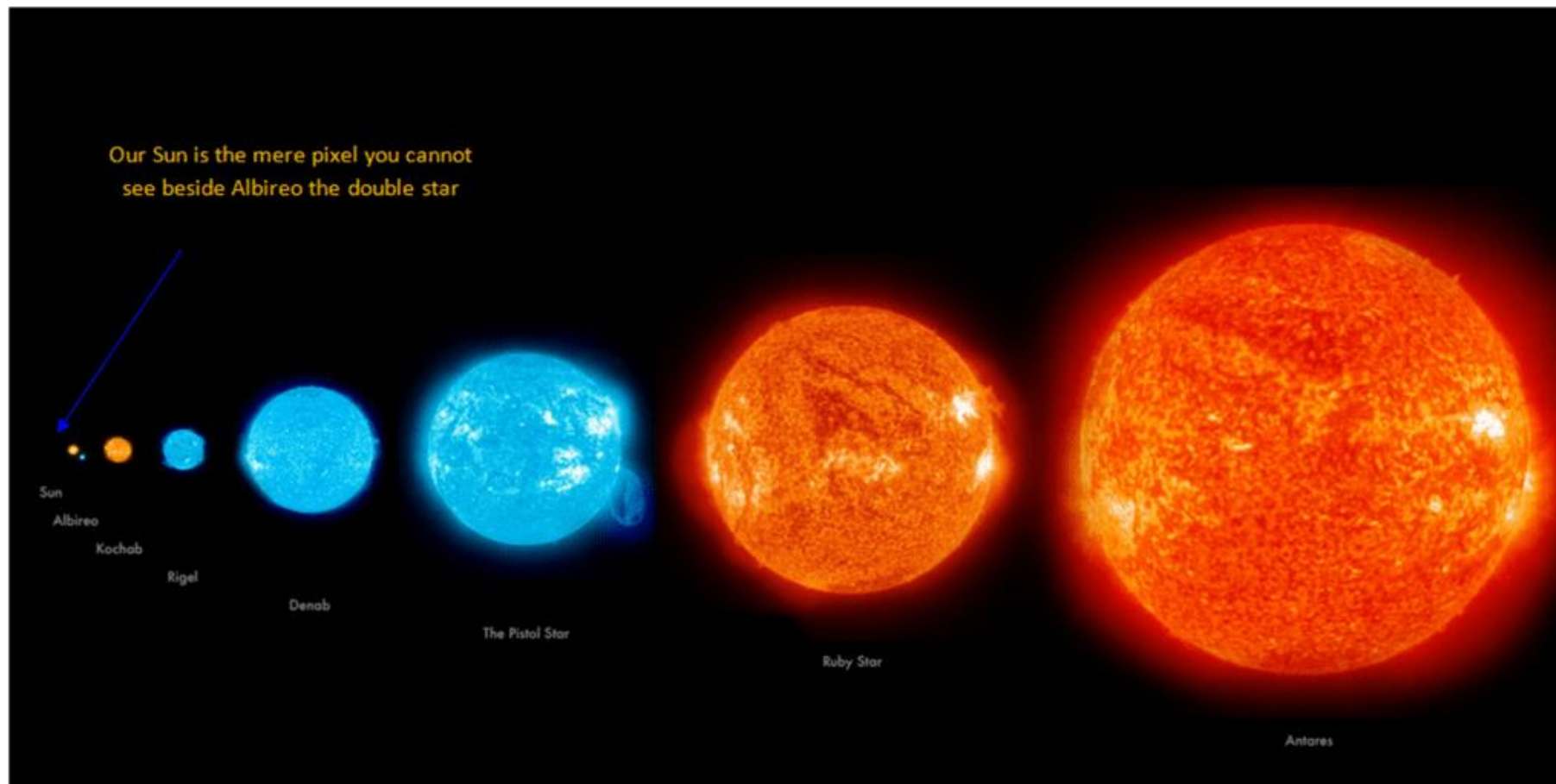


ms

Warm starts of Lambda function with Java 21 runtime
with SnapStart with Priming for different memory settings



Lambda Deployment Artifact Size

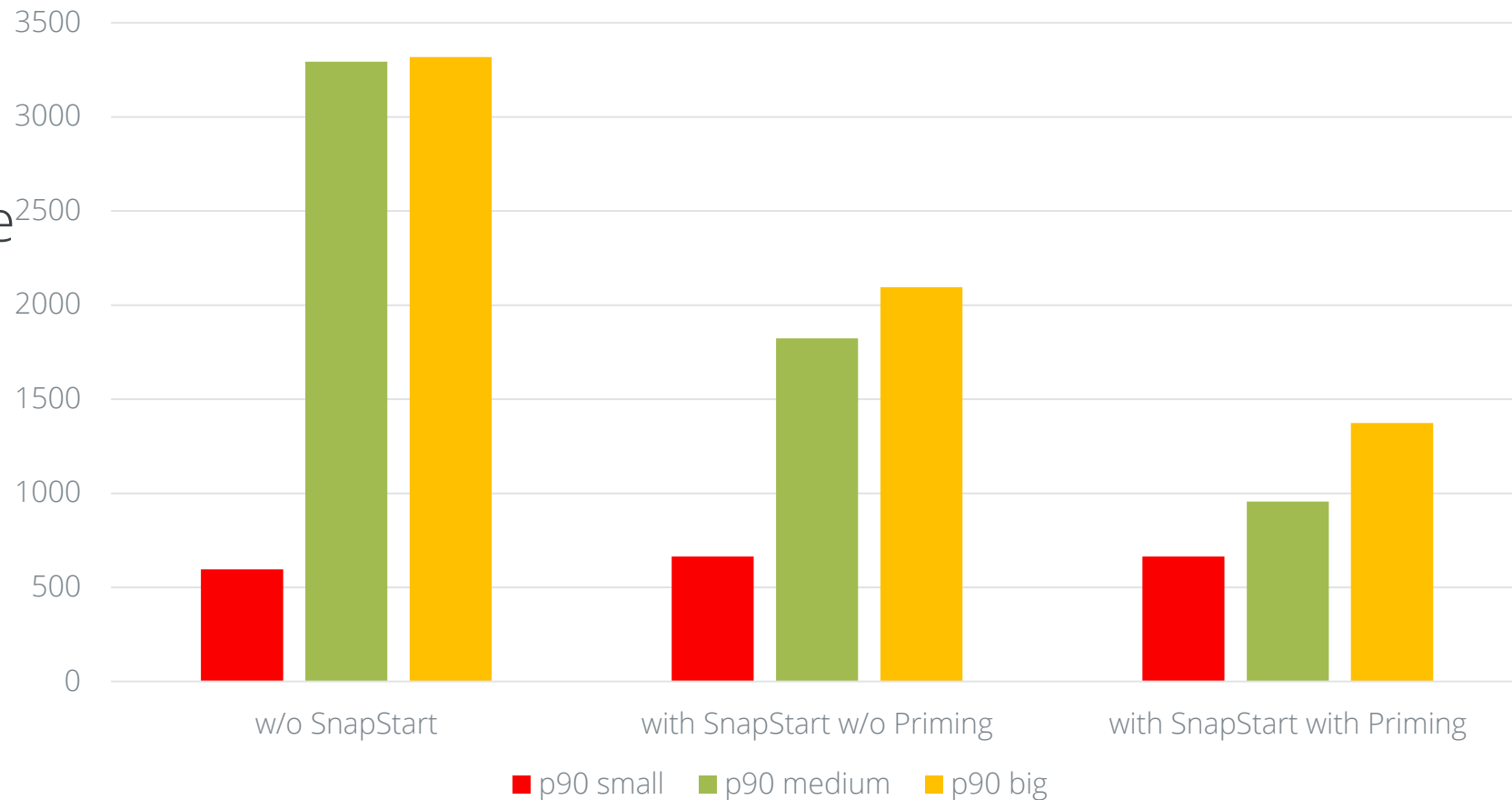




Cold starts of Lambda function with Java 21 runtime
using deployment artifact sizes for p90

ms

- Small -137 KB ("Hello World")
- Medium – 18 MB (our sample application)
- Big -50 MB (our sample application + additional dependencies other to AWS services)



Best Practices & 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 i.e. test frameworks like Junit

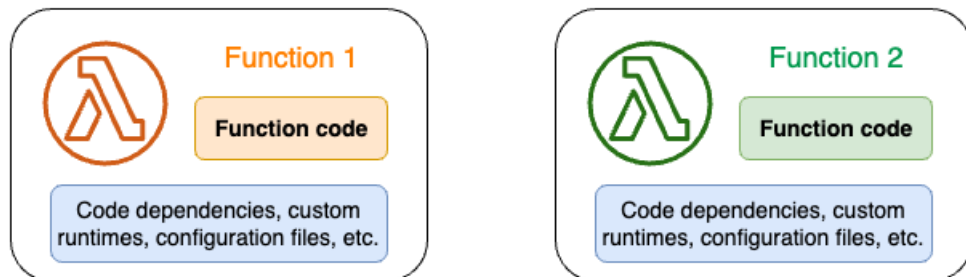
```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>bom</artifactId>
  <version>2.22.2</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>dynamodb</artifactId>
  <version>2.22.2</version>
</dependency>
```

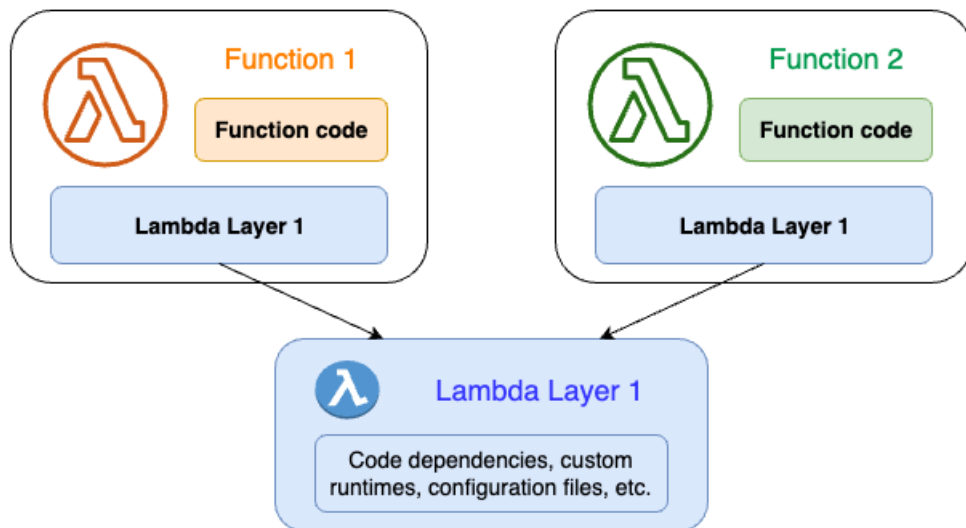
```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.4.2</version>
  <scope>test</scope>
</dependency>
```

Using Lambda Layers

Lambda function components: Without layers



Lambda function components: With layers



- aws-lambda-java-core
- aws-lambda-java-events
- org-crac
- slf4j-simple
- jackson-dataformat-xml

Type: AWS::Serverless::Function

Properties:

FunctionName: MyFunctionWithLayer

Layers:

- !Sub

arn:aws:lambda:\${AWS::Region}:\${AWS::AccountId}:layer:aws-pure-java-21-common-lambda-layer:1

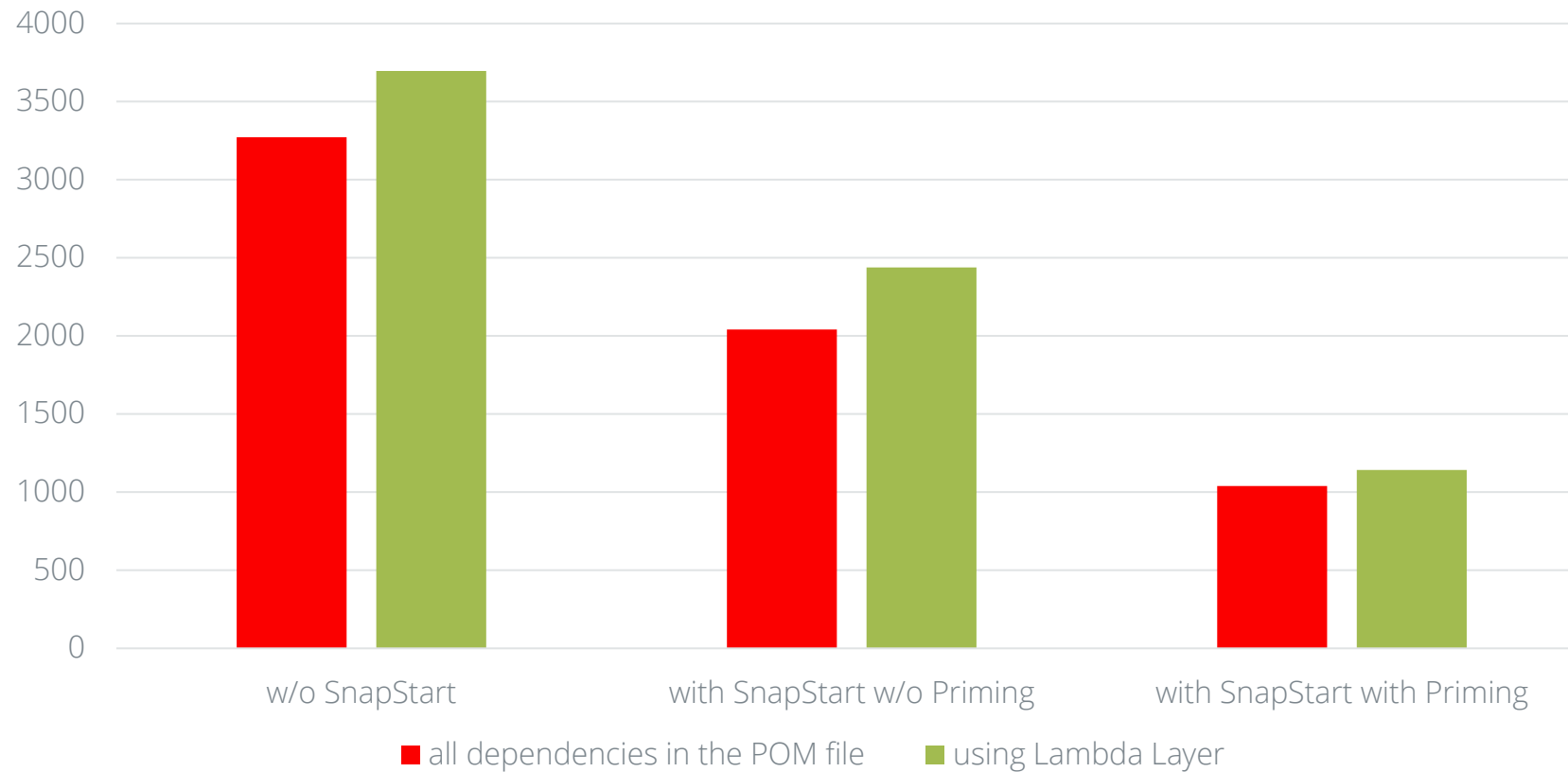
Handler:

