
AWS
re:Invent
NOV. 28 – DEC. 2, 2022 | LAS VEGAS, NV

DOP315

# Sustainability in the cloud with Rust and AWS Graviton


Emil Lerch (he/him)

Principal DevOps Specialist
AWS

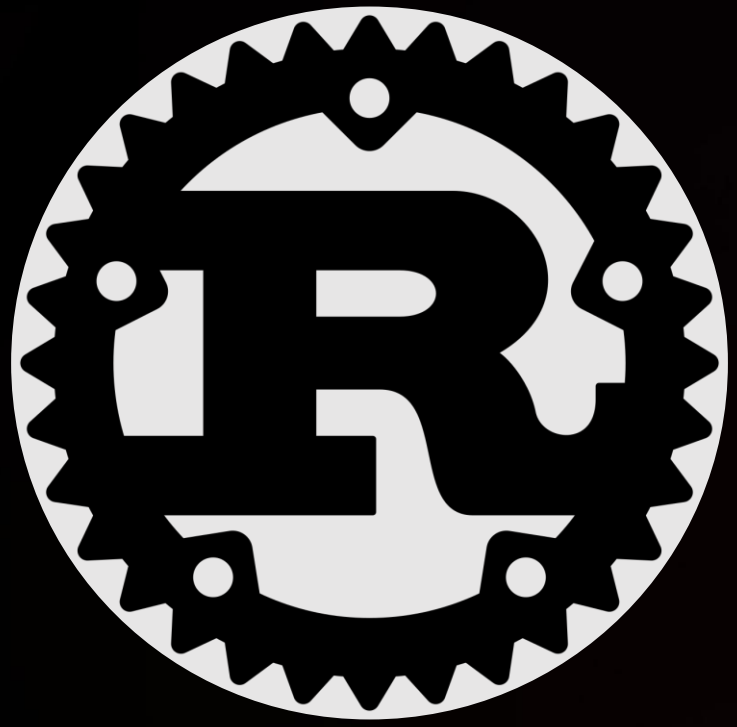
Esteban Küber (he/him)

