beyond

PAAS — AWS — FIRECRACKER

SERVERLESS — LAMBDA — MICRONAUT

JAVA — GRAAL VM — QUARKUS

beyond

Java on AWS Lambda

AWS Lambda

- Basics (Demo)
- Lambda Execution Environment
- Custom Runtime (Demo)
- AWS Firecracker

AWS Lambda and Java

- AWS Lambda w/ Java Runtime (Demo)
- Pitfalls
- Accelerating Serverless Java

Compile Time Frameworks

- Quarkus (Demo)
- Micronaut (Demo)
- Sketchy comparison

AWS Lambda

Introduction

Cloud Computing

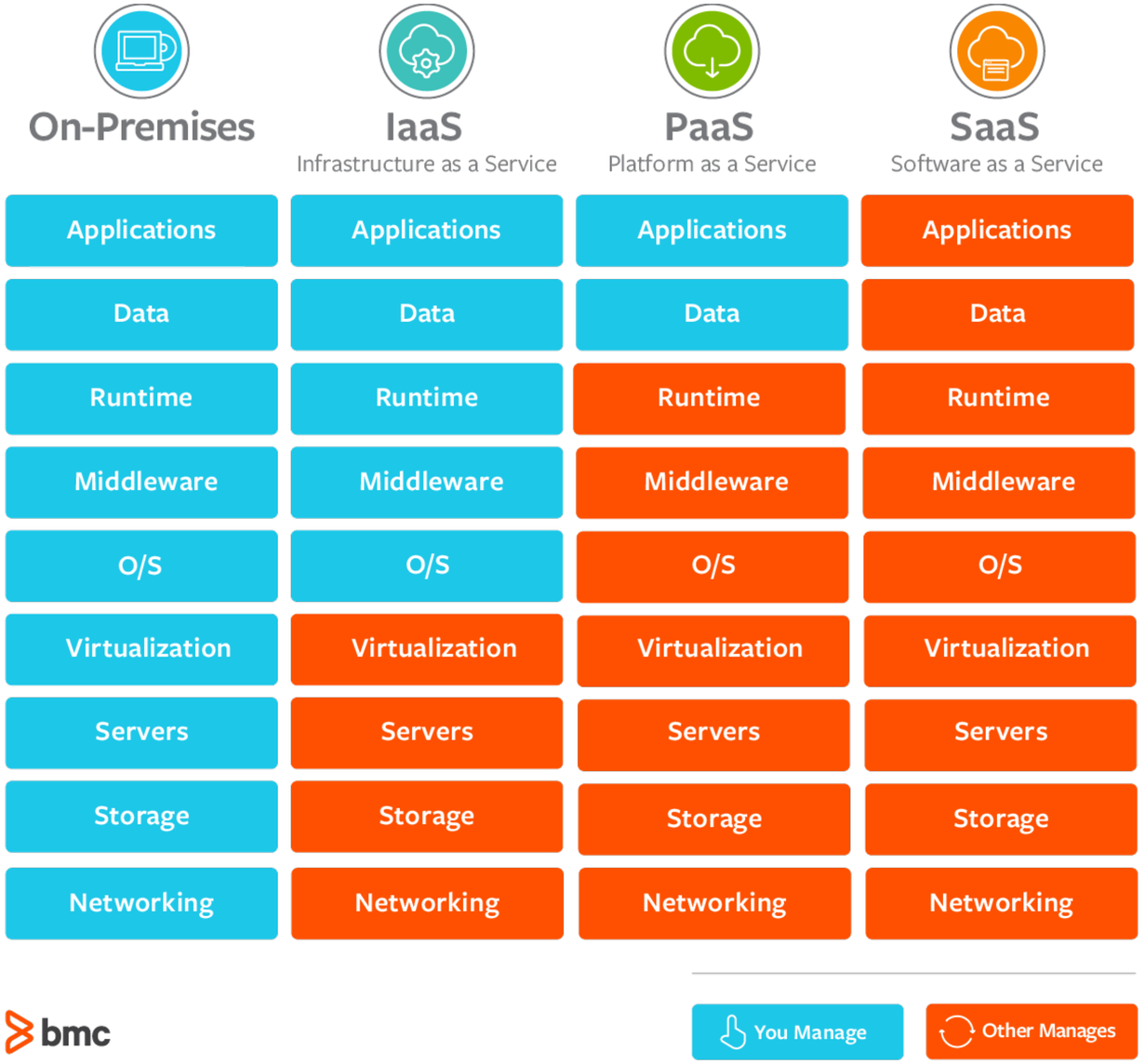
Cloud computing is the **on-demand availability of computer system resources**, especially data storage (cloud storage) and computing power, without direct active management by the user.

Cloud Computing Service Models

- IaaS
- **PaaS**
- SaaS

Cloud Computing Types

- **Public cloud** is cloud computing that's delivered via the internet and shared across organizations.
- Private cloud is cloud computing that is dedicated solely to your organization.
- Hybrid cloud is any environment that uses both public and private clouds.



AWS Lambda

Features

AWS Lambda is a

1. **secure**,
2. **on-demand**,
3. **event-driven**,
4. **compute service**
5. that **scales** as needed (up to 1000 concurrent lambda / account)
6. **billed only for what is used**.

https://aws.amazon.com/lambda/features/

Limitations

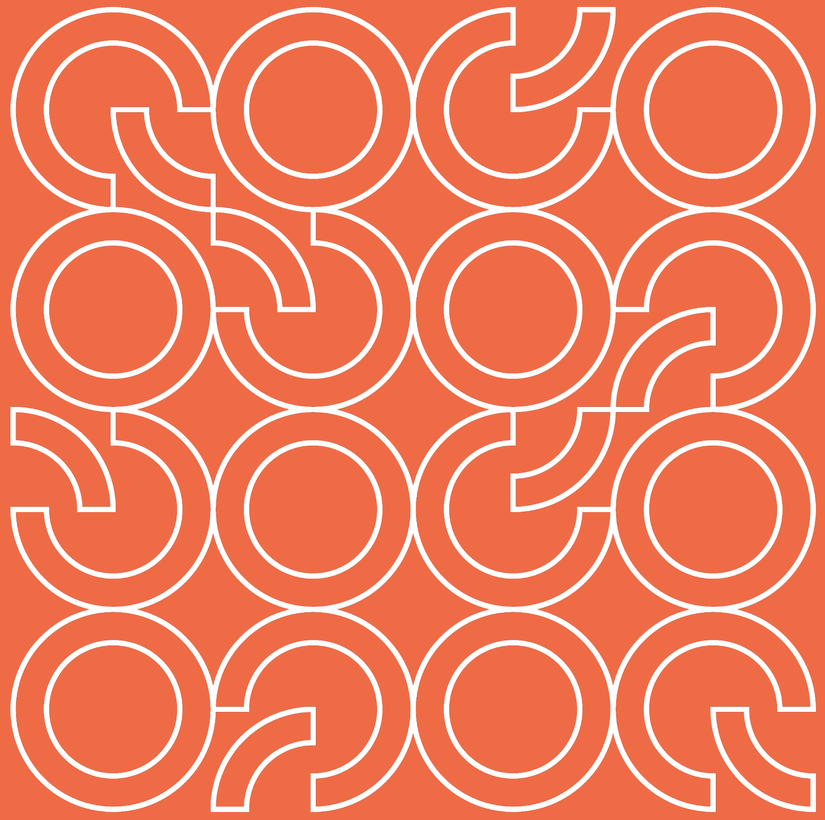
- Memory: Max 10 GB
- Max invocation time: **15 minutes**
- Max event size: 256 KB (async) / 6 MB (sync)
- Max function size (zip): 50 MB (zipped) / 250 MB (unzipped)
- Ephemeral storage: 512 MB – 10 GB