software...handler.GetProductByIdHandler::handleRequest



ms

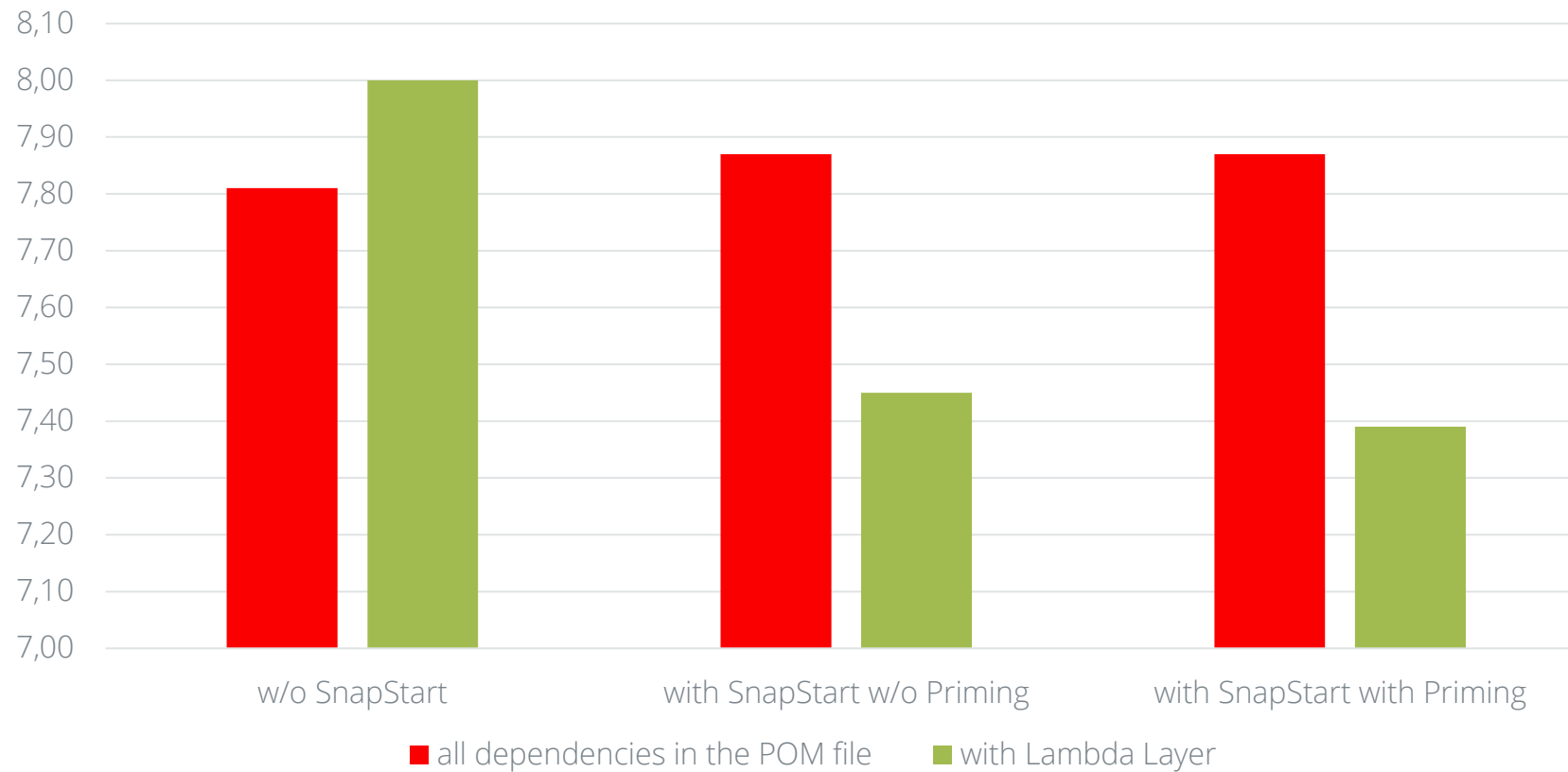
Cold starts of Lambda function with Java 21 runtime with
and without the usage of the Lambda Layer options for
p90



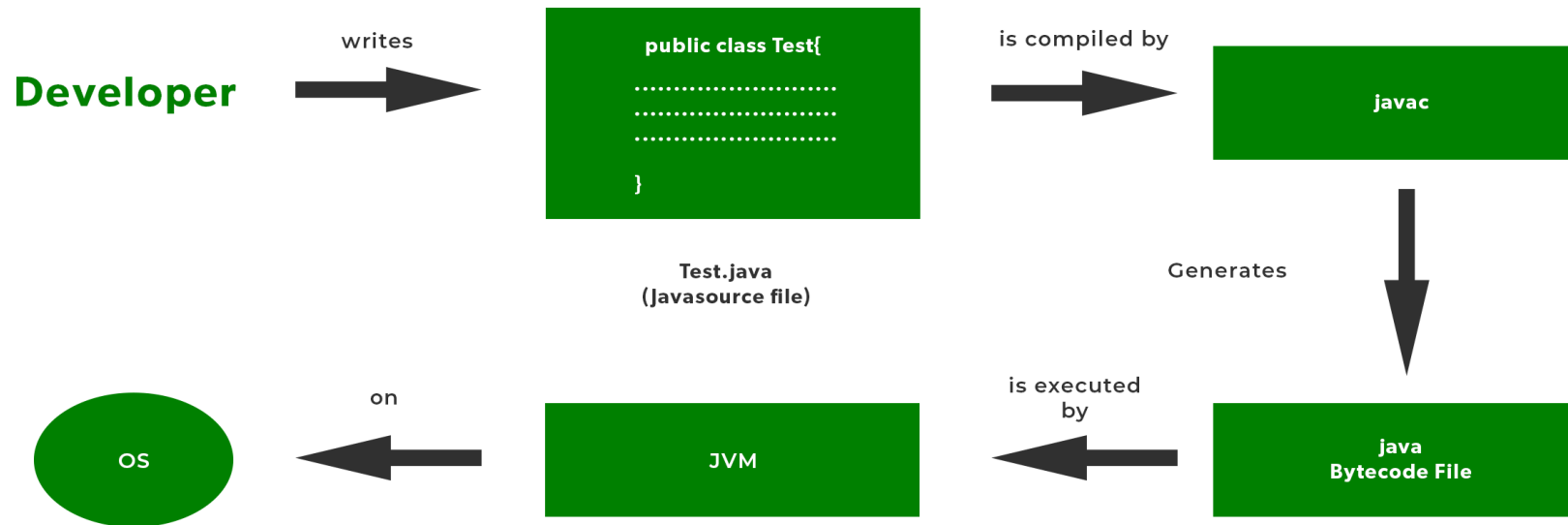


ms

Warm starts of Lambda function with Java 21 runtime
with and without the usage of the Lambda Layer options
for p90



Java Compilation



Step from java.code to machine code

- java source code is compiled.
- Transformed to bytecode by java compiler.
- Interpreted and executed by the java virtual machine on the underlying operating system.

Java Compilation Options

The image shows a composite of three screenshots from the AWS Lambda console illustrating the configuration of Java compilation options.

Top Screenshot (Configuration Tab): Shows the 'Configuration' tab for a Lambda function. The 'Environment variables' section is highlighted, showing 'Environment variables (0)' with an 'Edit' button.

Middle Screenshot (Add Environment Variable Dialog): A dialog box titled 'Choose Add environment variable. Add the following:' with a 'Bash' tab selected. It lists the following configuration:

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

Bottom Screenshot (Edit Environment Variables): Shows the 'Edit environment variables' page. It displays the configured environment variable:

Key	Value	Action
JAVA_TOOL_OPTIONS	-XX:+TieredCompilation -XX:TieredStop	Remove

Below the table is an 'Add environment variable' button and an 'Encryption configuration' section.

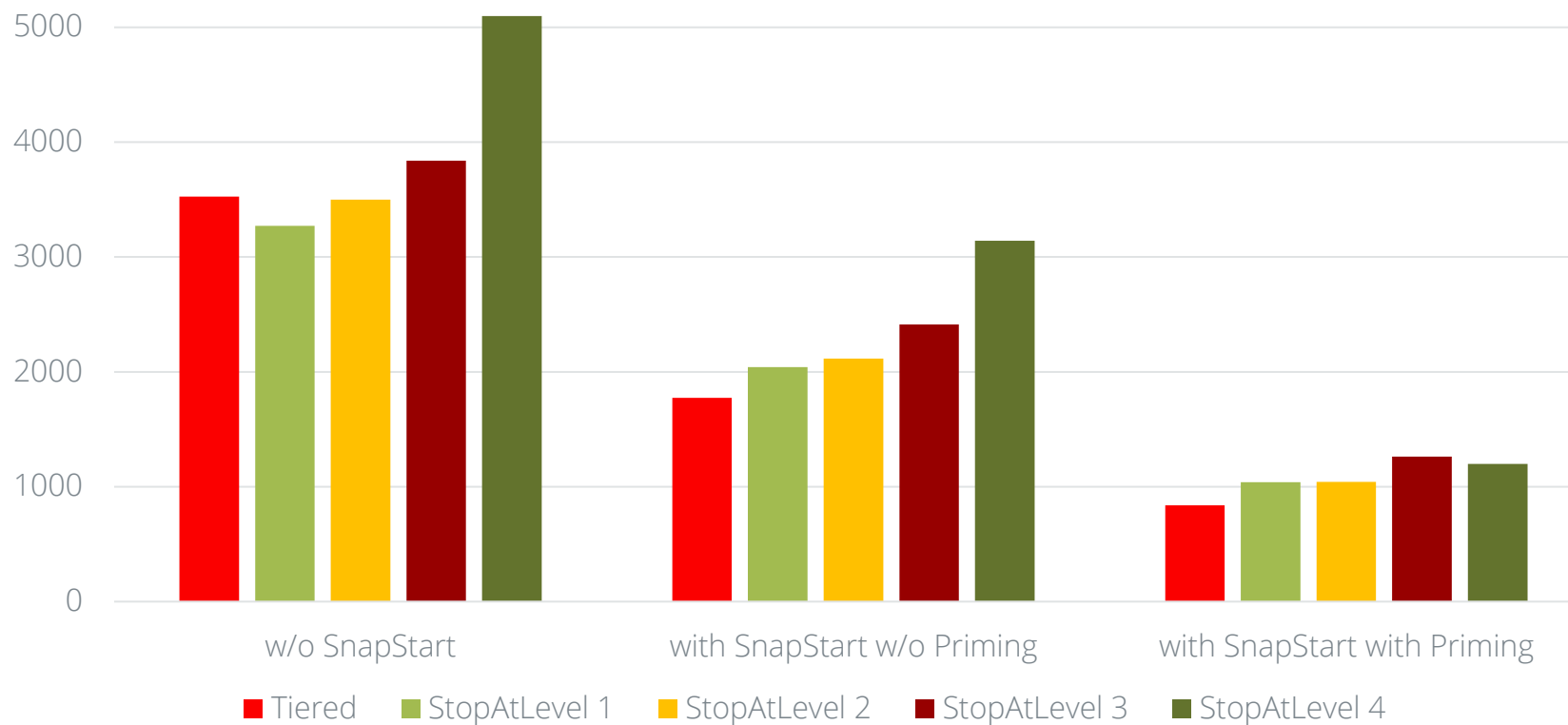
Right Screenshot (Tiered Compilation Levels): A vertical stack of five boxes representing the levels of Java compilation optimization:

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



ms

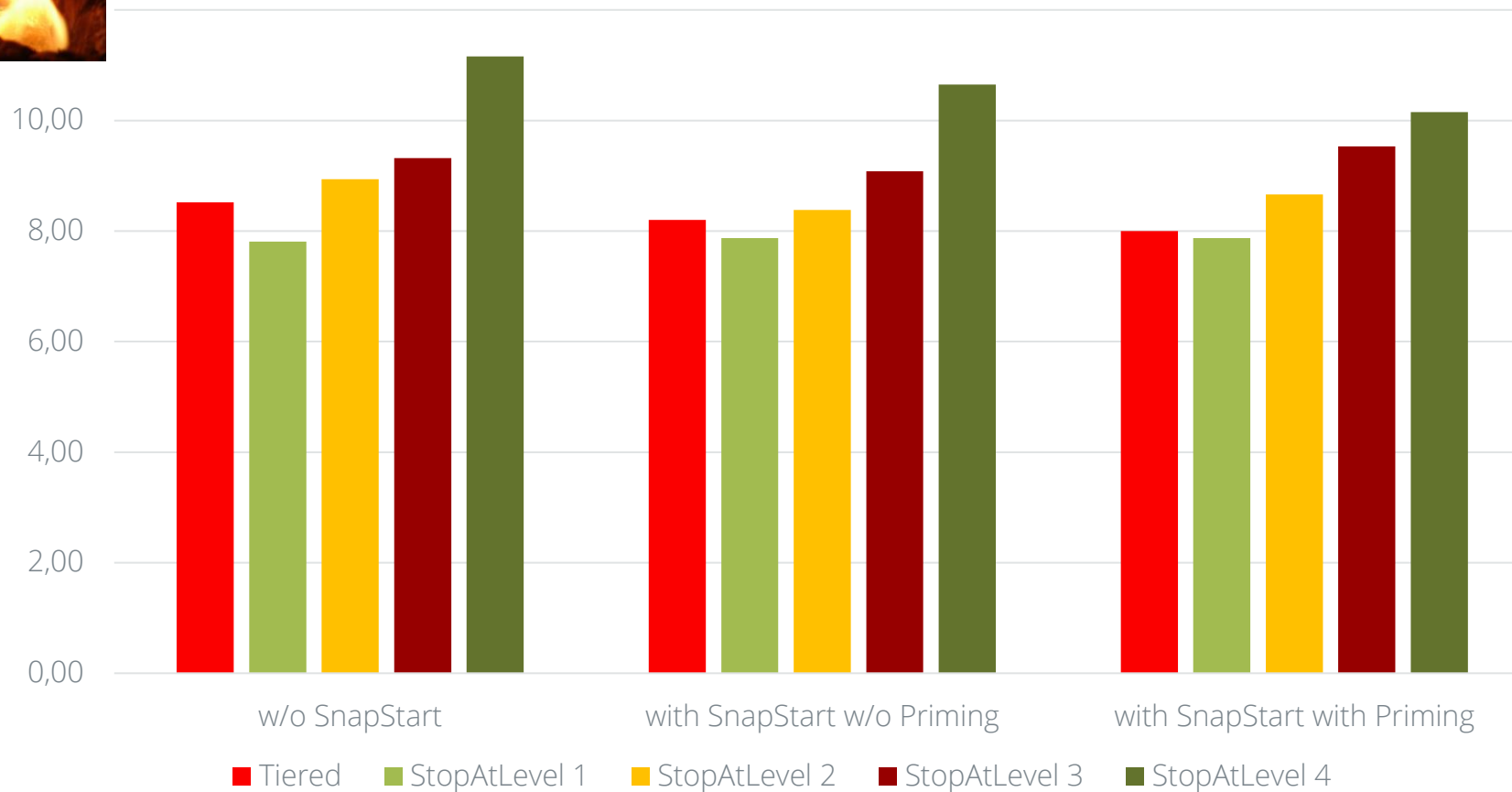
Cold starts of Lambda function with Java 21 runtime
using different compilation options for p90





ms

Warm starts of Lambda function with Java 21 runtime
using different compilation options for p90



Java Garbage Collection Algorithms



- Tested with default GC algorithms settings
- G1 default GC produces the best results for cold and warm starts without and with SnapStart (and priming) followed by :
- Parallel GC
- Shenandoah GC
- Couldn't make it work with ZGC (generational) GC

HTTP Clients to connect to DynamoDB



Setting synchronous DynamoDB Client

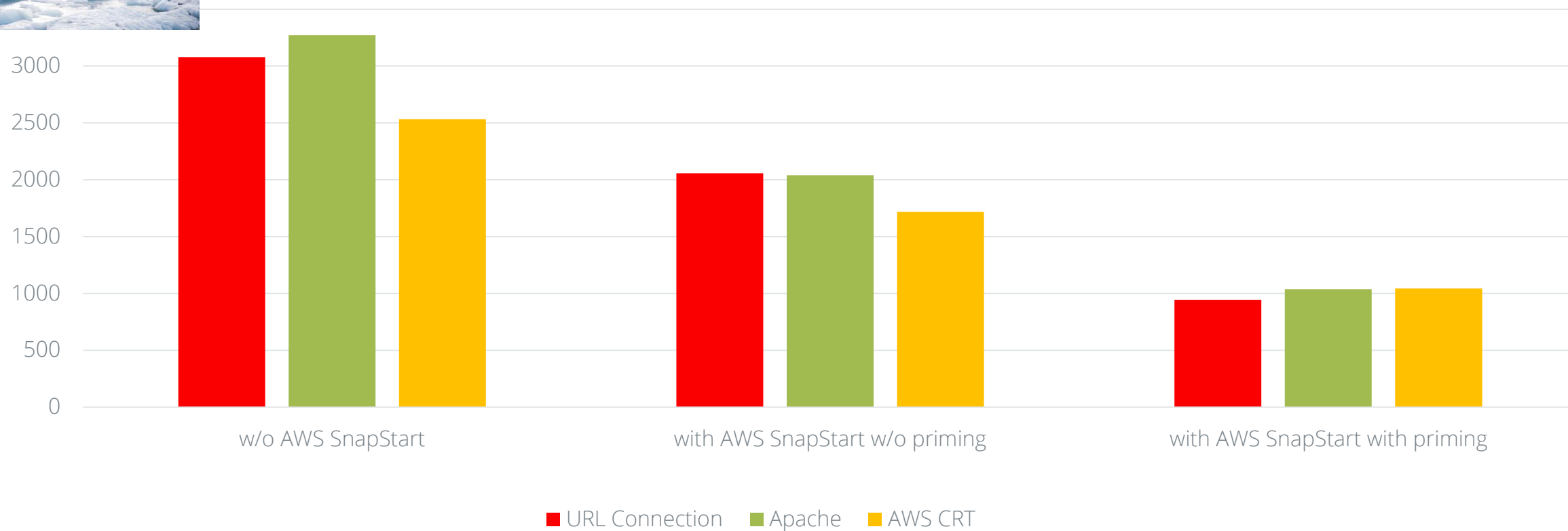
```
DynamoDbClient client = DynamoDbClient.builder().region(Region.EU_CENTRAL_1)
    .httpClient(ApacheHttpClient.create())
    //httpClient(URLConnectionHttpClient.create())
    //httpClient(AwsCrtHttpClient.create())
    .build();
```

Add dependency to the synchronous HTTP Client in use to pom.xml, i.e.

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>aws-crt-client</artifactId>
</dependency>
```

Cold starts of Lambda function with Java 21 runtime and StopAtLevel 1 compilation and different **synchronous** HTTP Clients for **p90**

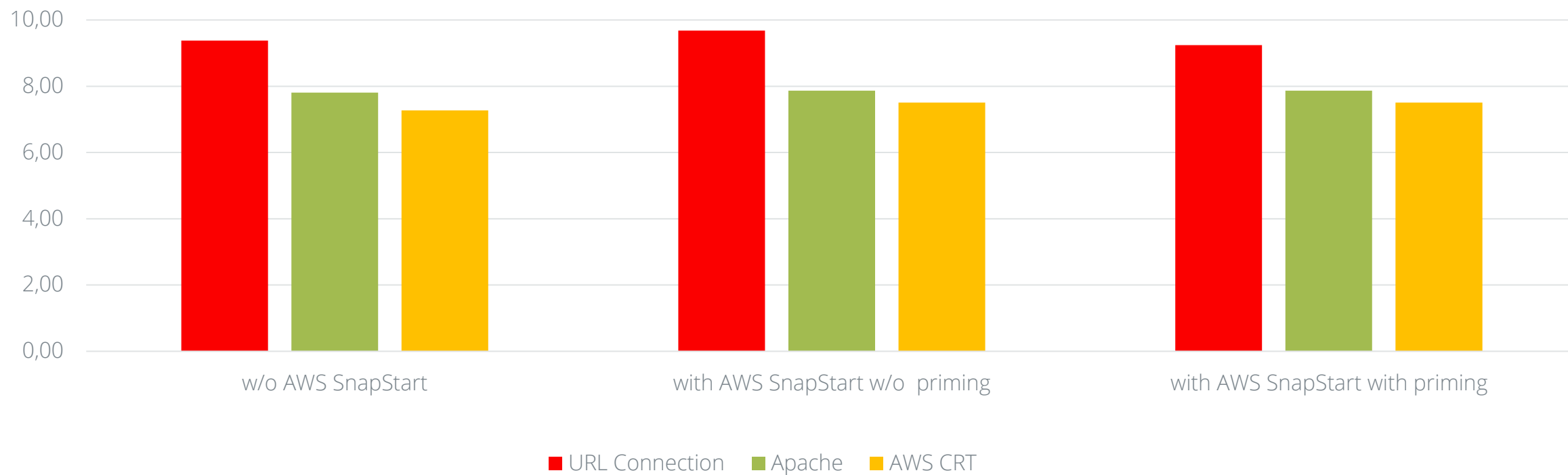
ms



Warm starts of Lambda function with Java 21 runtime and StopAtLevel 1 compilation and different **synchronous** HTTP Clients for **p90**



ms



Setting asynchronous DynamoDB Client

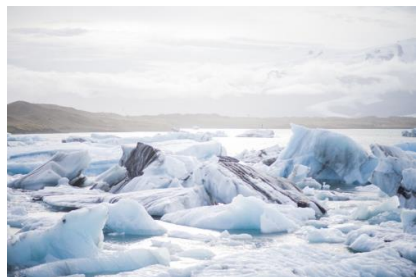
```
DynamoDbAsyncClient dynamoAsyncDbClient =  
DynamoDbAsyncClient.builder().region(Region.EU_CENTRAL_1)  
.httpClient(NettyNioAsyncHttpClient.create())  
//.httpClient(AwsCrtAsyncHttpClient.create())  
.build();  
  
CompletableFuture<GetItemResponse> getItemReponseAsync =  
dynamoAsyncDbClient.getItem(GetItemRequest.builder().  
key(Map.of("PK", AttributeValue.builder().  
s(id).build()))).tableName(PRODUCT_TABLE_NAME).build());  
  
GetItemResponse getItemResponse = getItemReponseAsync.join();
```

Setting asynchronous DynamoDB Client

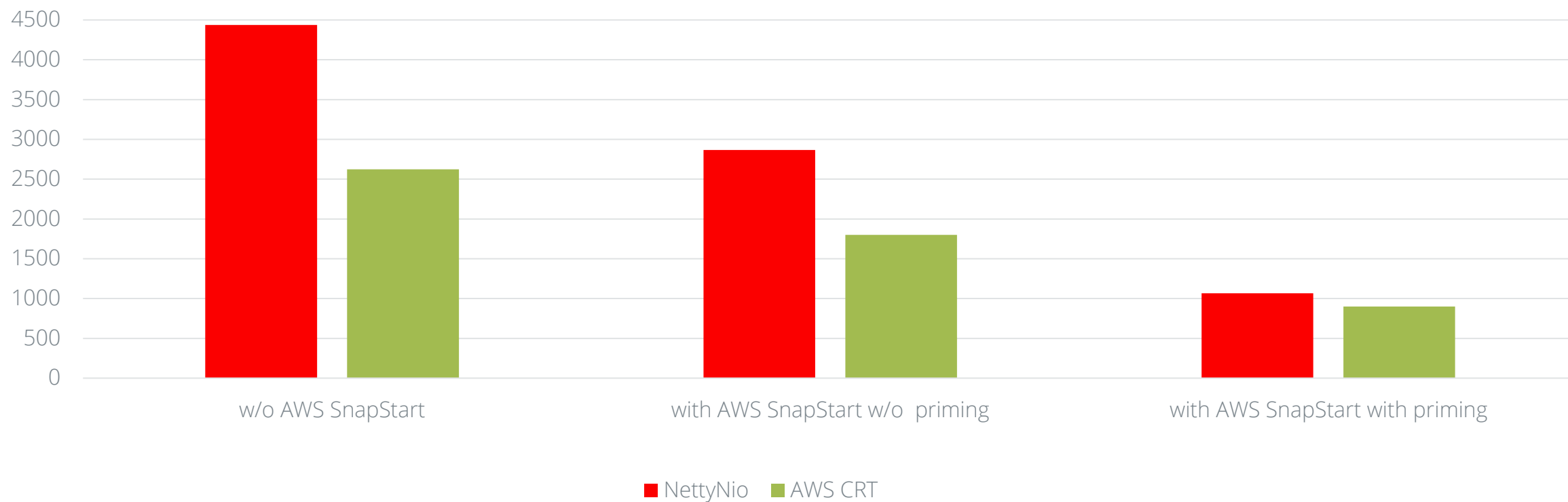
Add dependency to the **asynchronous** HTTP Client in use to pom.xml, i.e.