Principal Engineer/Rust Compiler Team Member
AWS

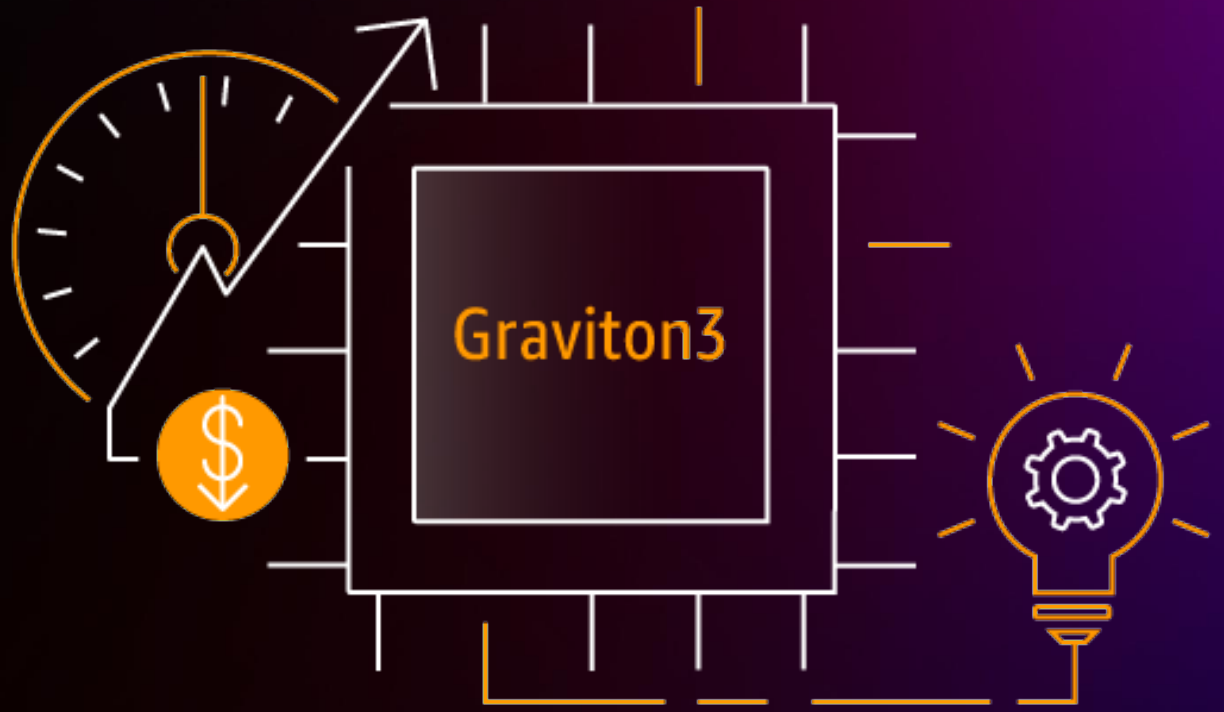


© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust and AWS Graviton



Rust logo by Mozilla / CC BY 4.0




aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust unofficial ethos

**The problem is hard,**
**but if we restrict the scope,**
**we can statically check it!**

aws

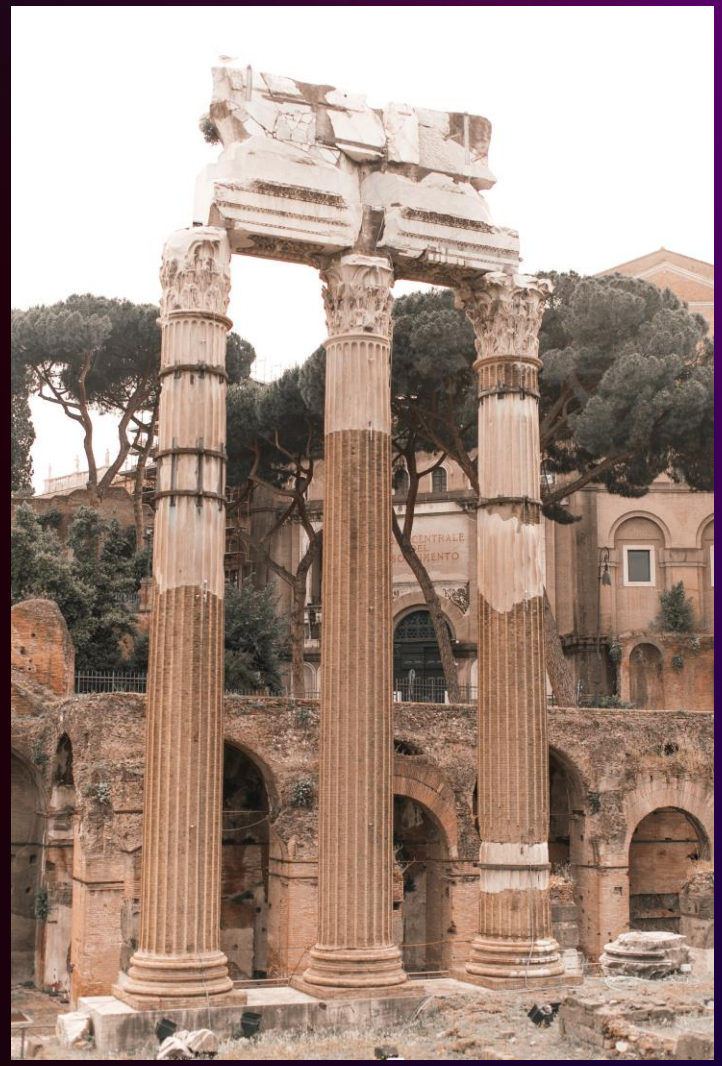
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust