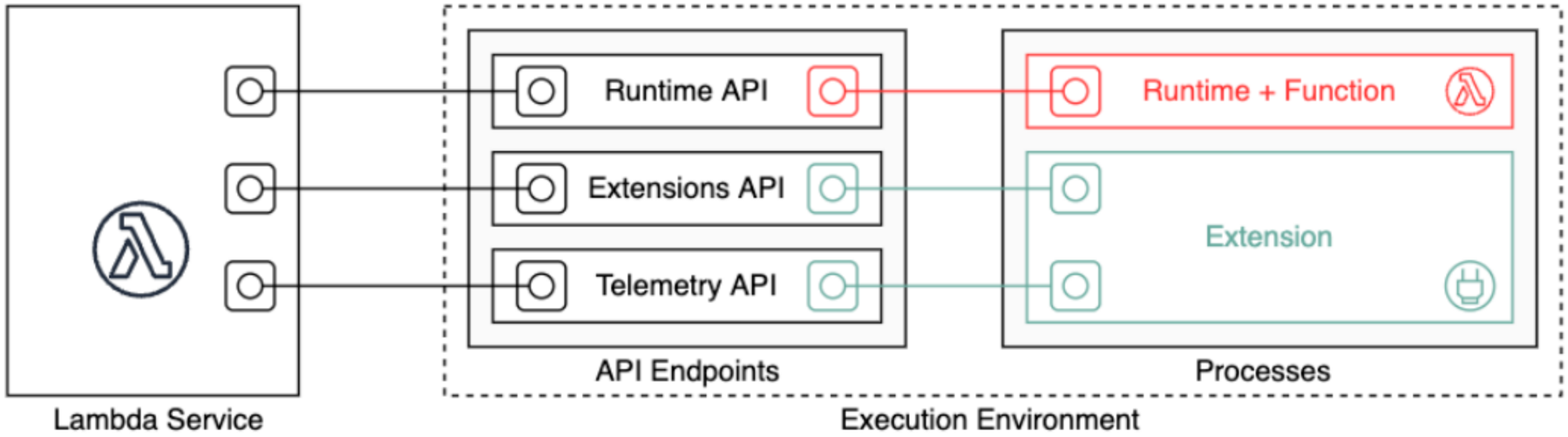
Demo #1

Lambda basics

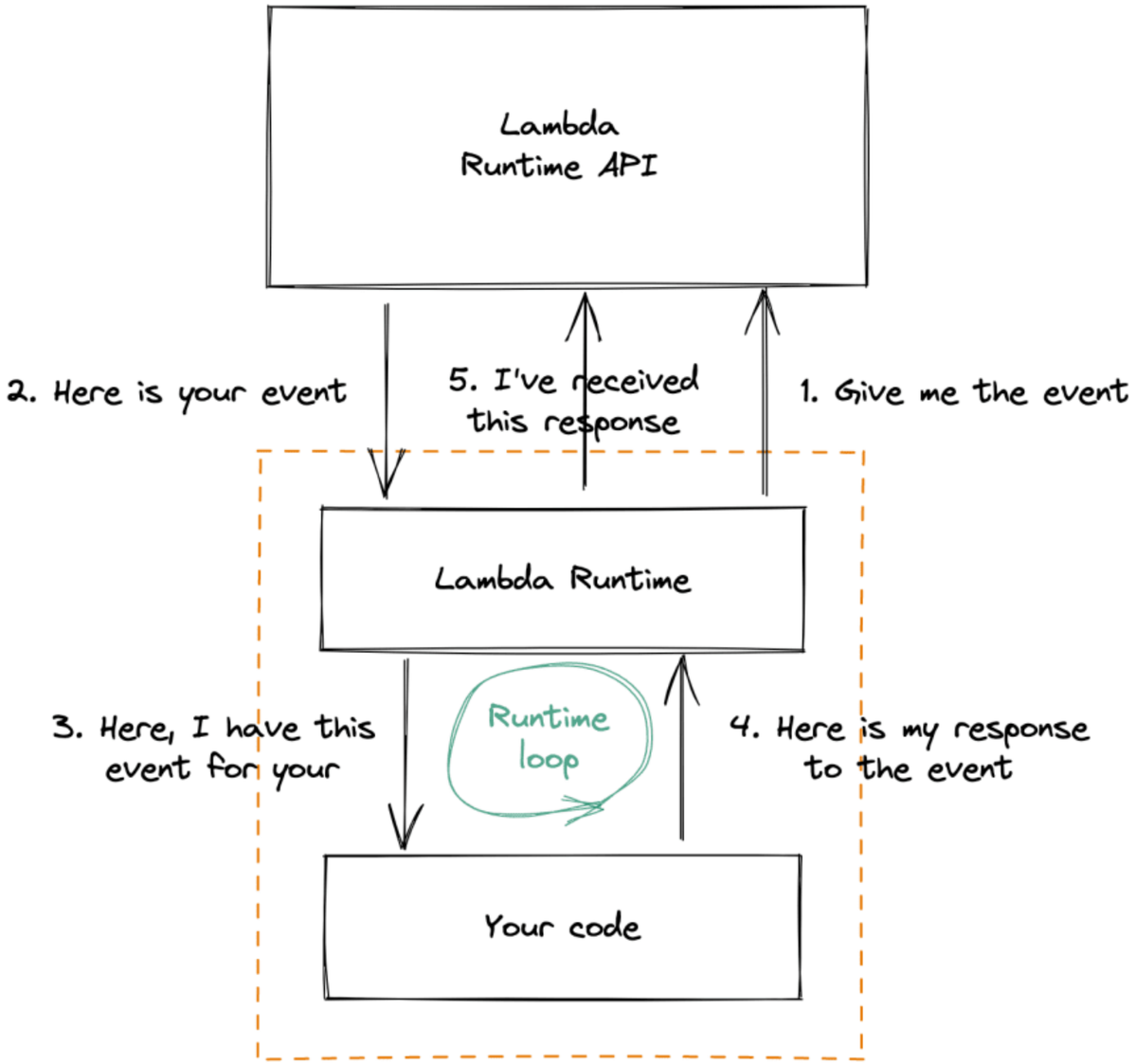
AWS Lambda

Under the hood





Lambda Execution Environment



Lambda Runtime

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.

Supported Runtimes

- Node.js 18; 16; 14; 12
- Python 3.9; 3.8; 3.7
- Java 8; 11;
- .NET Core 3.1; .NET 5; .NET 6
- Go 1.x
- Ruby 2.7*
- Custom Runtime

Operating System

- Amazon Linux v1
- Amazon Linux v2

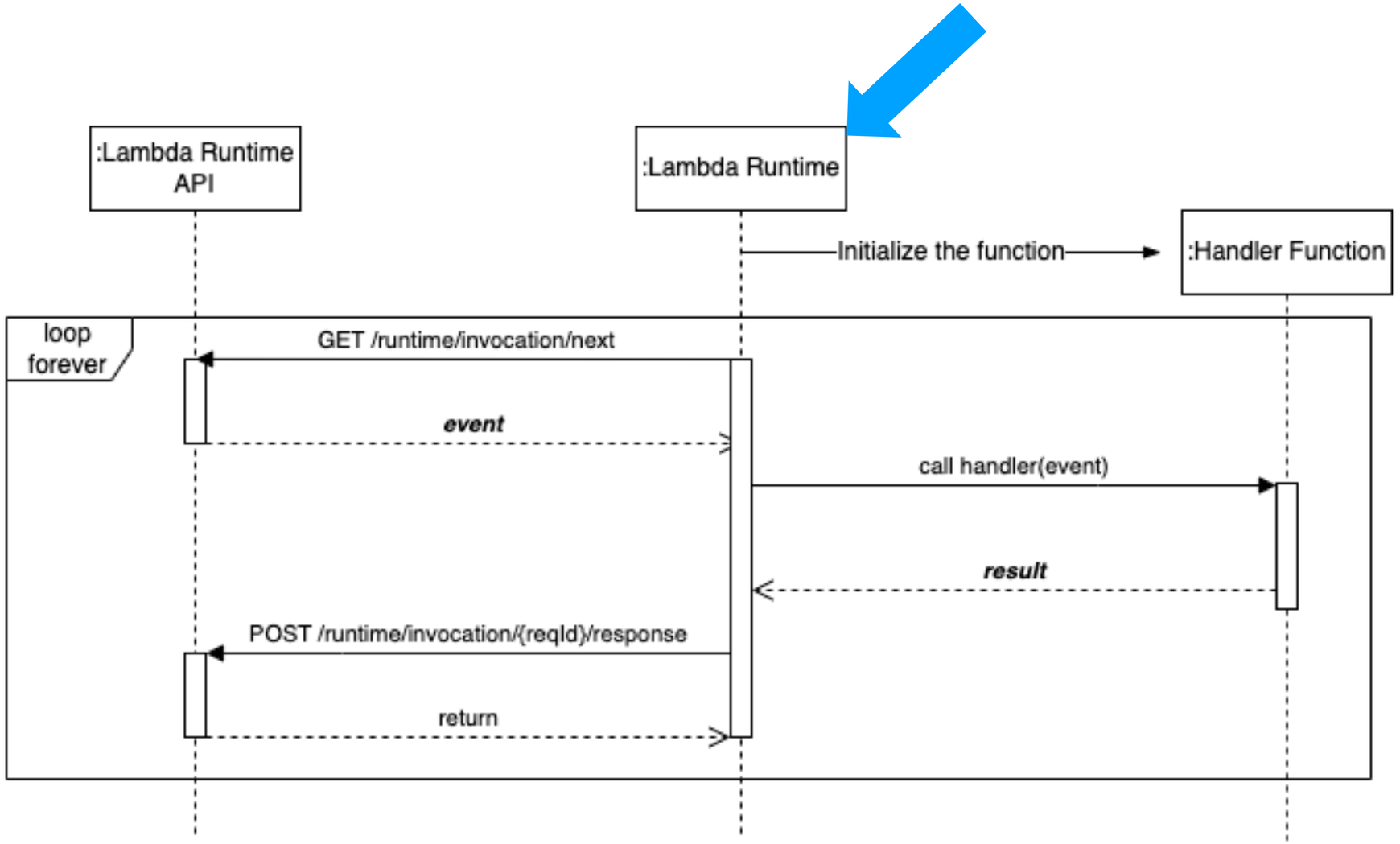
https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

Runtime API

HTTP API for runtimes to receive invocation events from Lambda and send response data back within the Lambda execution environment