```
<dependency>  
  <groupId>software.amazon.awssdk</groupId>  
  <artifactId>netty-nio-client</artifactId>  
</dependency>
```

Cold starts of Lambda function with Java 21 runtime and StopAtLevel 1 compilation and different asynchronous HTTP Clients for p90



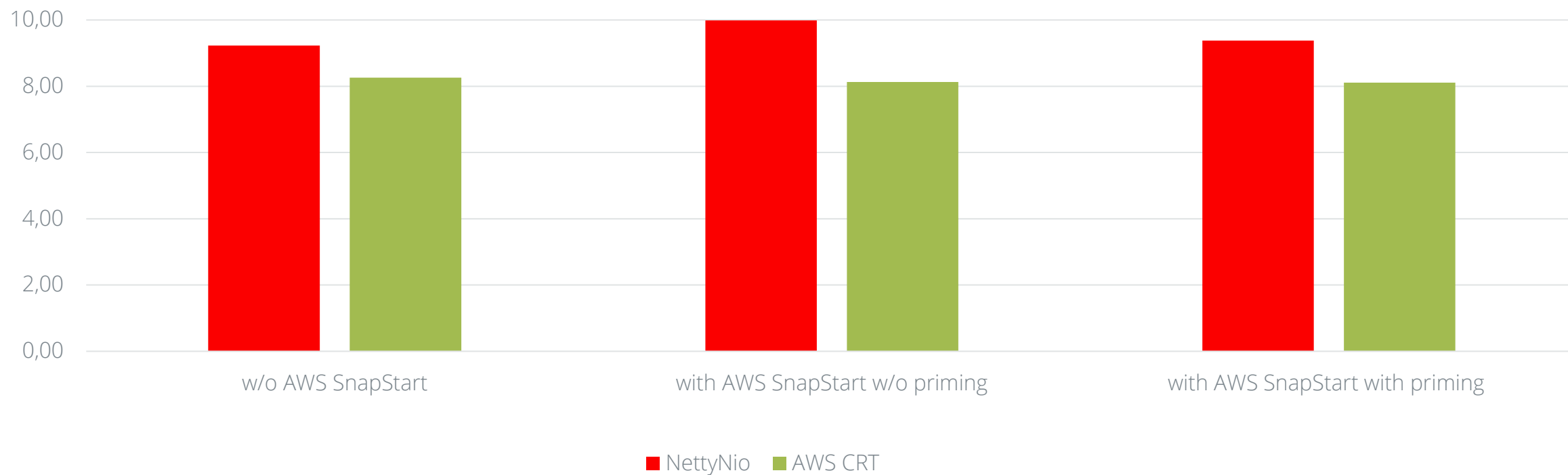
ms



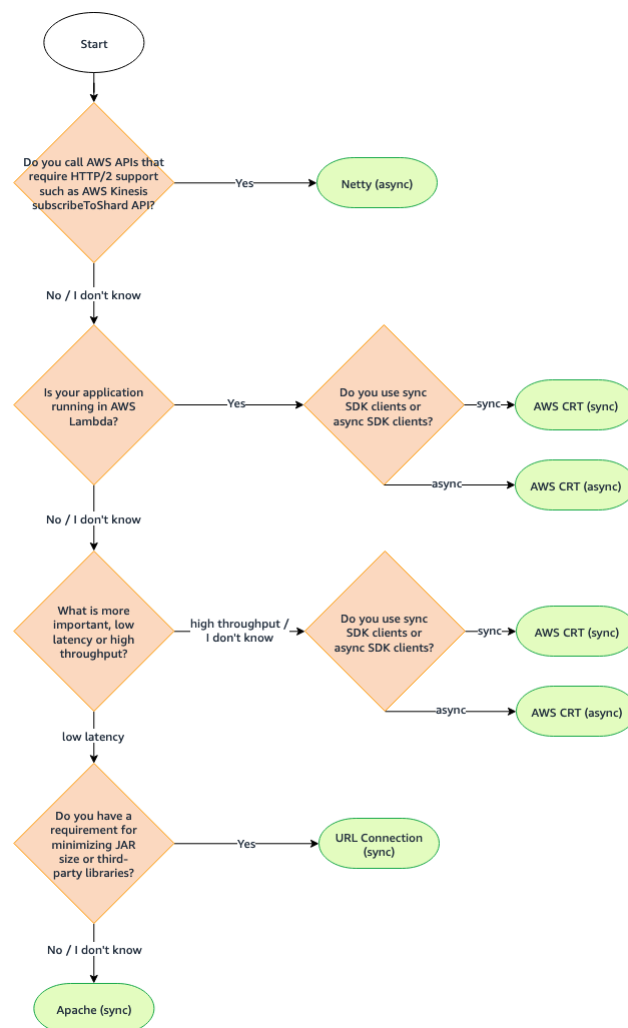
Warm starts of Lambda function with Java 21 runtime and StopAtLevel 1 compilation and different **asynchronous** HTTP Clients for **p90**



ms



HTTP client recommendations



Lambda x86_64 vs arm64 architecture

AWS Lambda now supports SnapStart for Java functions that use the ARM64 architecture

Posted on: Jul 18, 2024

Starting today, you can use Lambda SnapStart with Java functions that use the ARM64 instruction set architecture.

SnapStart for Java delivers up to 10x faster function startup performance at no extra cost, enabling you to build highly responsive and scalable Java applications using AWS Lambda without having to provision resources or implement complex performance optimizations. This launch expands SnapStart's performance benefits to functions running on ARM64 architecture, which enables upto 34% better price performance as compared to x86.

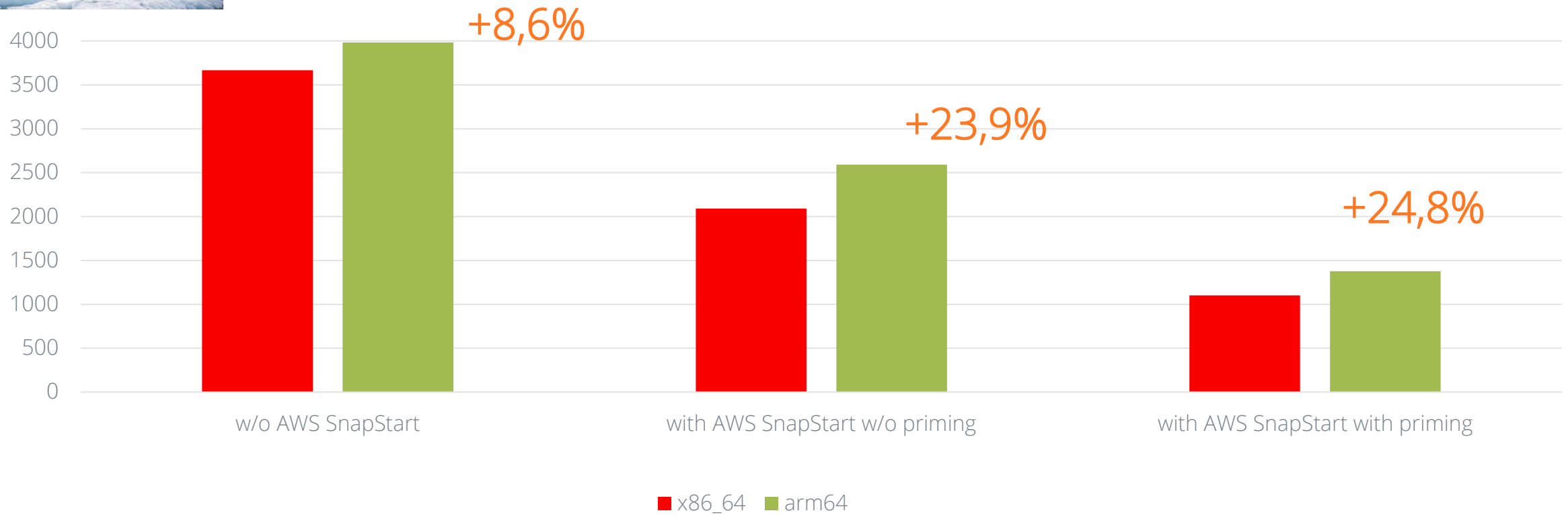
Lambda x86_64 vs arm64 architecture

For the same memory setting Lambda and Lambda function execution duration the choice of arm64 over x86_64 architecture is approx. 25% cheaper

Cold starts of Lambda function with Java 21 runtime with
1024 MB memory setting, Apache Http Client, compilation
-XX:+TieredCompilation -XX:TieredStopAtLevel=1 for p90



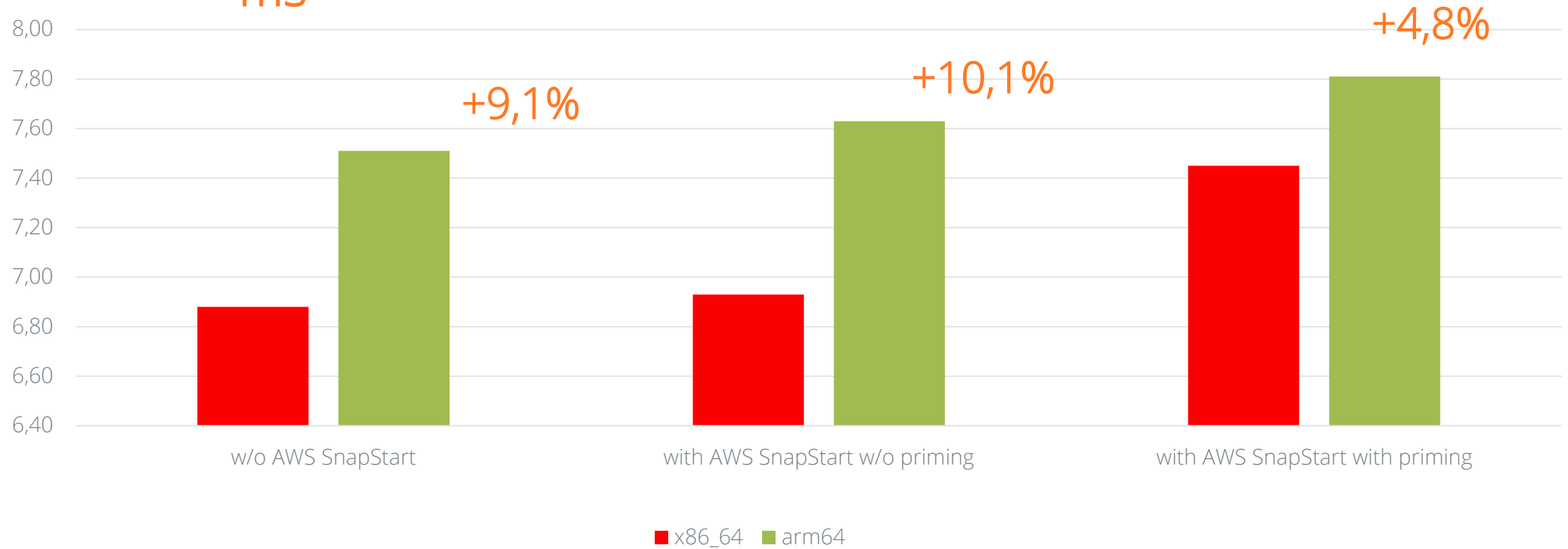
ms



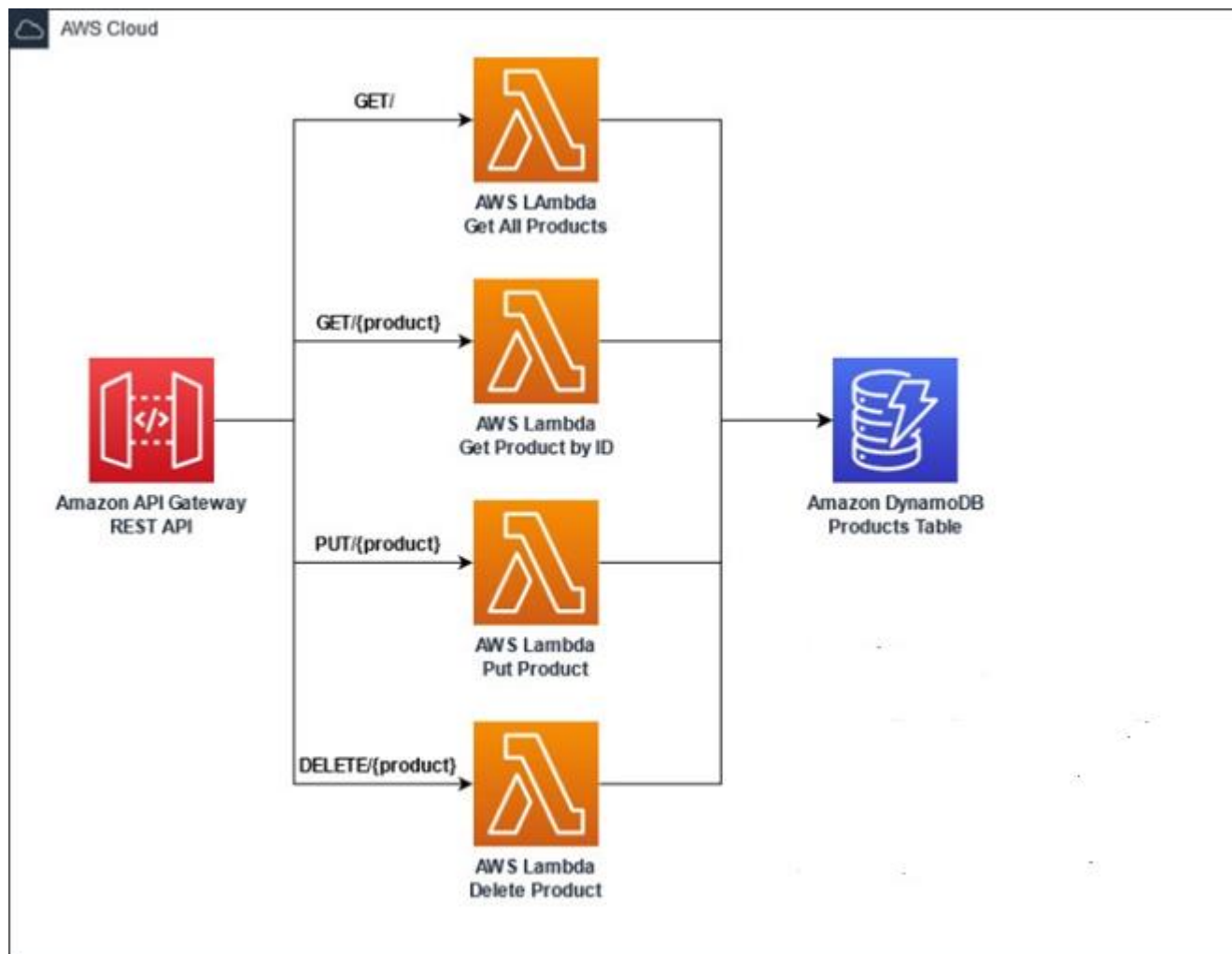


Warm starts of Lambda function with Java 21 runtime with
1024 MB memory setting, Apache Http Client, compilation
-XX:+TieredCompilation -XX:TieredStopAtLevel=1 for p90

ms



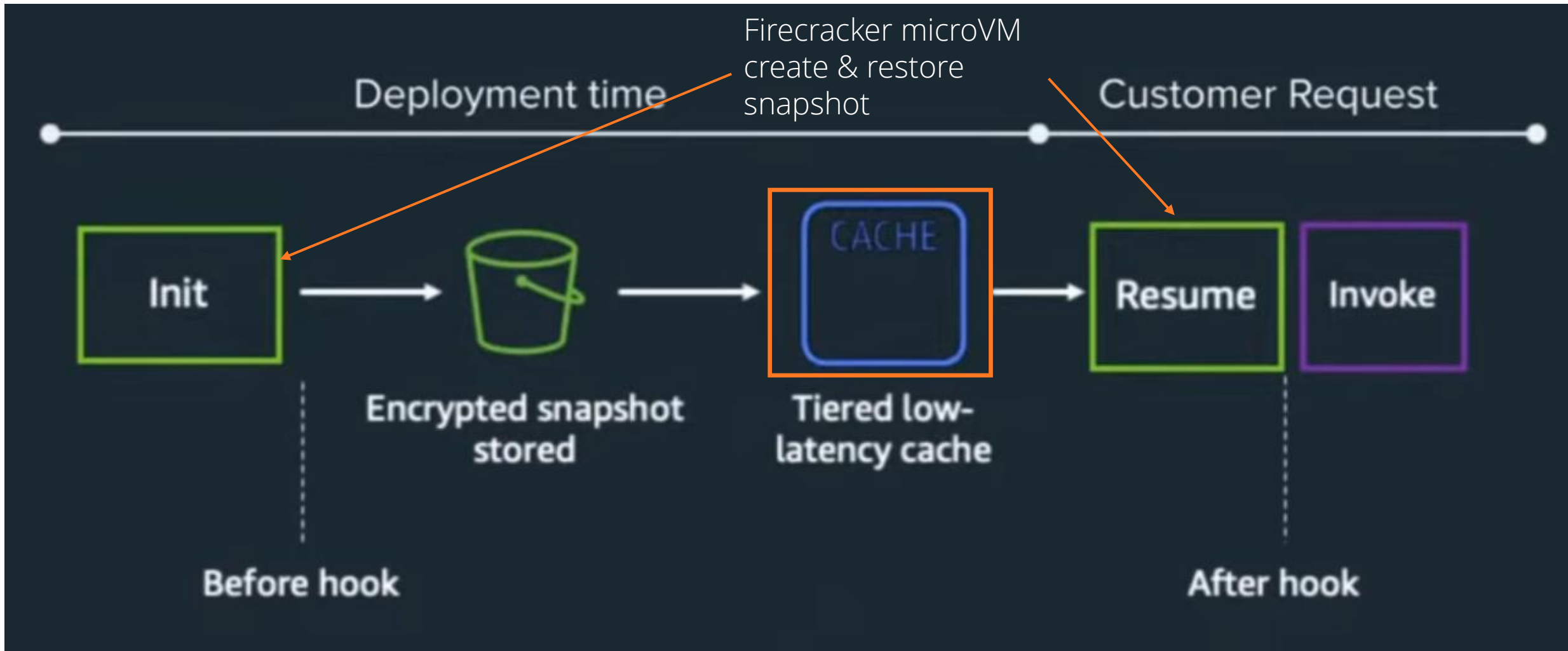
Demo Application



- Lambda has 1024 MB memory setting
 - Lambda uses x86 architecture
 - Default (Apache) Http Client for communication with DynamoDB
 - 18 MB artifact size, , all dependencies in the POM file
 - Java compilation option
XX:+TieredCompilation
XX:TieredStopAtLevel=1
 - Info about the experiments:
 - Approx. 1 hour duration
 - Approx. **first*** 100 cold starts
 - Approx. first 100.000 warm starts
- *after Lambda function being re-deployed



AWS SnapStart Deployment & Invocation



AWS SnapStart tiered cache



Vadym Kazulkin for AWS Community Builders