Performance

Reliability

Productivity



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust and sustainability

## Performance

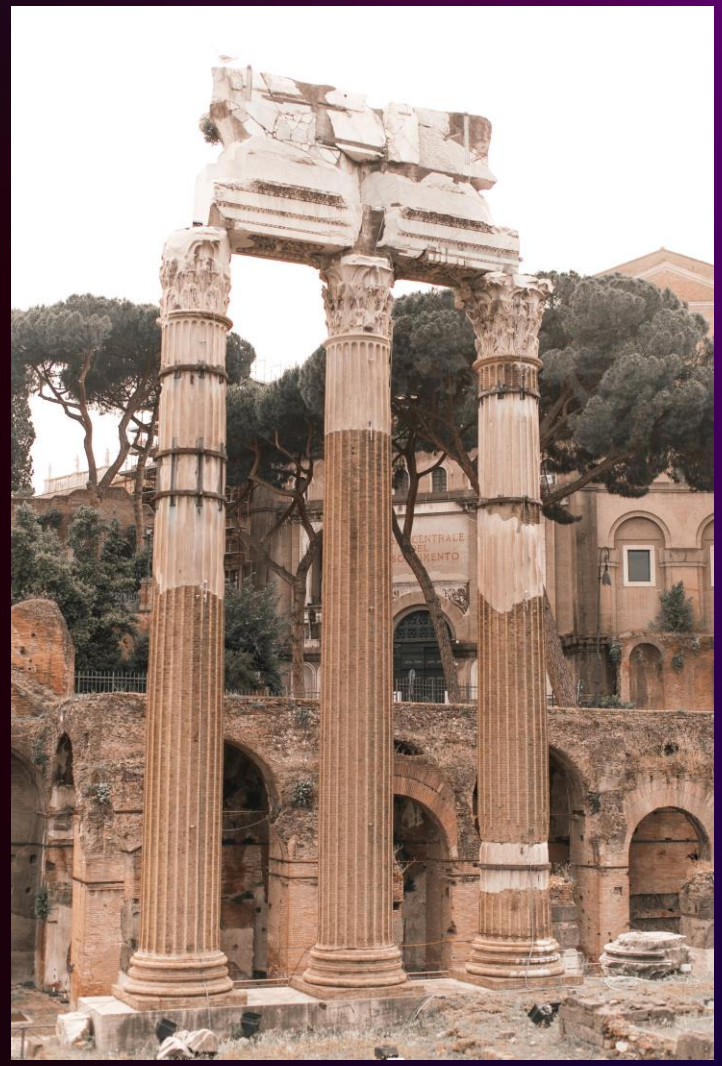
Faster code/fewer resources

## Reliability

Less retries/fewer resources

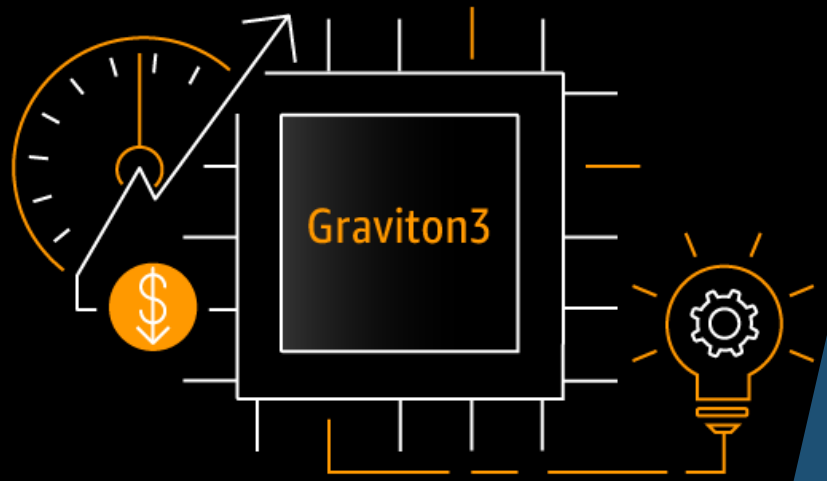
## Productivity

Faster coding/less testing/fewer resources



aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.






# AWS Graviton

The latest generation of AWS-designed processors built for the cloud

---

AWS Graviton3–based Amazon EC2 instances **use up to 60% less energy for the same performance** than comparable instances

Graviton3 server design **increases sockets/rack 50%** over other Amazon EC2 servers

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust sustainability

| Language | Energy usage | Memory usage |
|---|---|---|
| C# | Low | Medium |
| Go | Medium | Low |
| Java | Low | Medium |
| Node | Low | High |
| Python | High | Low |
| Ruby | Medium | Medium |
| Rust | Low | Low |

aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust and Java

## Rust

- No runtime*
- Monomorphic by default
- Stack allocation by default, need to box to allocate on the heap
- Only AOT
- Hygienic macros
- Async Rust
- Safe/Unsafe

## Java

- Java Virtual Machine – GC
- Dynamic dispatch (vtable)
- Heap allocation (except for primitive types)
- Primarily JIT
  - Interpreter, C1, C2
  - Graal AOT
- Reflection
- Project Loom

aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Memory layout

## Rust

- Stack allocated by default
- Heap allocation opt-in
- Type-erased trait objects on the heap opt-in
- Zero-cost abstractions