Endpoints

- **GET** /runtime/invocation/next
- **POST** /runtime/invocation/{AwsRequestId}/response
- **POST** /runtime/invocation/{AwsRequestId}/error
- **POST** /runtime/init/error

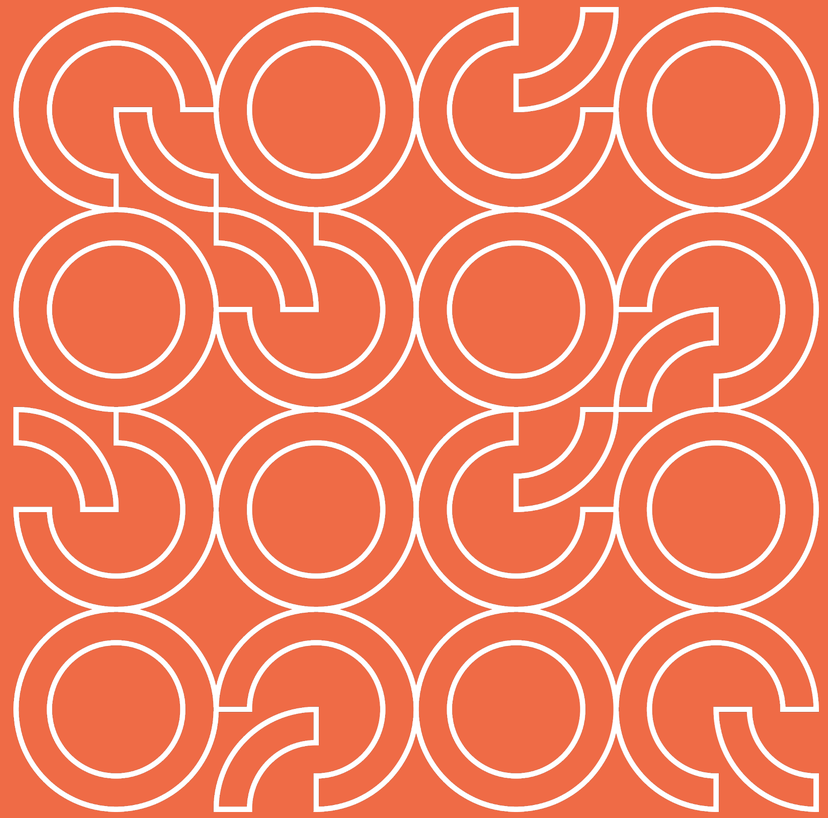


Implement our own Custom Runtime

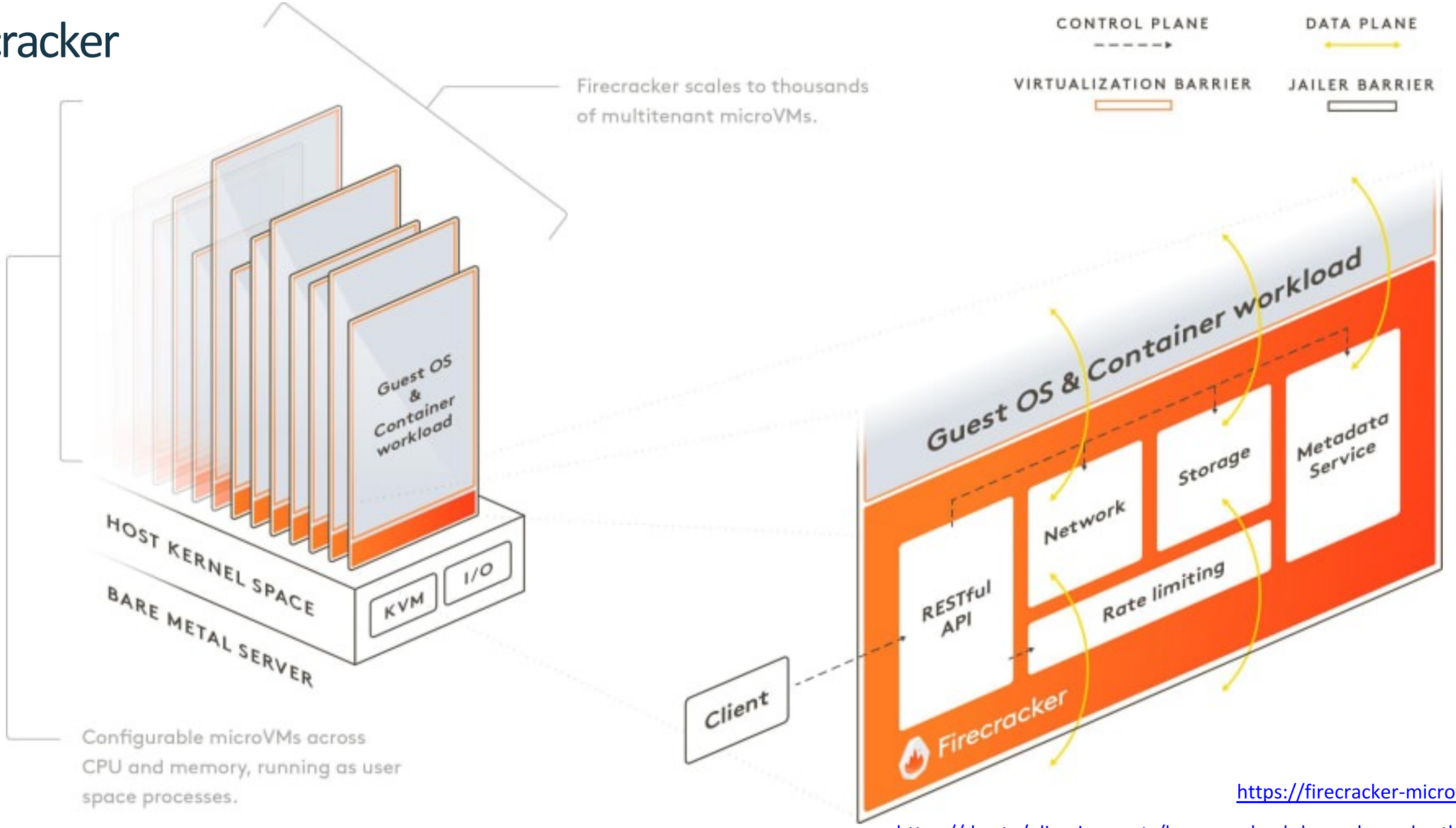
Demo #2

Implement our own Custom Runtime

How AWS Lambda Works?