Posted on Mar 11 • Updated on Mar 16

Edit Manage Stats



AWS SnapStart - Part 17 Impact of the snapshot tiered cache on the cold starts with Java 21

- Due to the effect of **snapshot tiered cache**, cold start times **reduces** with the number of invocations
- After certain number of invocations reached the cold start times becomes stable

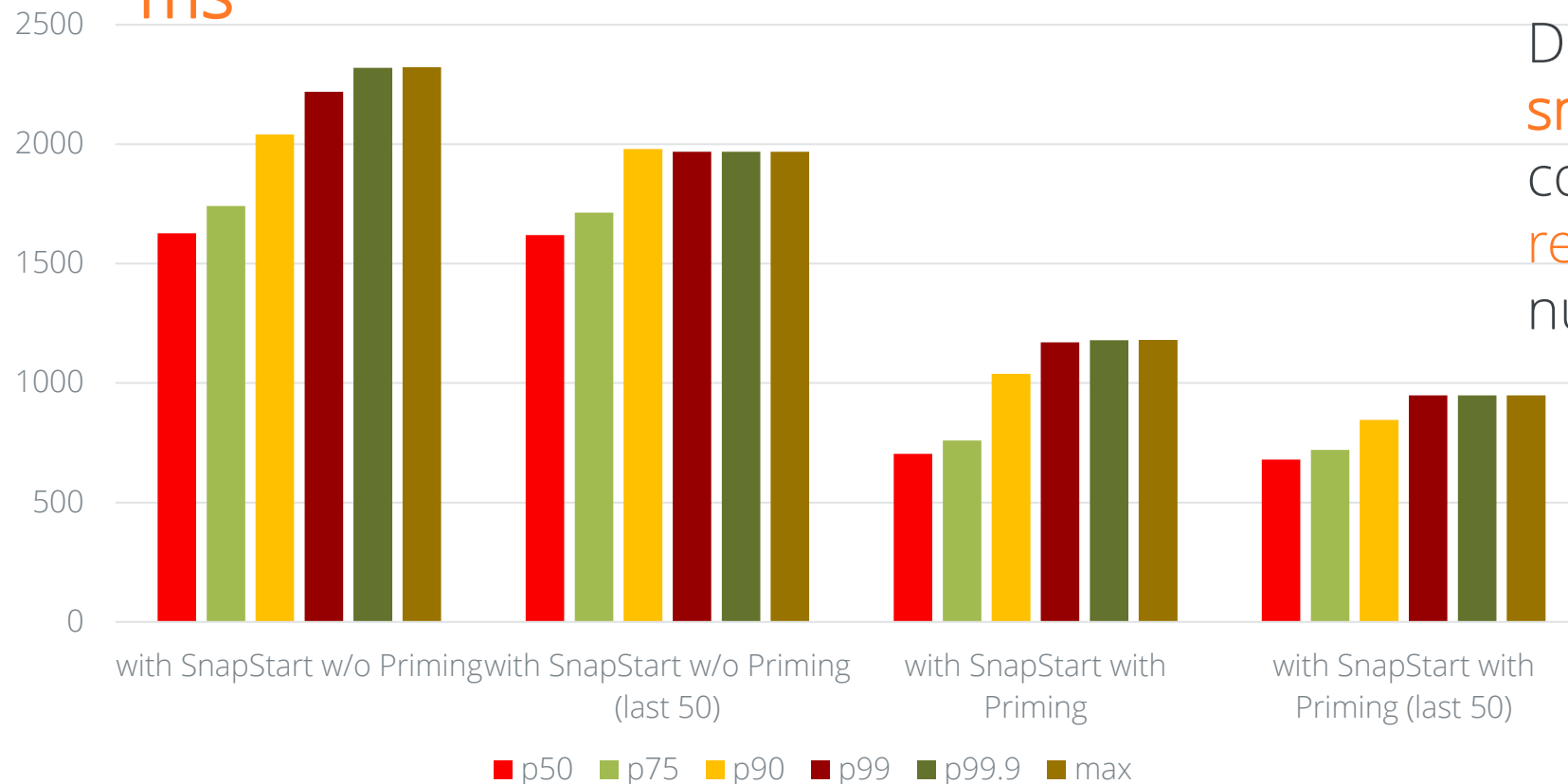


Cold starts of **all approx. 100 and last 50** Lambda functions with Java 21 runtime with

1024 MB memory setting, Apache Http Client, compilation

`-XX:+TieredCompilation -XX:TieredStopAtLevel=1`

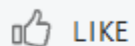
ms



Due to the effect of **snapshot tiered cache**, cold start times **reduces** with the number of invocations

AWS Lambda under the Hood

AWS Lambda under the Hood



LIKE



MAR 14, 2024 • 9 MIN READ

by



Mike Danilov

FOLLOW

Senior Principal Engineer @AWS Lambda

reviewed by

Steef-Jan
Wiggers

FOLLOW

Cloud Queue Lead Editor



Key Takeaways

- Lambda allows users to execute code on demand without the overhead of server management and operations, enabling efficient execution across various integrated languages.
- Lambda offers synchronous and asynchronous invocation models; synchronous invokes ensure a rapid response, whereas asynchronous invokes queue requests for deferred execution.

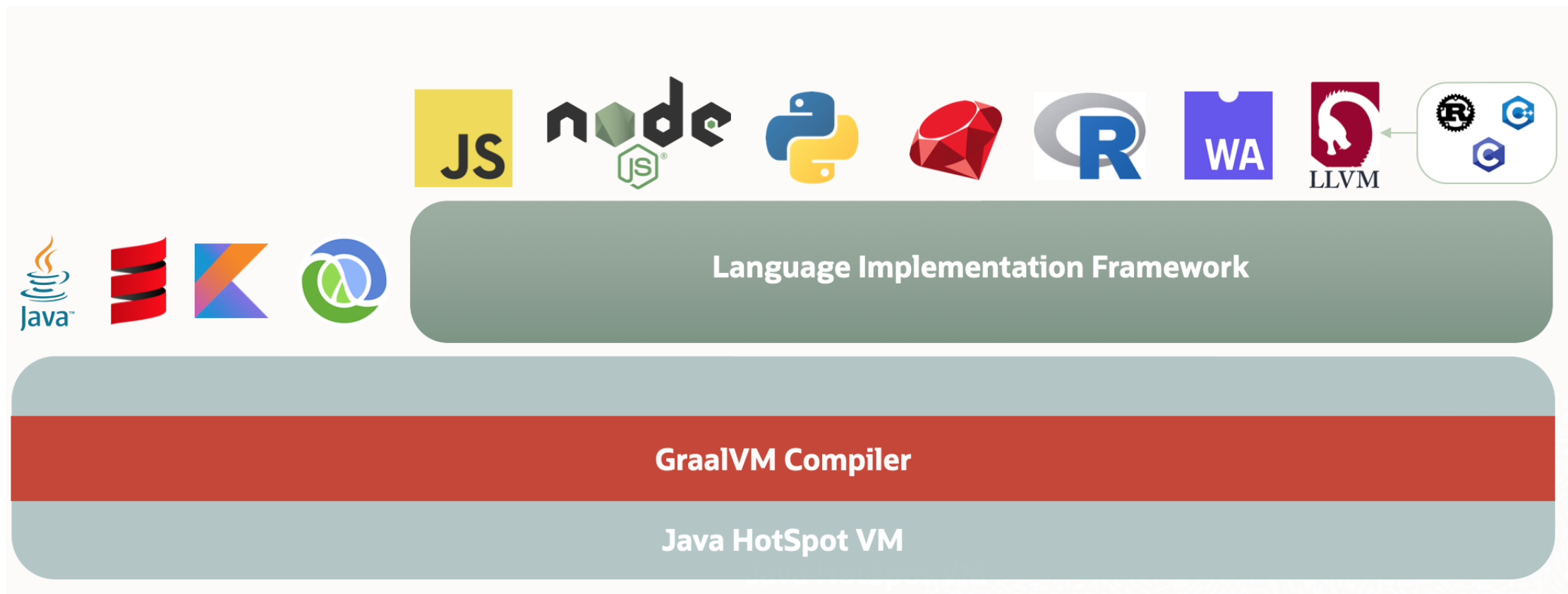
The logo for GraalVM, featuring the word "Graal" in white and "VM" in orange, with a small "TM" trademark symbol to the right. The logo is positioned in the bottom right corner of the slide, which has a solid blue background. The left side of the slide features a blurred image of a modern building with large glass windows and greenery in the foreground.

GraalVM™

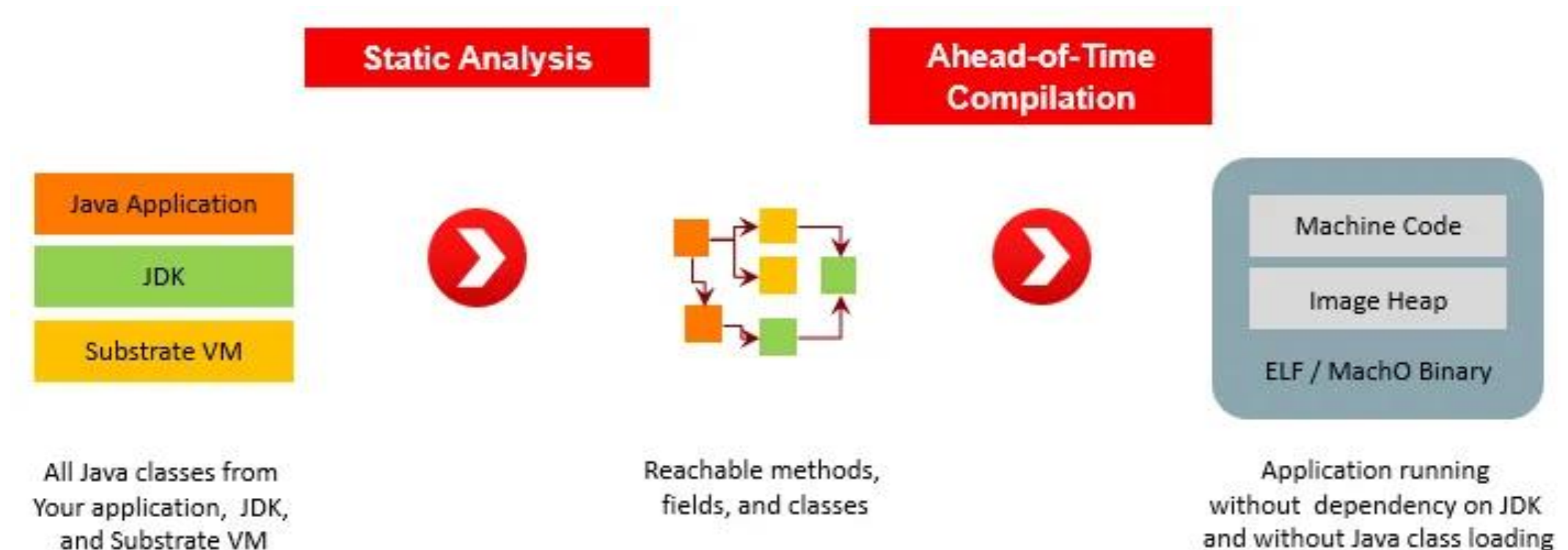
GraalVM Goals

- Low footprint ahead-of-time mode for JVM-based languages
- High performance for all languages
- Convenient language interoperability and polyglot tooling

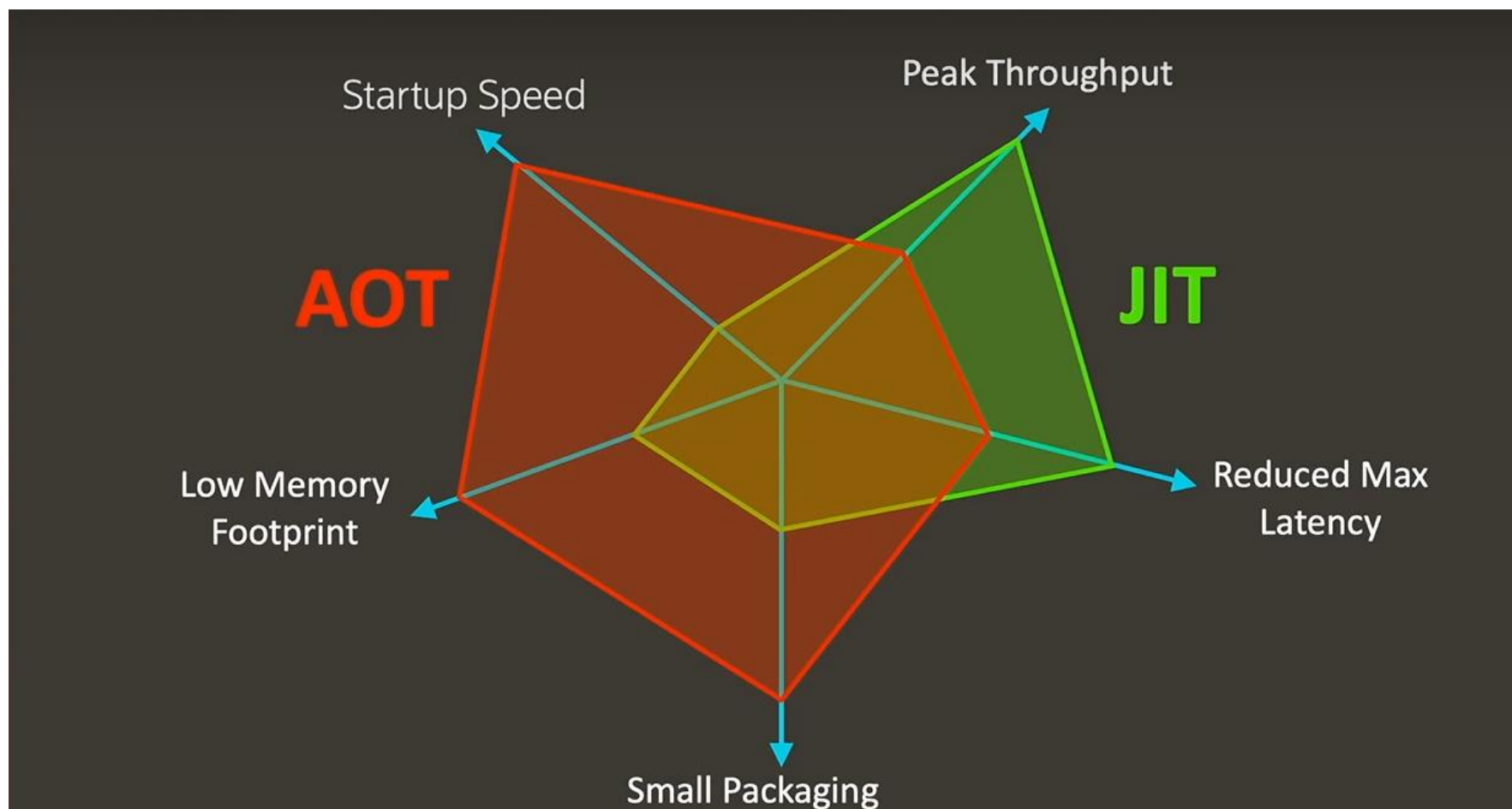
GraalVM Architecture



GraalVM Ahead-of-Time Compilation



AOT vs JIT



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

GraalVM Native Image

Compilation into a **native executable** using **GraalVM** 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

Runtime [Info](#)

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

Java 11 (Corretto)

Ruby 2.7

Other supported

Java 8 on Amazon Linux 1

Java 8 on Amazon Linux 2

Node.js 12.x

Python 3.6

Python 3.7

Python 3.8