## Java

- Objects are heap allocated
- JIT can perform escape analysis and hoist object fields onto CPU registers or the local stack
- Heap-allocated objects have header overhead

aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust trade-offs

- Allocation on the heap is not cheap or free
- Rust’s type system encourages unique ownership
  - Aliasing the same memory is discouraged
  - Requires a new programming style based on ownership – some old patterns don’t work
- Lack of runtime reflection
- Very limited dynamic library loading
- “Difficult” language

aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# JVM trade-offs

- Garbage collection
  - Allocating objects is very fast
  - Easy memory management, but not a panacea
- Native compilation
  - Ship bytecode, run it on any platform with a JVM
  - C2 is a profile-guided optimizing compiler
  - Compilation contributes to startup time
  - Graal-native – provides a fully native precompilation path

aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# An experiment

We decided to replicate the “magic-wormhole” application

- https://github.com/magic-wormhole/magic-wormhole
- Used to securely and easily move files from one system to another

Rationale

- Let developers choose an app they like, and reimplement it, measure, and observe the process

Capturing the developer experience

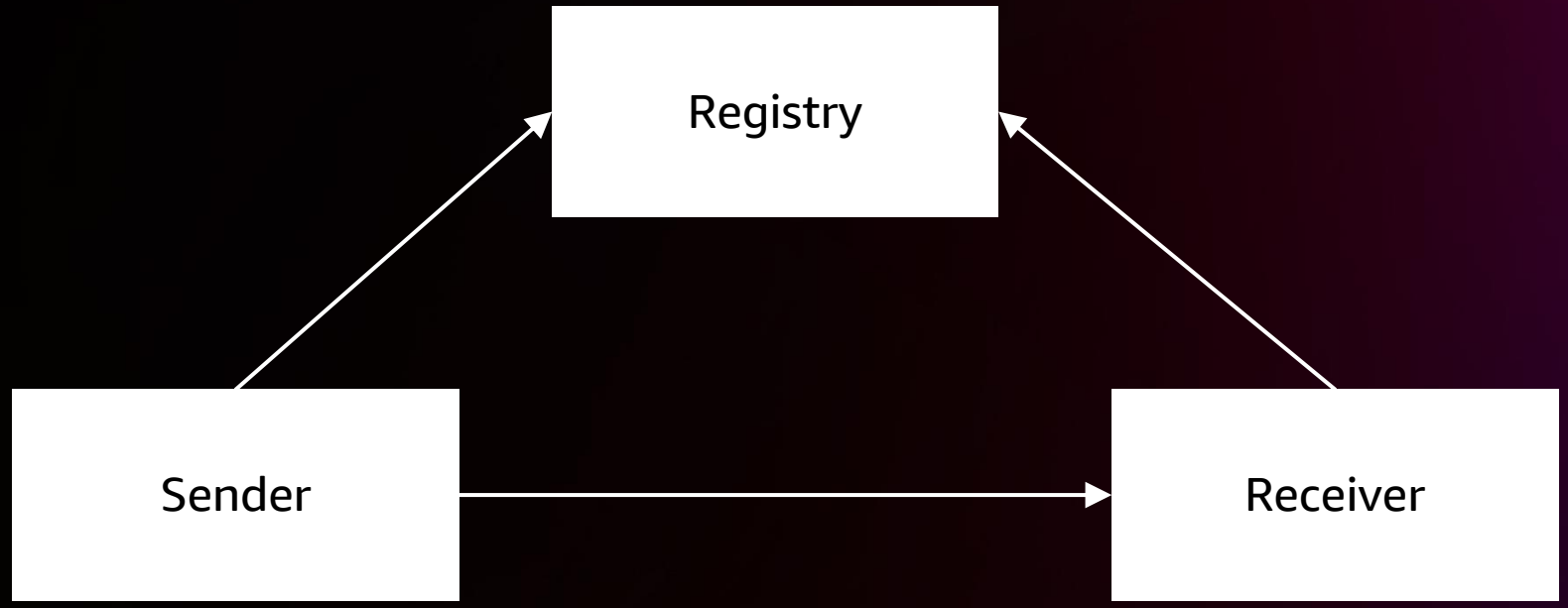
- What tools could be used?
- What are the languages’ semantic differences?
- What libraries could be used?