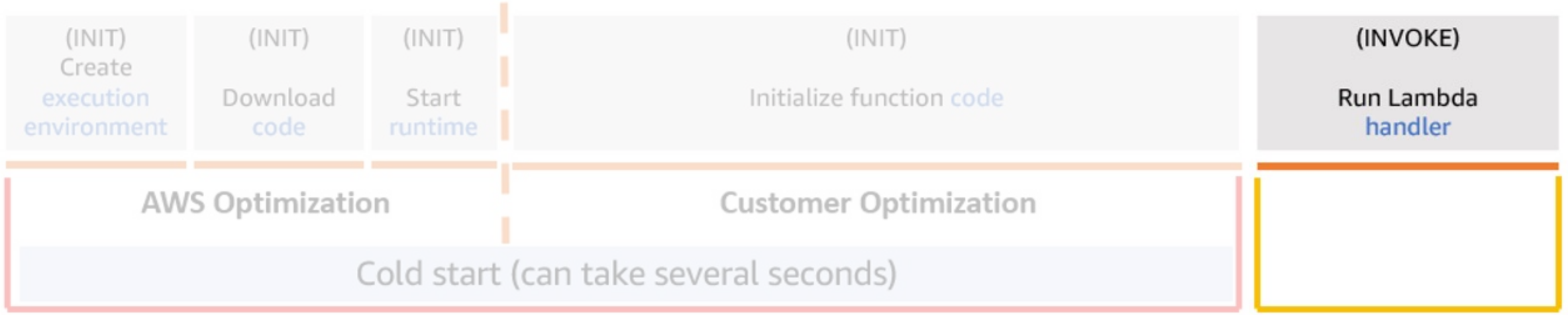
Firecracker



https://firecracker-microvm.github.io

https://dev.to/oliverjumpertz/how-aws-lambda-works-under-the-hood-2iae

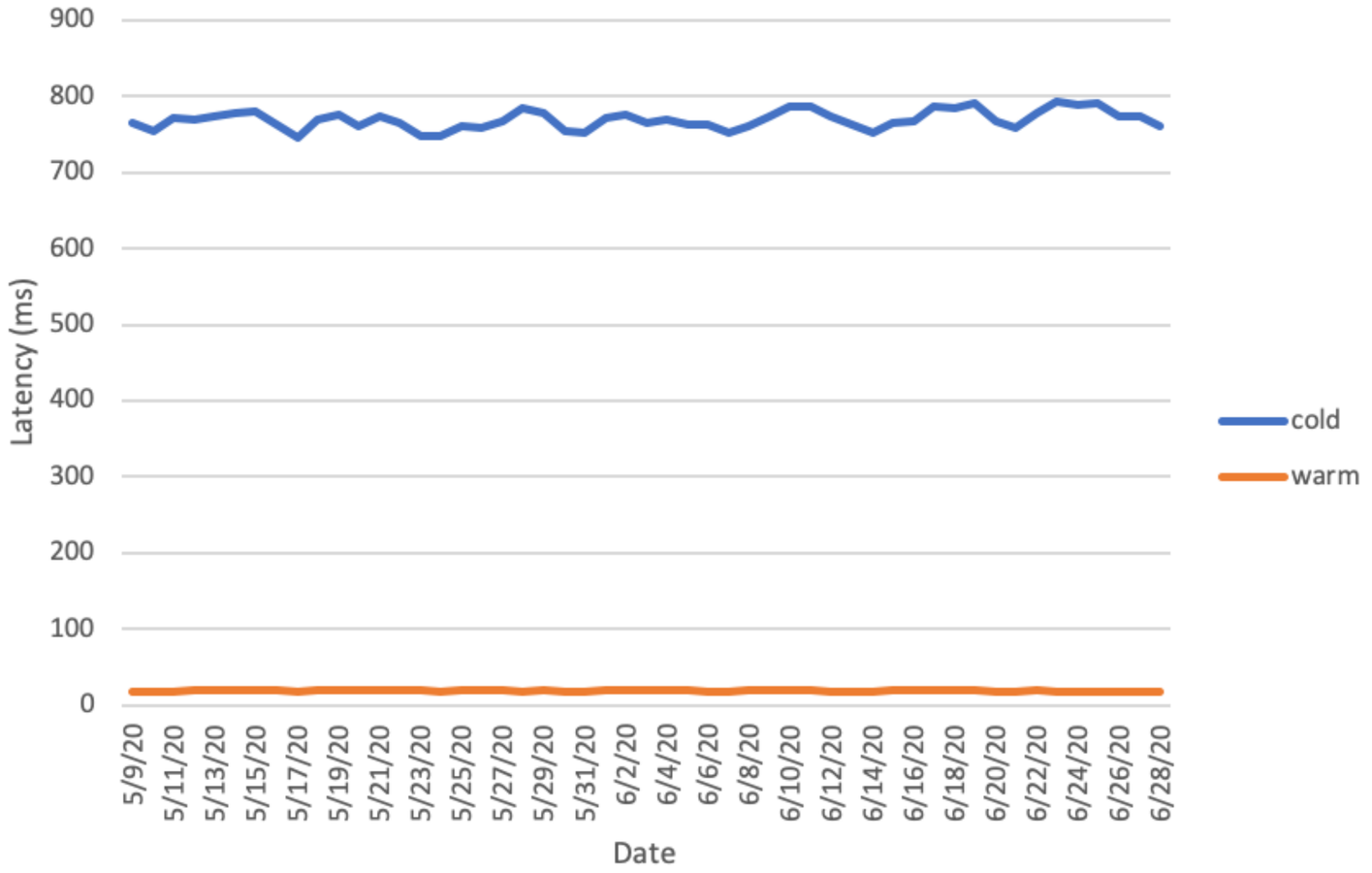
Cold Start



Warm Start

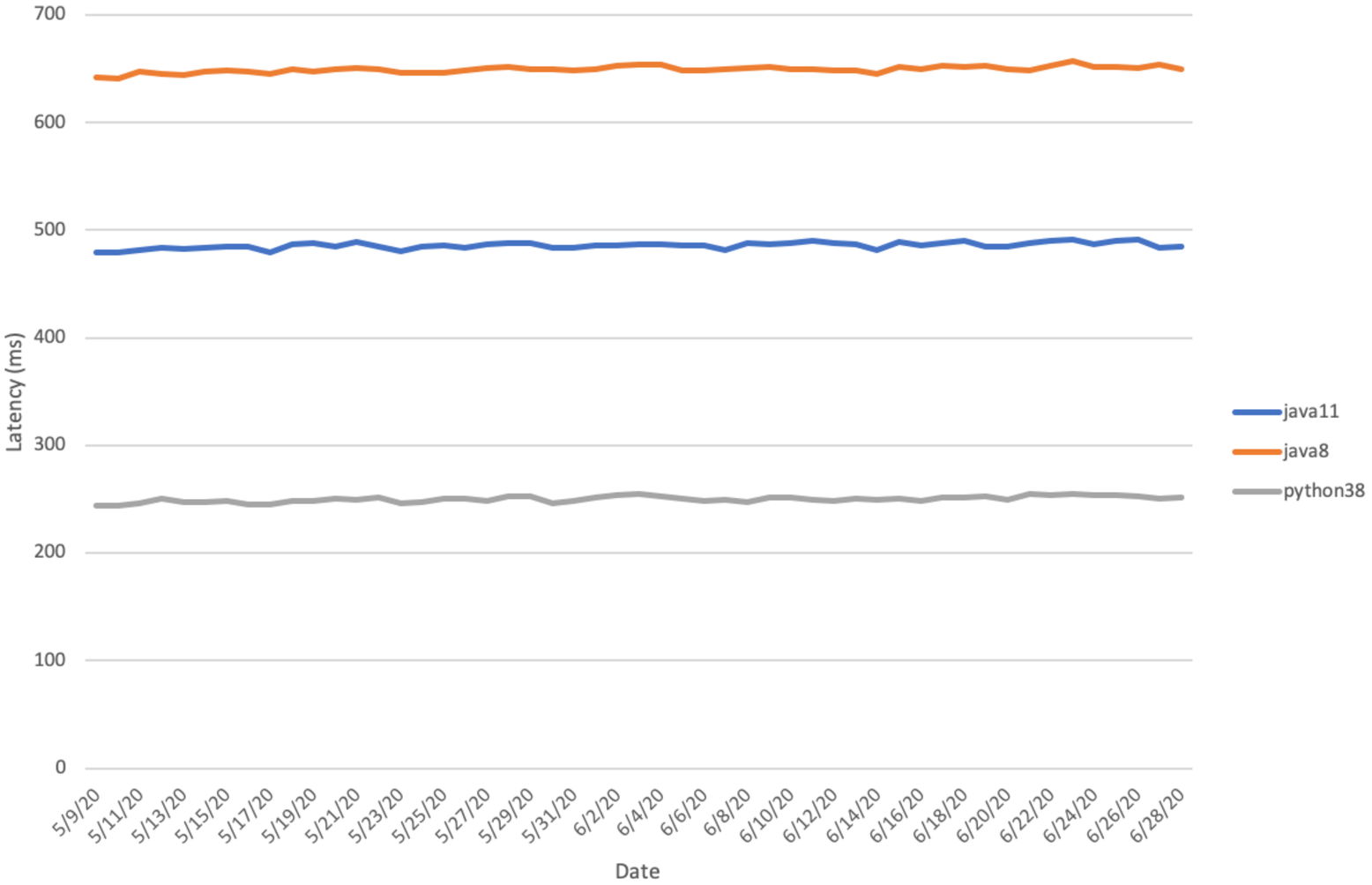
After the execution completes, the execution environment is frozen. To improve resource management and performance, the Lambda service retains the execution environment for a non-deterministic period of time. During this time, if another request arrives for the same function, the service may reuse the environment. This second request typically finishes more quickly, since the execution environment already exists and it’s not necessary to download the code and run the initialization code. This is called a “warm start”.

Java - Cold Start vs. Warm Start



https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda

Java vs. Other Runtime Cold Start

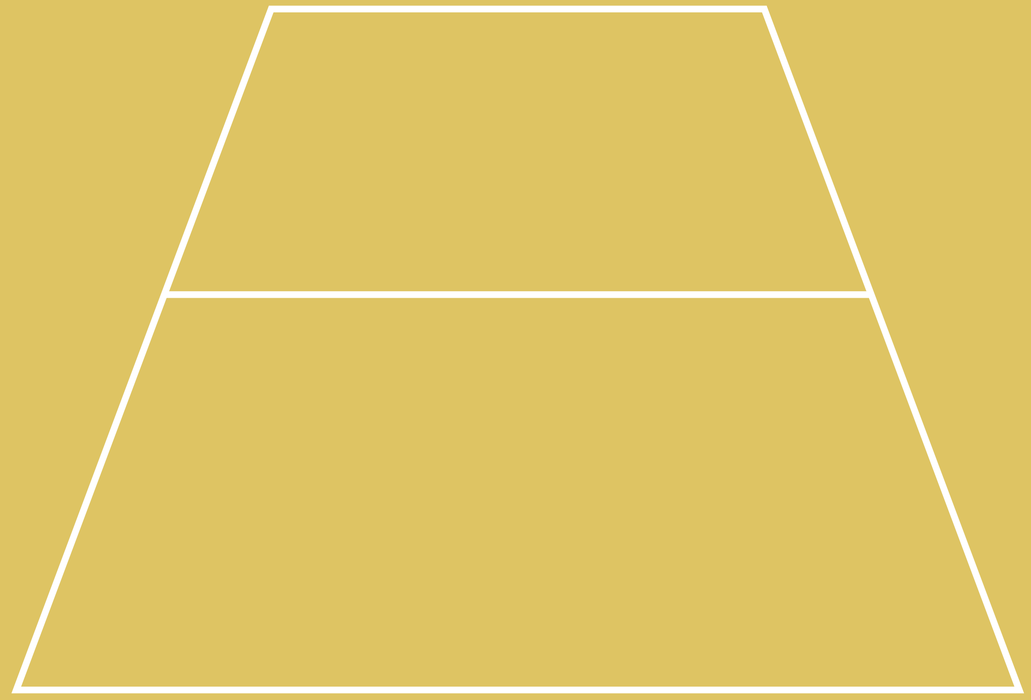


https://blog.symphonia.io/posts/2020-06-30_analyzing_cold_start_latency_of_aws_lambda