Custom runtime

Use default bootstrap on Amazon Linux 1

Provide your own bootstrap on Amazon Linux 2

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 [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 Release Calendar

GraalVM Release Calendar

GraalVM aligns with the six-month JDK release cadence and uses the JDK numbering scheme (for example, "GraalVM for JDK 21.0.3").

Feature Release

There are two feature releases per year to support the latest JDK version. A new feature release supersedes all previous releases.

Critical Patch Update (CPU)

Critical Patch Updates (CPU) for Oracle GraalVM and GraalVM Community Edition follow the schedule for all Oracle CPU and OpenJDK CPU releases respectively. Critical Patch Updates are released on the third Tuesday of January, April, July, and October. All active releases receive patch updates.

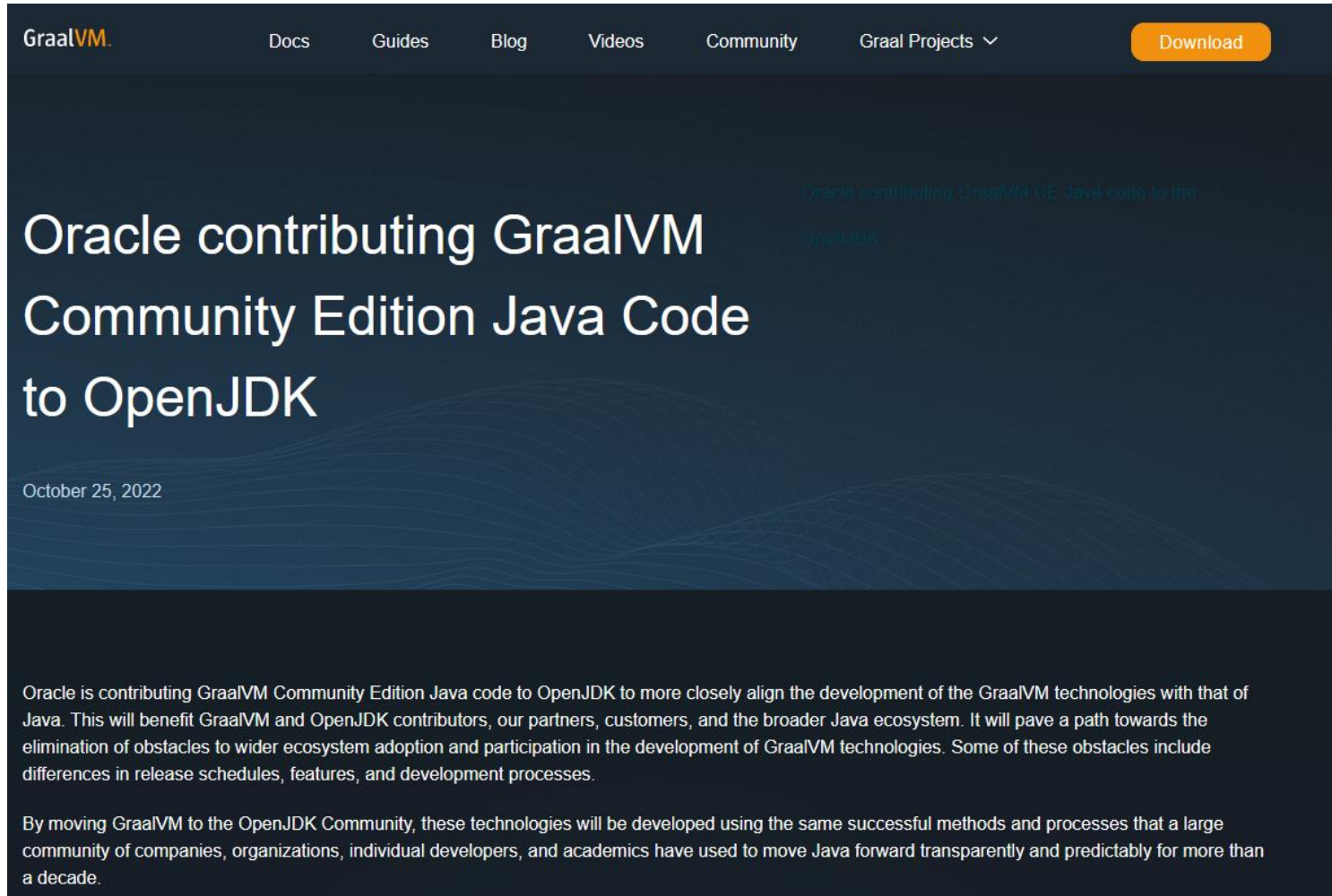
Long-Term-Support (LTS) Release

Oracle GraalVM for JDK 21 is the current long-term support (LTS) release and will receive free updates under the [GraalVM Free Terms and Conditions \(GFTC\) including License for Early Adopter Versions](#) license until one full year after the subsequent LTS release, which is JDK 25. Oracle also provides customers of Oracle GraalVM and Oracle GraalVM Enterprise Edition with [Oracle Premier Support](#).

Planned Releases

Date	Type	Oracle GraalVM	GraalVM Community Edition
July 16, 2024	CPU	17.0.12, 21.0.4, 22.0.2	22.0.2
September 17, 2024	Feature	23	23
October 15, 2024	CPU	17.0.13, 21.0.5, 23.0.1	23.0.1
January 21, 2025	CPU	17.0.14, 21.0.6, 23.0.2	23.0.2
March 18, 2025	Feature	24	24
April 15, 2025	CPU	17.0.15, 21.0.7, 24.0.1	24.0.1
July 15, 2025	CPU	17.0.16, 21.0.8, 24.0.2	24.0.2

GraalVM CE is based on OpenJDK



The screenshot shows the GraalVM website with a dark blue background and white text. The header includes the GraalVM logo, navigation links (Docs, Guides, Blog, Videos, Community, Graal Projects), and a Download button. The main headline reads 'Oracle contributing GraalVM Community Edition Java Code to OpenJDK'. Below this, the date 'October 25, 2022' is displayed. The body text explains that Oracle is contributing GraalVM CE Java code to OpenJDK to align development and benefit the ecosystem. It also mentions that by moving GraalVM to the OpenJDK Community, these technologies will be developed using the same successful methods and processes as Java.

GraalVM. Docs Guides Blog Videos Community Graal Projects ▾ Download

Oracle contributing GraalVM Community Edition Java Code to OpenJDK

October 25, 2022

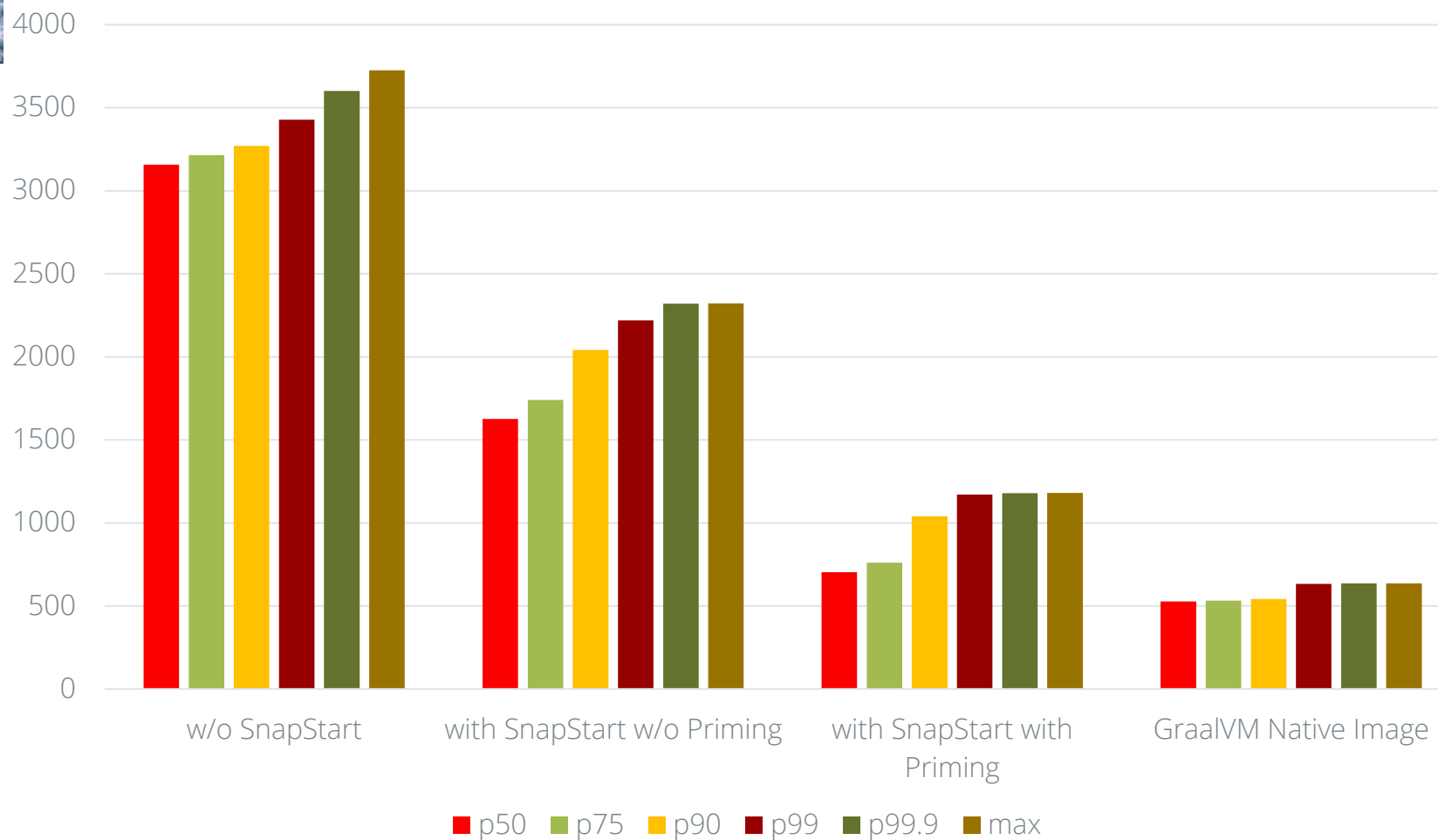
Oracle is contributing GraalVM Community Edition Java code to OpenJDK to more closely align the development of the GraalVM technologies with that of Java. This will benefit GraalVM and OpenJDK contributors, our partners, customers, and the broader Java ecosystem. It will pave a path towards the elimination of obstacles to wider ecosystem adoption and participation in the development of GraalVM technologies. Some of these obstacles include differences in release schedules, features, and development processes.

By moving GraalVM to the OpenJDK Community, these technologies will be developed using the same successful methods and processes that a large community of companies, organizations, individual developers, and academics have used to move Java forward transparently and predictably for more than a decade.

<https://www.graalvm.org/2022/openjdk-announcement/>
<https://blogs.oracle.com/java/post/graalvm-free-license>

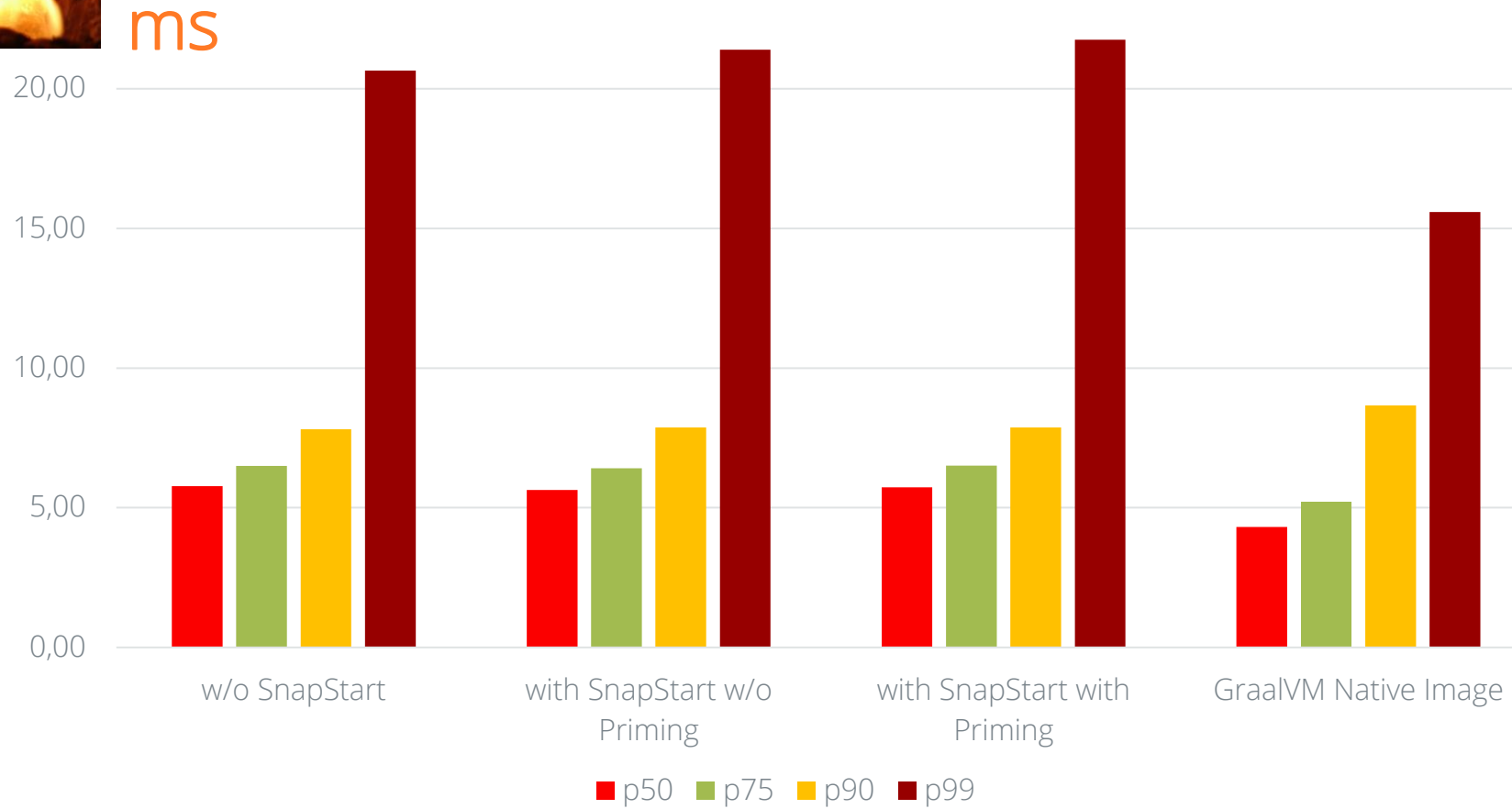
ms

Cold starts of Lambda function with Java 21 runtime with.
1024 MB memory setting, Apache Http Client



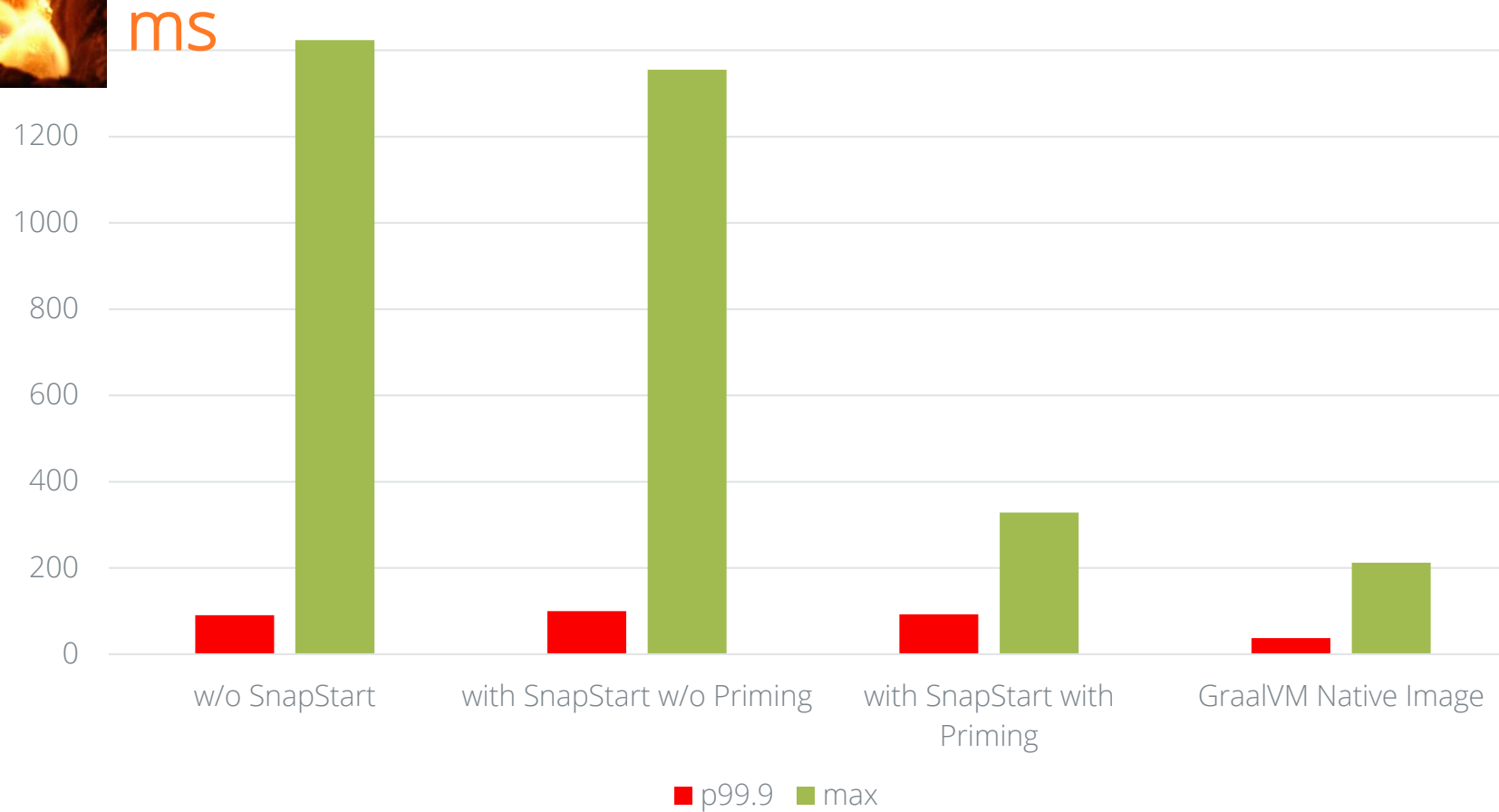


Warm starts of Lambda function with Java 21 runtime with 1024 MB memory setting, Apache Http Client





Max warm starts of Lambda function with Java 21 runtime with 1024 MB memory setting, Apache Http Client



Lambda Memory Setting for Custom Runtime (GraalVM Native Image)



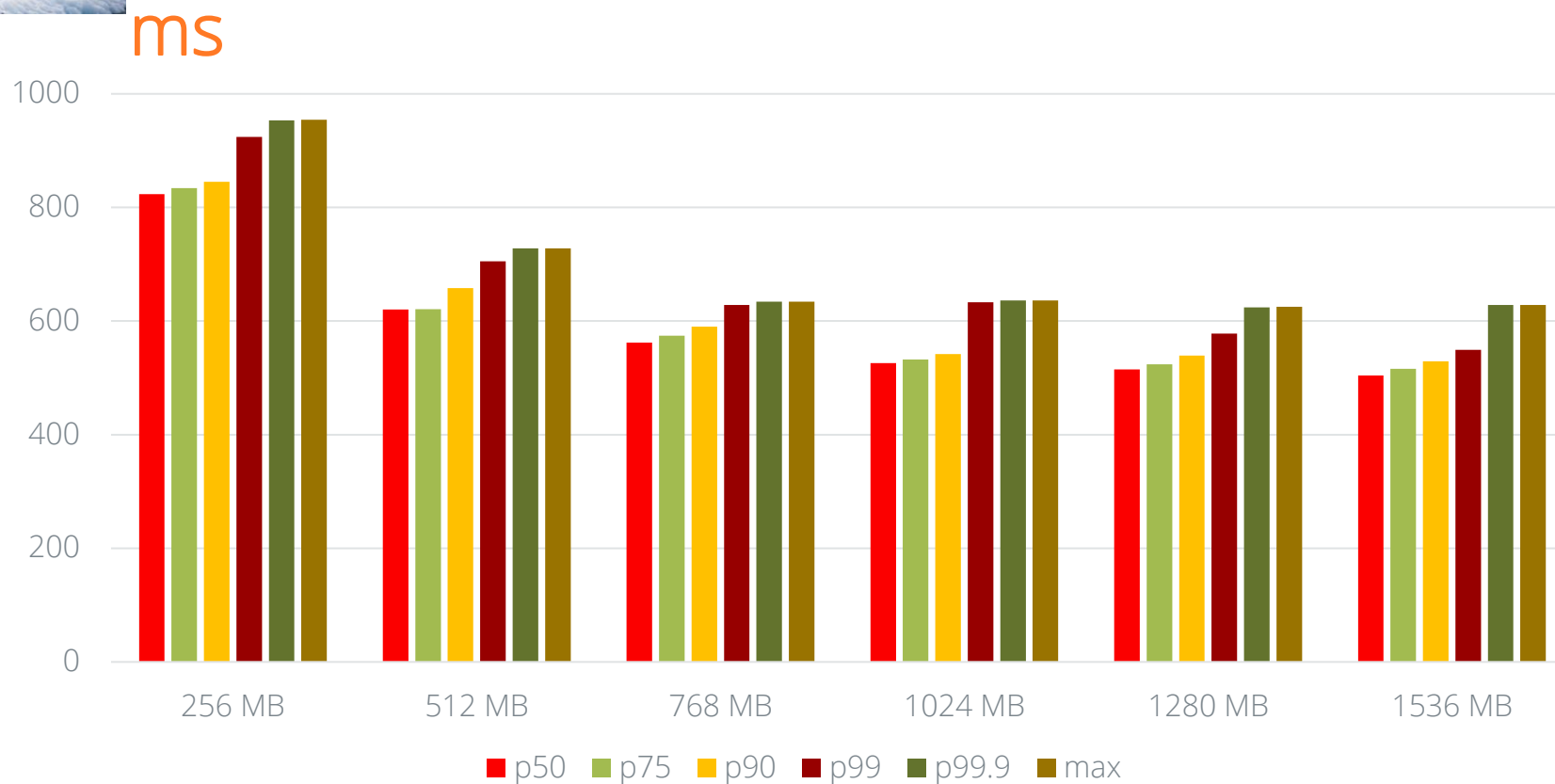
ONE SECOND



ONE GB



Cold starts of Lambda function with GraalVM 21 Native Image for different memory settings





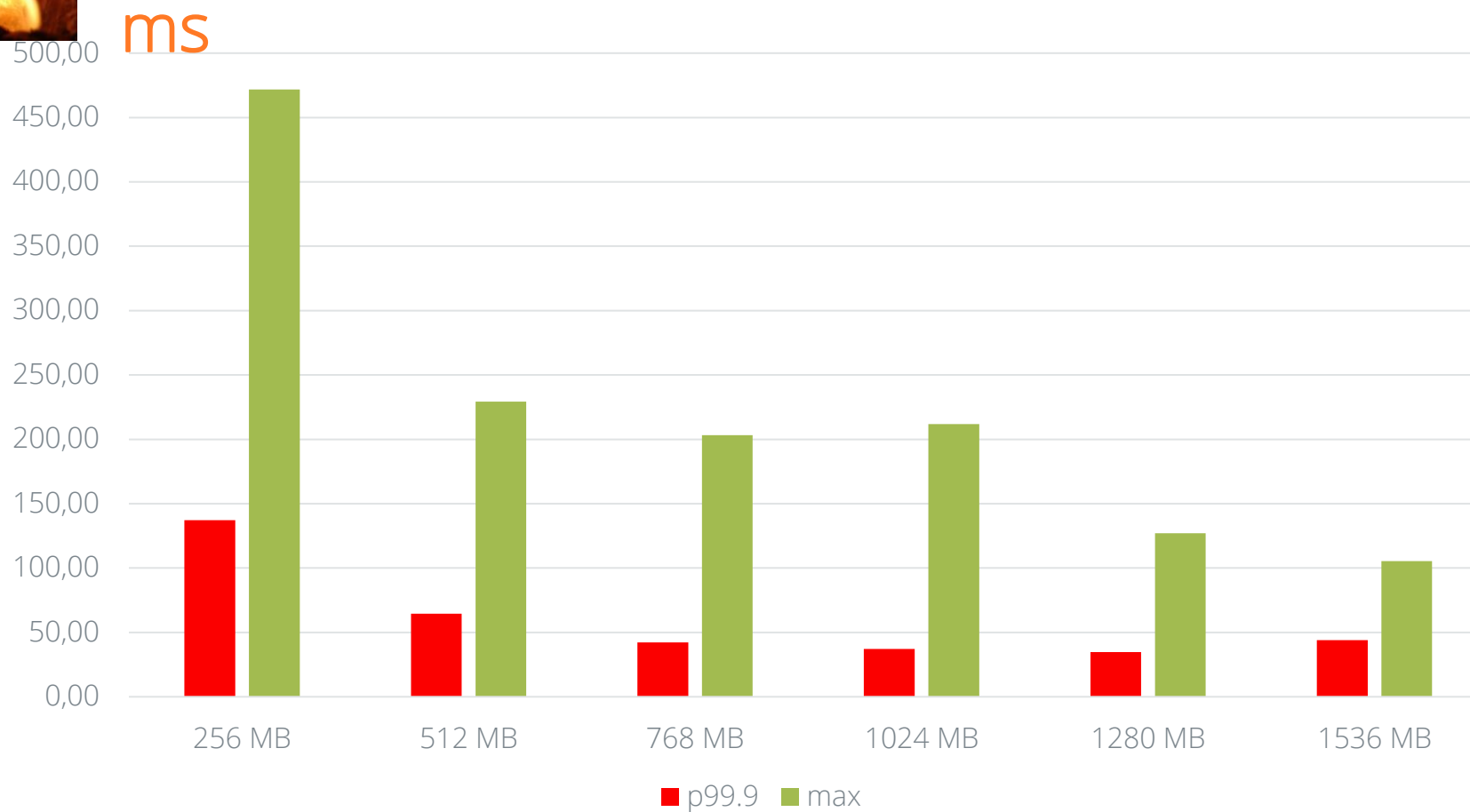
Warm starts of Lambda function with GraalVM 21 Native Image for different memory settings

ms





Warm starts of Lambda function with GraalVM 21 Native Image for different memory settings



GraalVM Conclusion

- GraalVM is really powerful and has a lot of potential
- GraalVM Native Image improves cold starts and memory footprint significantly
- GraalVM Native Image is currently not without challenges
 - AWS Lambda Custom Runtime requires Linux executable only
 - Building Custom Runtime requires some additional effort
 - e.g. you need a scalable CI/CD pipeline to build memory-intensive native image
- 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 11, Java17 and Java 21

GraalVM Conclusion

- You can run into errors when application is running