Demo #3.1

AWS Lambda w/ Java Runtime

Accelerate Serverless Java



Accelerate Serverless Java

Keep it warm

Ping Lambda function in every 5-15 minutes thus AWS Lambda Service won’t destroy the execution environment (i.e.: firecracker microVM)

Disadvantages

- Extra mechanism required (i.e.: additional functionality in lambda, scheduled trigger)
- Does not solve the issue in case of unpredictable number of lambda instances (scaling out)
- Dirty workaround

"I ping my lambda function every 15 minutes to make sure it's always warm"

Accelerate Serverless Java

Tiered Compilation

- Use JVM C1 compiler only (disable C2 and the required profiling)
- The C1 compiler is optimized for fast start-up time
- ~40% improvement

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

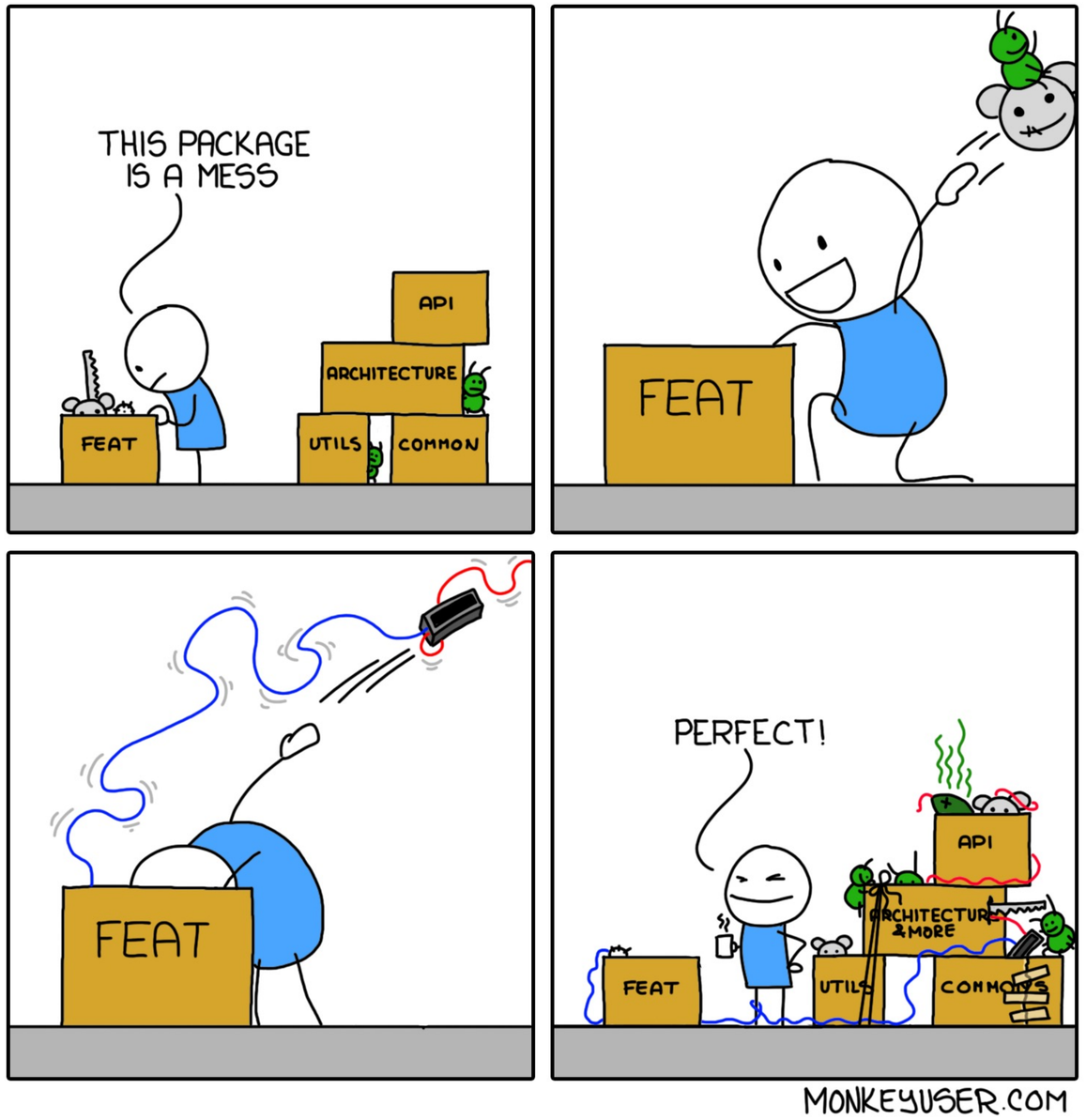
https://catalog.workshops.aws/java-on-aws-lambda/en-US/01-migration/02-optimizations/tiered

Accelerate Serverless Java

Improved Code

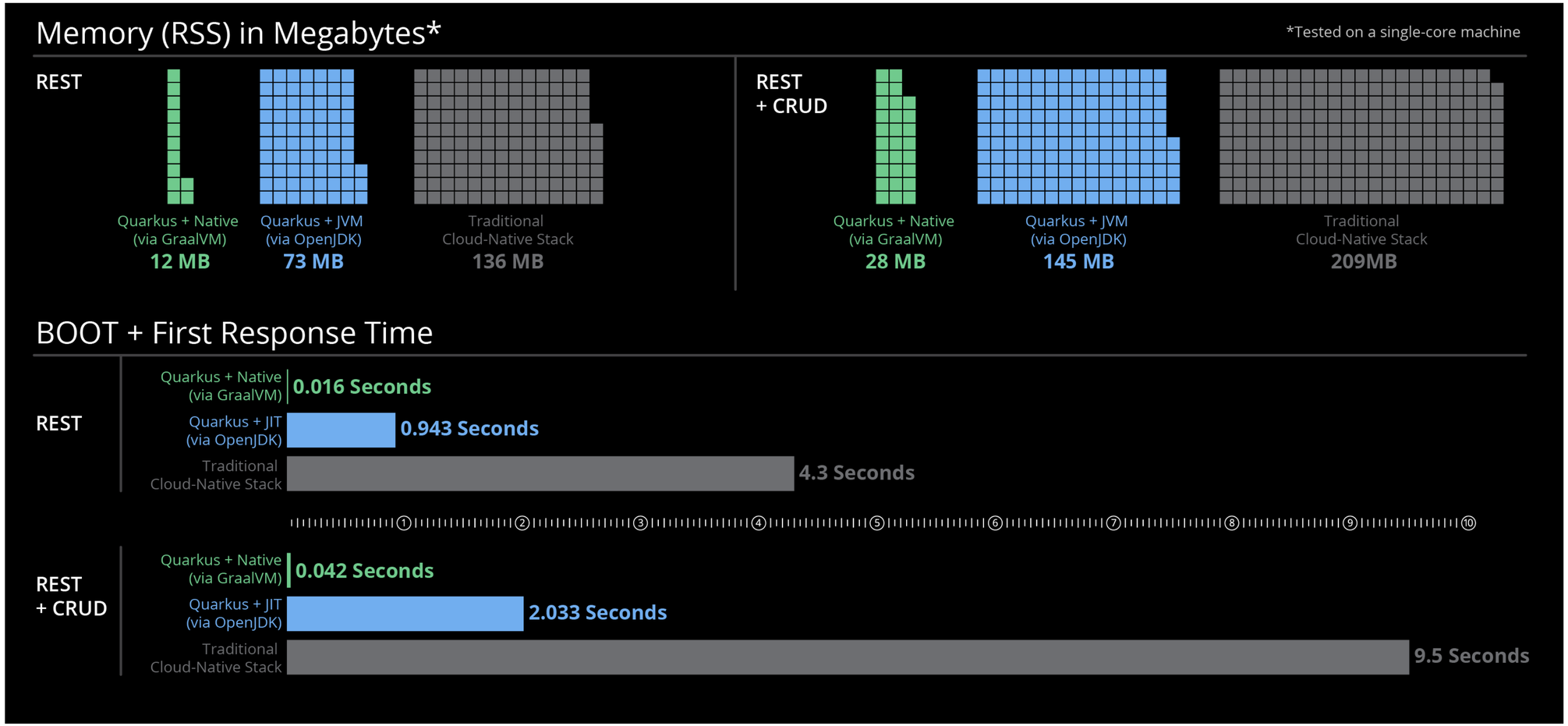
Write cloud-native Java application with serverless mindset.