```
[
  {
    "name": "software.amazonaws.example.product.handler.GetProductByIdHandler",
    "allPublicConstructors": true,
    "allPublicMethods": true
  },
  {
    "name": "software.amazonaws.example.product.handler.CreateProductHandler",
    "allPublicConstructors": true,
    "allPublicMethods": true
  },
  {
    "name": "org.joda.time.DateTime",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  },
  {
    "name": "software.amazonaws.example.product.entity.Product",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  },
  {
    "name": "software.amazonaws.example.product.entity.Products",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent$ProxyRequestContext",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  },
  {
    "name": "com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent$requestIdentity",
    "allPublicConstructors": true,
    "allDeclaredFields": true
  }
]
```


Frameworks and libraries Ready for GraalVM 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

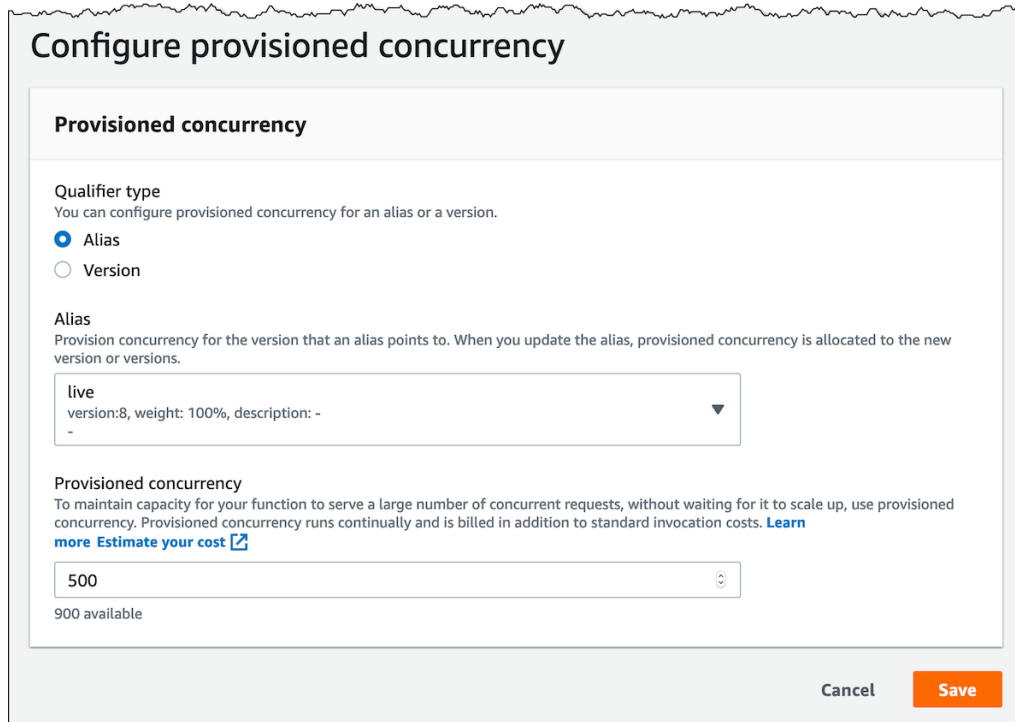
The following table lists libraries and frameworks from the Java ecosystem that are tested with GraalVM Native Image. Each item in the list is annotated with a *test level*, as follows:

- **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	★

Lambda Provisioned Concurrency



Configure provisioned concurrency

Provisioned concurrency

Qualifier type
You can configure provisioned concurrency for an alias or a version.

☒ Alias
☐ Version

Alias
Provision concurrency for the version that an alias points to. When you update the alias, provisioned concurrency is allocated to the new version or versions.

live
version:8, weight: 100%, description: -
-

Provisioned concurrency
To maintain capacity for your function to serve a large number of concurrent requests, without waiting for it to scale up, use provisioned concurrency. Provisioned concurrency runs continually and is billed in addition to standard invocation costs. [Learn more](#) [Estimate your cost](#)

500
900 available

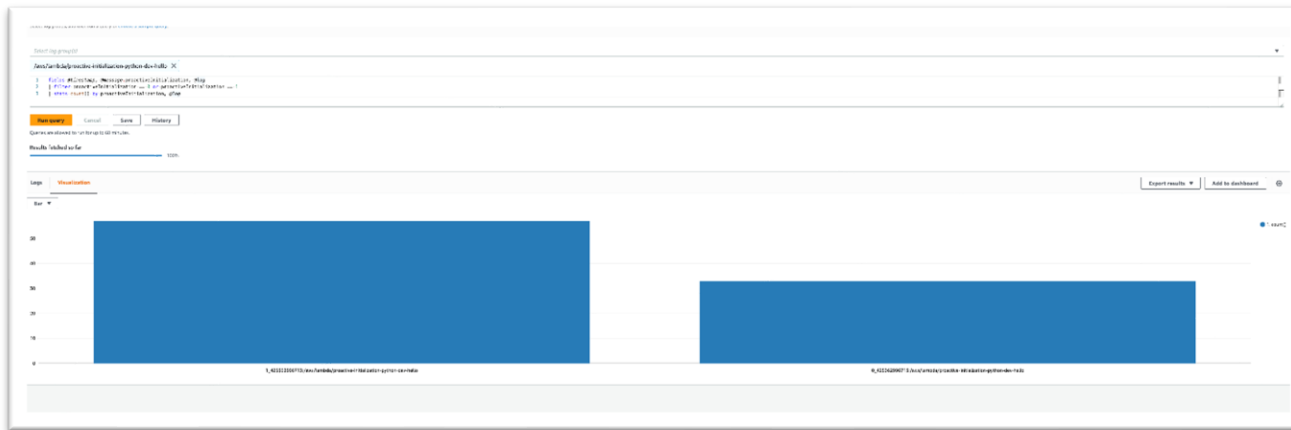
Cancel Save

Requires manually managing start and end time when provisioned concurrency should apply (can be tricky for spikey workloads)

- You pay for unused capacity


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 proactive** (versus 50% to 25% which were true cold starts)

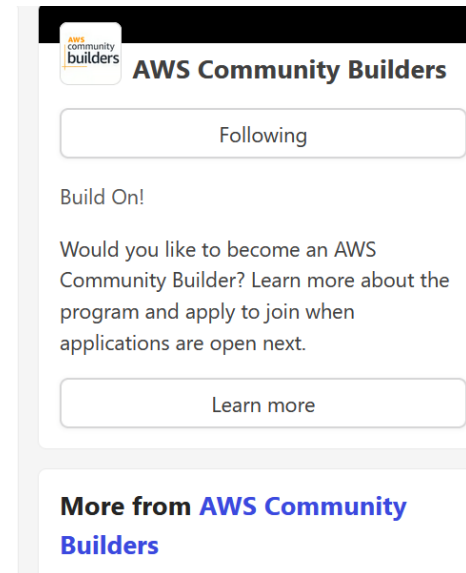
“Data API for Amazon Aurora Serverless v2 with AWS SDK for Java” series

 **Vadym Kazulkin** for AWS Community Builders
Posted on Feb 5

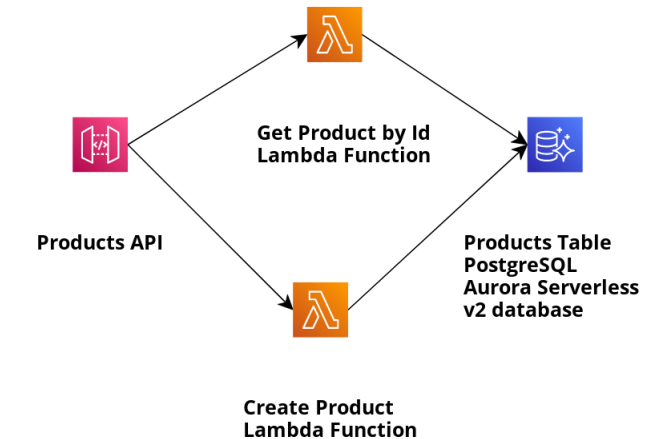
Edit Manage Stats

5


Data API for Amazon Aurora Serverless v2 with AWS SDK for Java - Part 1 Introduction and set up of the sample application



Article series also covers cold and warm start time measurements and optimization techniques



“Spring Boot 3 application on AWS Lambda” series

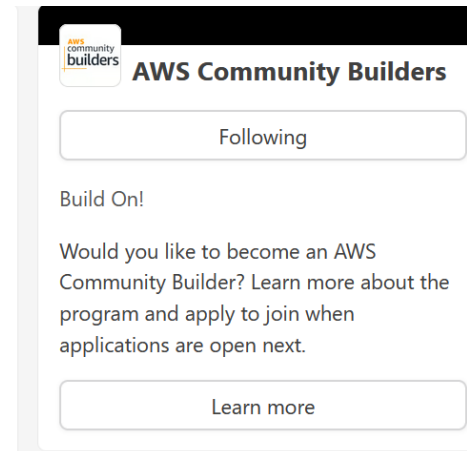
 **Vadym Kazulkin** for AWS Community Builders
Posted on Mar 25

Edit Manage Stats

❤️ 4

Spring Boot 3 application on AWS Lambda - Part 1 Introduction to the series

#aws #serverless #java #springboot



Article series covers different ways to write Spring Boot 3 application on AWS Lambda

- AWS Serverless Java Container
- AWS Lambda Web Adapter
- Spring Cloud Functions
- Custom Docker Image
- GraalVM Native Image

Cold and warm start time measurements are also provided

Wrap up and personal suggestions

- With AWS SnapStart and GraalVM Native Image you can reduce **cold start times** of the AWS Lambda with Java 21 runtime to the acceptable values
- If you're willing to accept slightly **higher cold and warm start times** for certain the Lambda function(s) and solid priming is applicable -> use fully managed **AWS SnapStart with priming**
- If a very high performance for certain the Lambda function(s) is really crucial for your business -> go for **GraalVM Native Image**

Project Leyden

Project Leyden *Capturing Lightning in a Bottle*

Mark Reinhold
Chief Architect, Java Platform Group, Oracle

John Rose
JVM Senior Architect, Java Platform Group, Oracle

JVM Language Summit
2023/8/8

Copyright © 2023, Oracle and/or its affiliates



The primary goal of this Project is to improve the startup time, time to peak performance, and footprint of Java programs.

Word of caution



Re-measure for your use case!

Even with my examples measurements might already produce different results due to:

- Lambda Amazon Corretto Java 21 managed runtime minor version changes
- Lambda SnapStart snapshot create and restore improvements
- Firecracker VM improvements
- GraalVM (major and minor version) and Native Image improvements
- There are still servers behind Lambda
- Java Memory Model impact (L or RAM caches hits and misses)

FAQ Ask me Anything



Thank you