1. Utilize CPU burst on startup (move everything to static) – decreased billed execution time but sacrificed performance during cold start
2. Use Compile Time frameworks such as Micronaut, Quarkus
3. Avoid reflections
4. Reduce dependencies as much as possible; Use lightweight dependencies;
5. ~25-35% improvement



Accelerate Serverless Java

AWS Lambda Custom Runtime + GraalVM



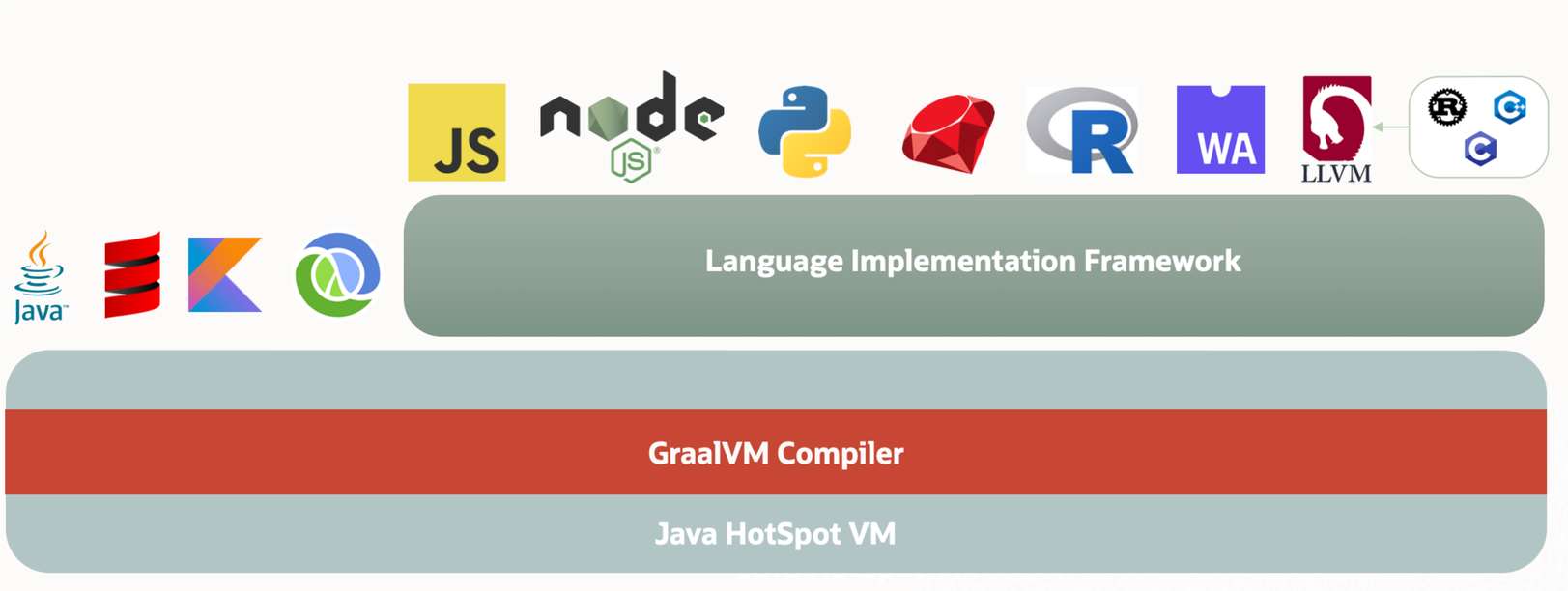
Accelerate Serverless Java

AWS Lambda Custom Runtime + GraalVM

GraalVM

GraalVM is an umbrella term that basically consists of three different technologies

1. Graal is a high-performance **JIT compiler** written in Java.
2. **Truffle**: GraalVM's language implementation framework – which makes GraalVM a polyglot virtual machine (supporting a large set of languages: JavaScript, Ruby, Python, R, C/C++)
3. **Native Image** is a technology to compile Java code ahead-of-time to a binary – a native executable.



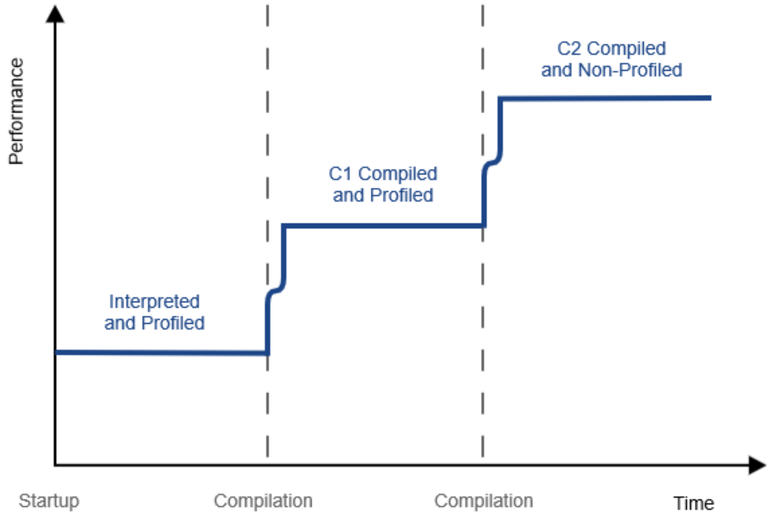
Accelerate Serverless Java

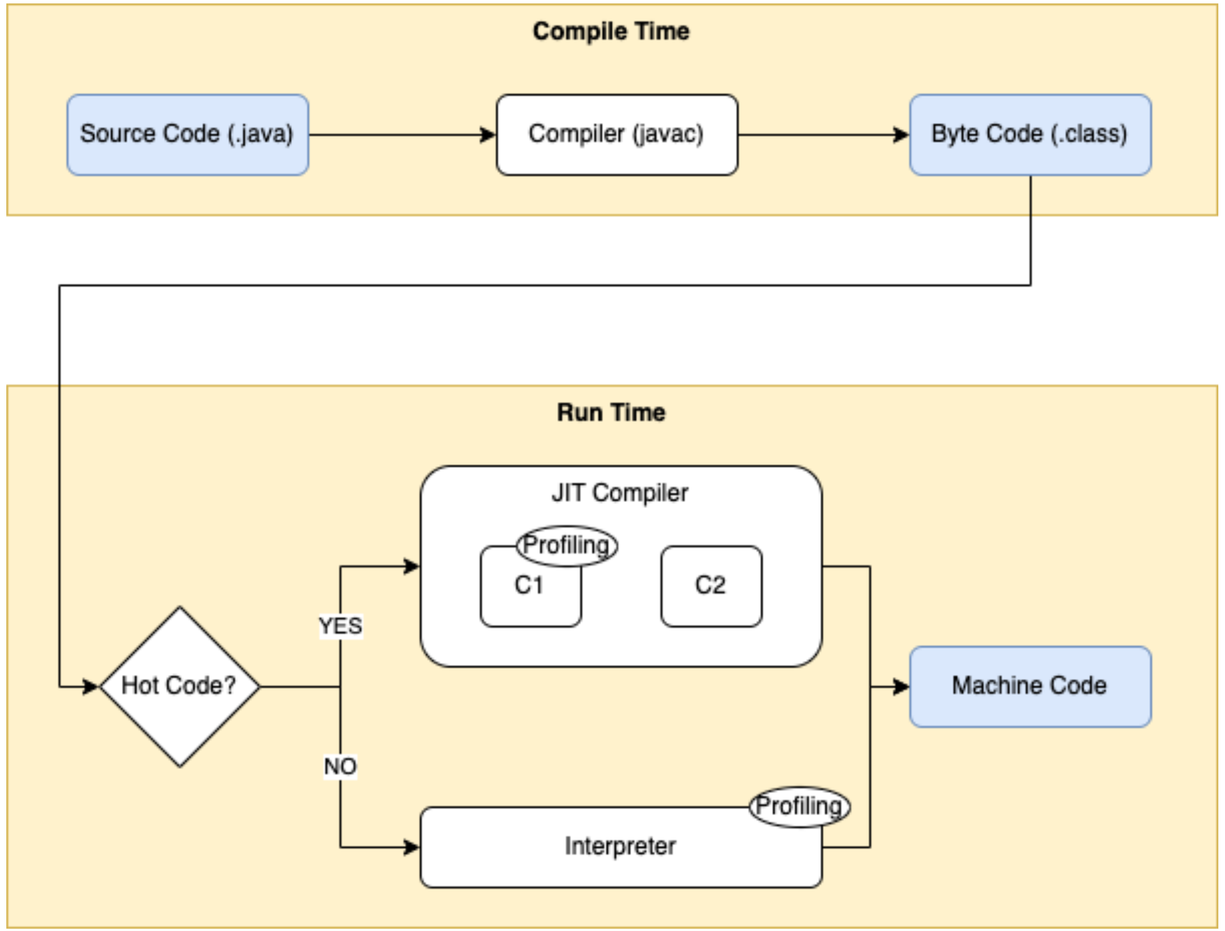
AWS Lambda Custom Runtime + GraalVM

JIT Compilation

The JVM interprets and executes bytecode at runtime. In addition, it makes use of the just-in-time (JIT) compilation to boost performance.

A JIT compiler **compiles bytecode to native code for frequently executed sections**.





https://www.baeldung.com/jvm-tiered-compilation

Accelerate Serverless Java

AWS Lambda Custom Runtime + GraalVM Native Compilation

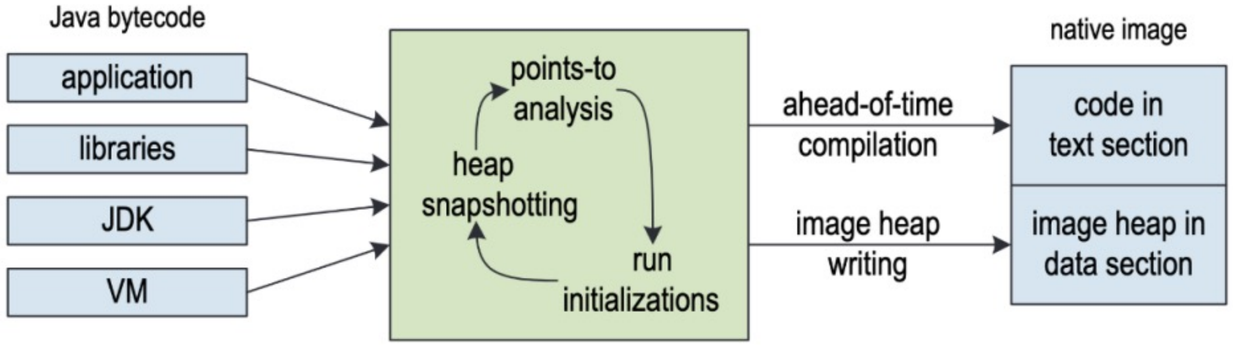
- Graal can run in both just-in-time and **ahead-of-time** compilation modes to produce native code.
- It allows statically compiling bytecode directly into machine code at **build time**

Pros

- Decreased artifact size: use only a fraction of the resources required by the JVM.
- Applications start in milliseconds - deliver peak performance immediately, no warmup.
- Reduced attack surface (No dynamic loading, close world)

Cons

- Close World Assumption *(solution: Tracking Agent)*
- Significantly (!) increased build time
- Platform dependent artifact



https://www.cesarsotovalero.net/blog/aot-vs-jit-compilation-in-java.html

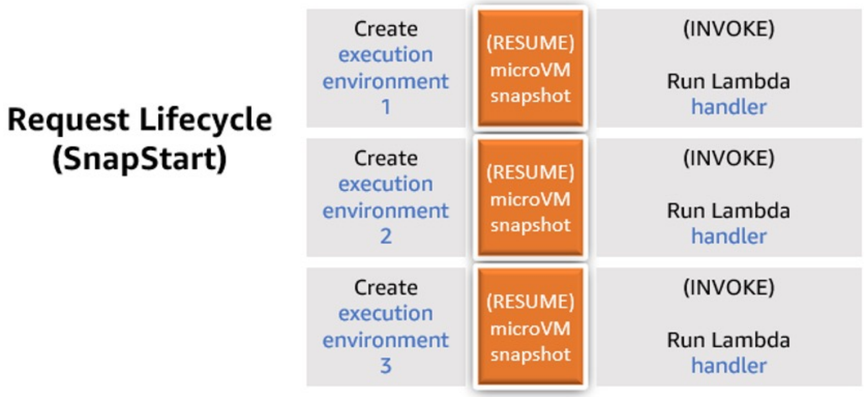
Accelerate Serverless Java

SnapStart

1. Announced at AWS re:Invent 2022 (**Dec 2022**)
2. Introduced **explicitly for Java runtime** in order to solve Java Cold Start issue out of the box
3. SnapStart helps you **improve** startup **performance** by up to **10x** at **no extra cost**
4. **Snapshot** created from the initialized function at deploy time. (i.e.: state after the cold start)
5. Lambda resumes the function from the cached snapshot upon invocation


```
--snap-start ApplyOn=PublishedVersions
```

```
publish-version
```



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

https://catalog.workshops.aws/java-on-aws-lambda/en-US/01-migratio

https://quarkus.io/blog/quarkus-support-for-aws-lambda-snapstart/

Accelerate Serverless Java

Comparison

Optimization	p0	p95	Improvement in %	SnapStart p0
No optimization	11.378	13.256		1.287
Tiered compilation	6.725	8.387	~40%	1.15
Lightweight dependencies	5.052	6.245	~25%	0.901
Function handler (Spring Cloud functions)	4.571	5.606	~10%	0.779
Micronaut JVM	3.499	4.284	~25%	0.765
No framework	1.850	2.588	~45%	0.705
GraalVM (Spring Native)	0.683	0.890	~65%	n/a

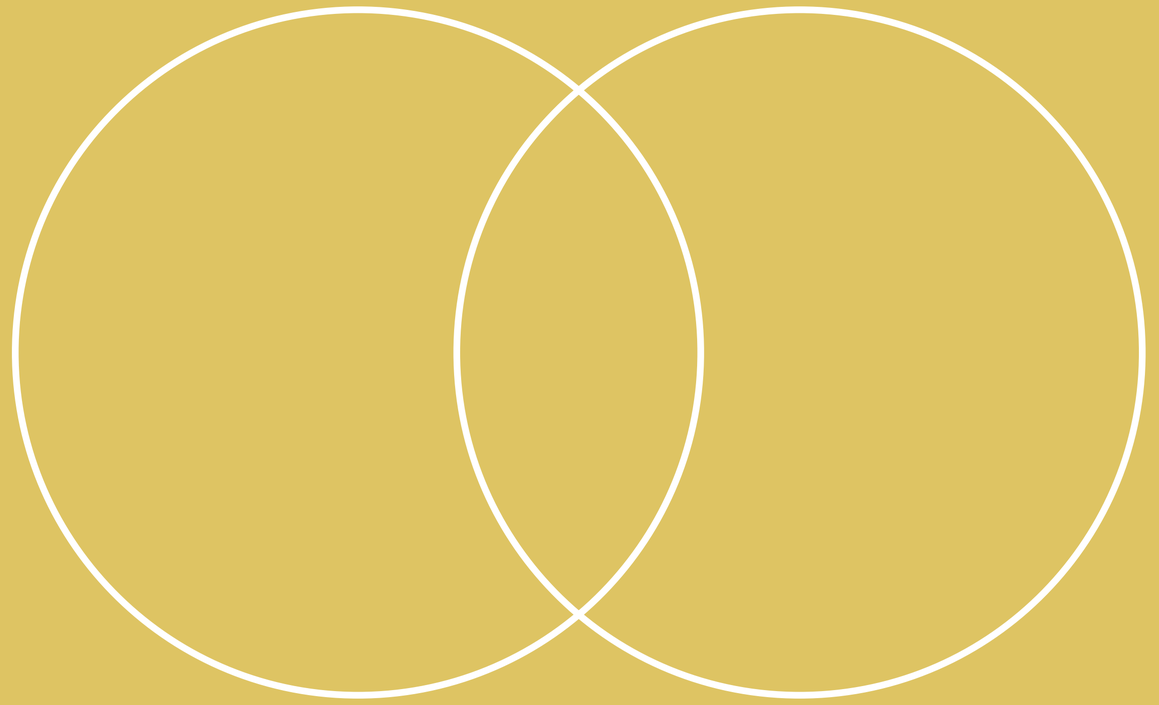
https://catalog.workshops.aws/java-on-aws-lambda/en-US/01-migration/results

Demo #3.2

Enabling SnapSt

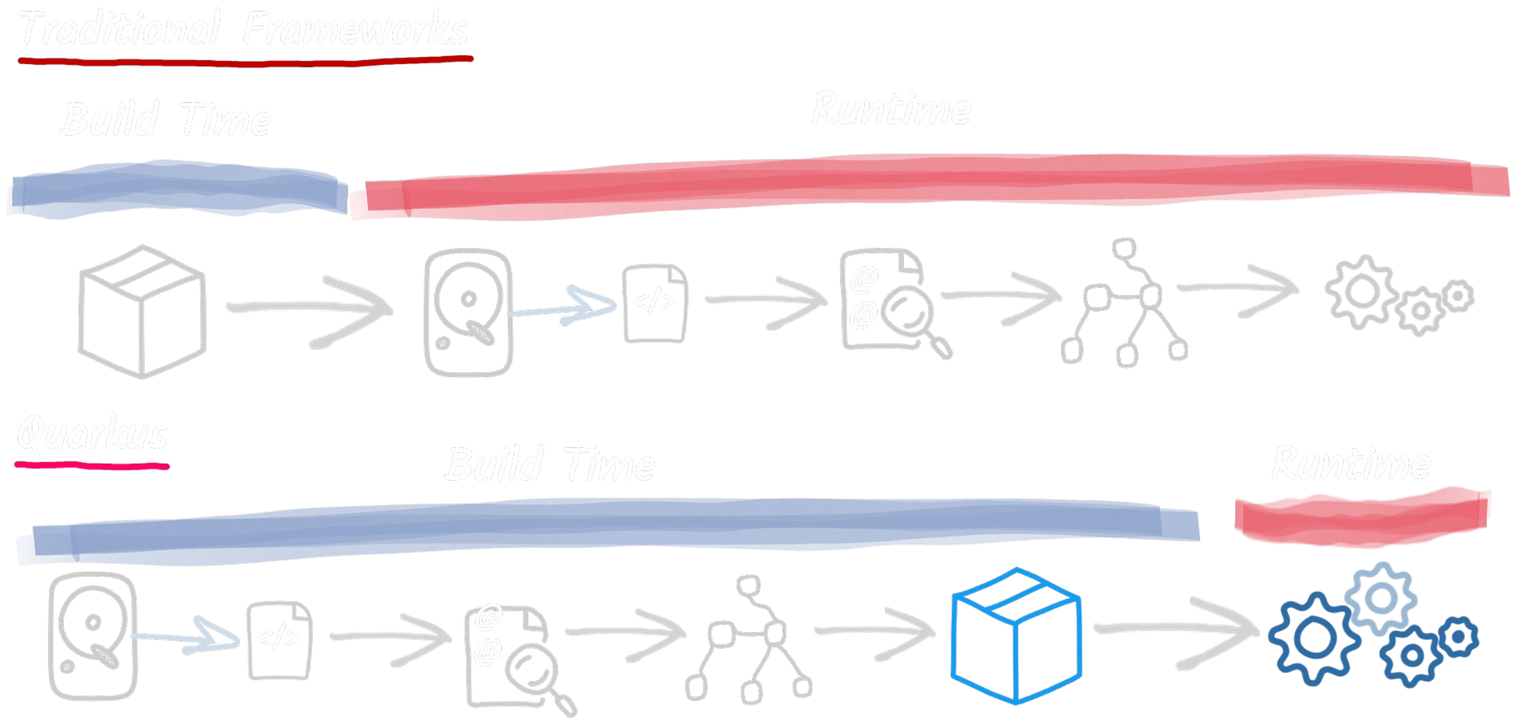
Compile Time Frameworks

Micronaut and Quarkus



Compile Time Framework – Shift left

Both frameworks are optimised for low memory usage and fast startup times. Do at build-time what traditional frameworks do at runtime. Generate meta-information (in form of additional byte codes) at build time about your code.



https://quarkus.io/container-first/

https://micronaut.io/2023/03/21/micronaut-framework-code-generation/

Build Time Actions

What artifacts does **Micronaut** generate as bytecode?

1. **Bean definitions** to power the Micronaut Framework dependency injection engine.
2. **Bean Introspections** to power features such as **reflection-free** serialization.
3. **Proxies** to power AOP features.
4. **Bean Factories**. For example, Micronaut Kubernetes and Micronaut Oracle Cloud generate factories for external SDK Clients from an annotation processor.

What artifacts does **Quarkus** generate as bytecode?

1. Custom **proxies** instead of dynamic proxies
2. **Replace reflection** calls with regular invocations
3. Arc (Quarkus DI framework): eliminates reflections, **evaluates DI injection graph at build time**

GraalVM **Native Executable support** has been an **essential part of the design** for Quarkus from the beginning

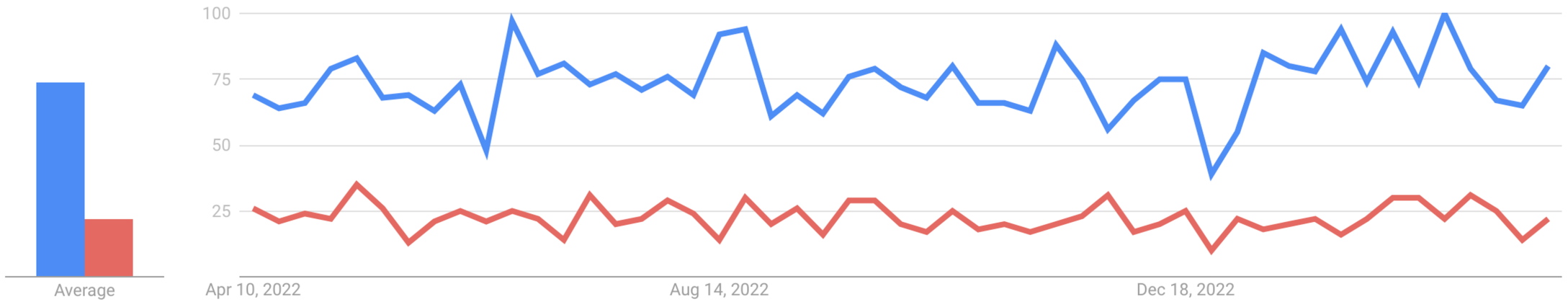
https://quarkus.io/container-first/

https://micronaut.io/2023/03/21/micronaut-framework-code-generation/

https://quarkus.io/guides/building-my-first-extension

Comparison

Google Trends



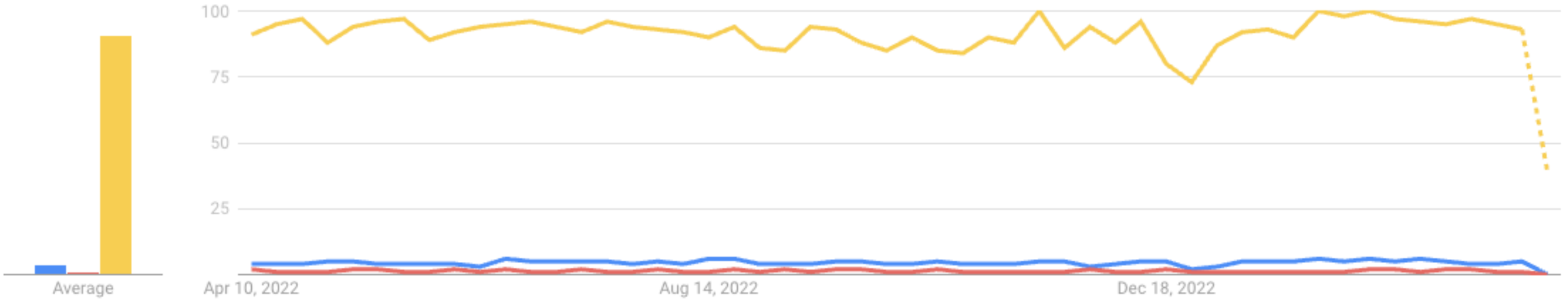
Quarkus
Computer program

Micronaut
Search term

Worldwide | Past 12 months | Programming | Web Search

Comparison

Google Trends / Part II



Comparison

Github

Micronaut

1. **Nr of Releases**: 154
2. **Contributors**: 365
3. **Stars**: 5700
4. **Forks**: 971
5. **License**: Apache 2.0

Initial Release Date: May 2018

x2

Quarkus

1. **Nr of Releases**: **226**
2. **Contributors**: **785**
3. **Stars**: **11600**
4. **Forks**: **2200**
5. **License**: Apache 2.0

Initial Release Date: March 2019

https://github.com/micronaut-projects/micronaut-core

https://github.com/quarkusio/quarkus

Demo #4.1

AWS Lambda w/ Quarkus

https://quarkus.io/guides/amazon-lambda

Demo #4.2

AWS Lambda w/ Micronaut

https://quarkus.io/guides/amazon-lambda

Performance Comparison of the Hello World Lambda

```
Micronaut
Duration: 395.65 ms Billed Duration: 810 ms Memory Size: 256 MB Max Memory Used: 91 MB Init Duration: 414.21 ms
Duration: 122.96 ms Billed Duration: 123 ms Memory Size: 256 MB Max Memory Used: 92 MB
Duration: 132.69 ms Billed Duration: 133 ms Memory Size: 256 MB Max Memory Used: 93 MB
Duration: 392.77 ms Billed Duration: 393 ms Memory Size: 256 MB Max Memory Used: 93 MB

Quarkus
Duration: 607.88 ms Billed Duration: 839 ms Memory Size: 256 MB Max Memory Used: 63 MB Init Duration: 230.68 ms
Duration: 126.73 ms Billed Duration: 127 ms Memory Size: 256 MB Max Memory Used: 64 MB
Duration: 408.96 ms Billed Duration: 409 ms Memory Size: 256 MB Max Memory Used: 65 MB
Duration: 125.66 ms Billed Duration: 126 ms Memory Size: 256 MB Max Memory Used: 65 MB
```

https://quarkus.io/container-first/

https://micronaut.io/2023/03/21/micronaut-framework-code-generation/

What we haven’t touched…

AWS Lambda Deployment Alternative

- Deploying lambda functions as container images (from ECR)

Additional Frameworks and Supports

- Spring Boot 3 and Spring Framework 6 official native support (since September 2022)
- Quarkus Funqy (Cloud agnostic Functions)

Building Event Driven Application

- Choreography vs. Orchestration
- Event Driven Architecture
- AWS Step Functions – Serverless workflow orchestration
- Lambda proxy integrations in API Gateway
- And much more: AWS Event Bridge, SQS and SNS

THANKS THANKS THANKS

THANKS THANKS THANKS

THANKS THANKS THANKS

beyond

beyond