How did they each perform?

aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# An experiment

Magic-wormhole




© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# An experiment

- Both setups would implement the following:
  - A web service running the registry
  - A receiver process that listens on a TCP port and writes to disk
  - A sender process that connects to the registry, connects to the receiver, reads a file, and writes to the TCP connection
  - Timeboxed development to <2 days total effort (Ha!)
  - Ideally, limit the code to less than 300 lines

aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust

```
let mut contents = vec![0; BUF_SIZE];
…
let n = stream.read(&mut contents).await?;
tokio::io::copy(&mut &contents[..n], &mut file).await?;
```

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Experiment ecosystems

## Rust

- Tokio
- Warp
- Clap
- Serde
- Rayon
- Cargo

## Java

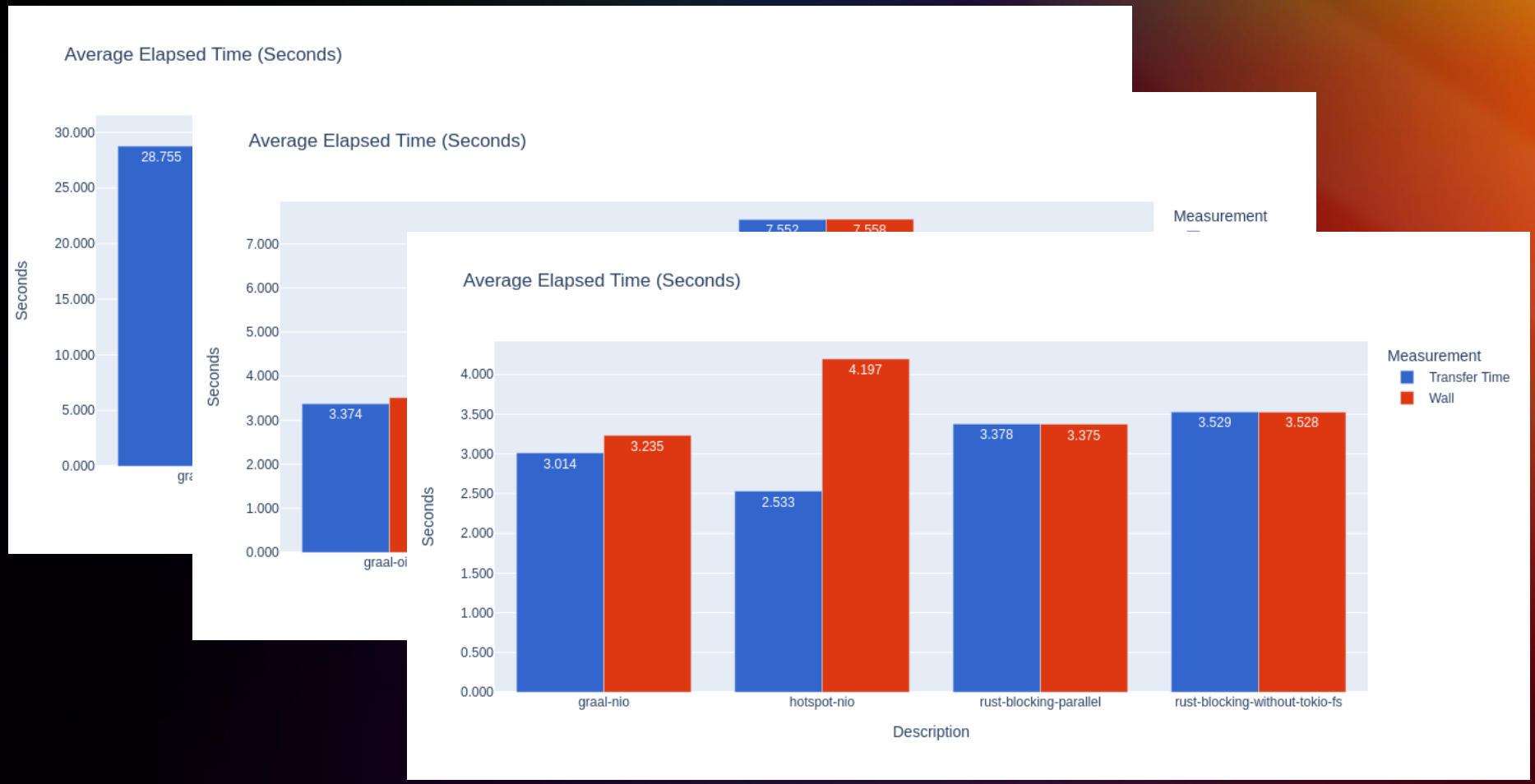
- Micronaut
- Jackson
- Netty
- Maven

- Hotspot 18.0.1
- GraalVM 22.1.0 Java 17 CE

aws

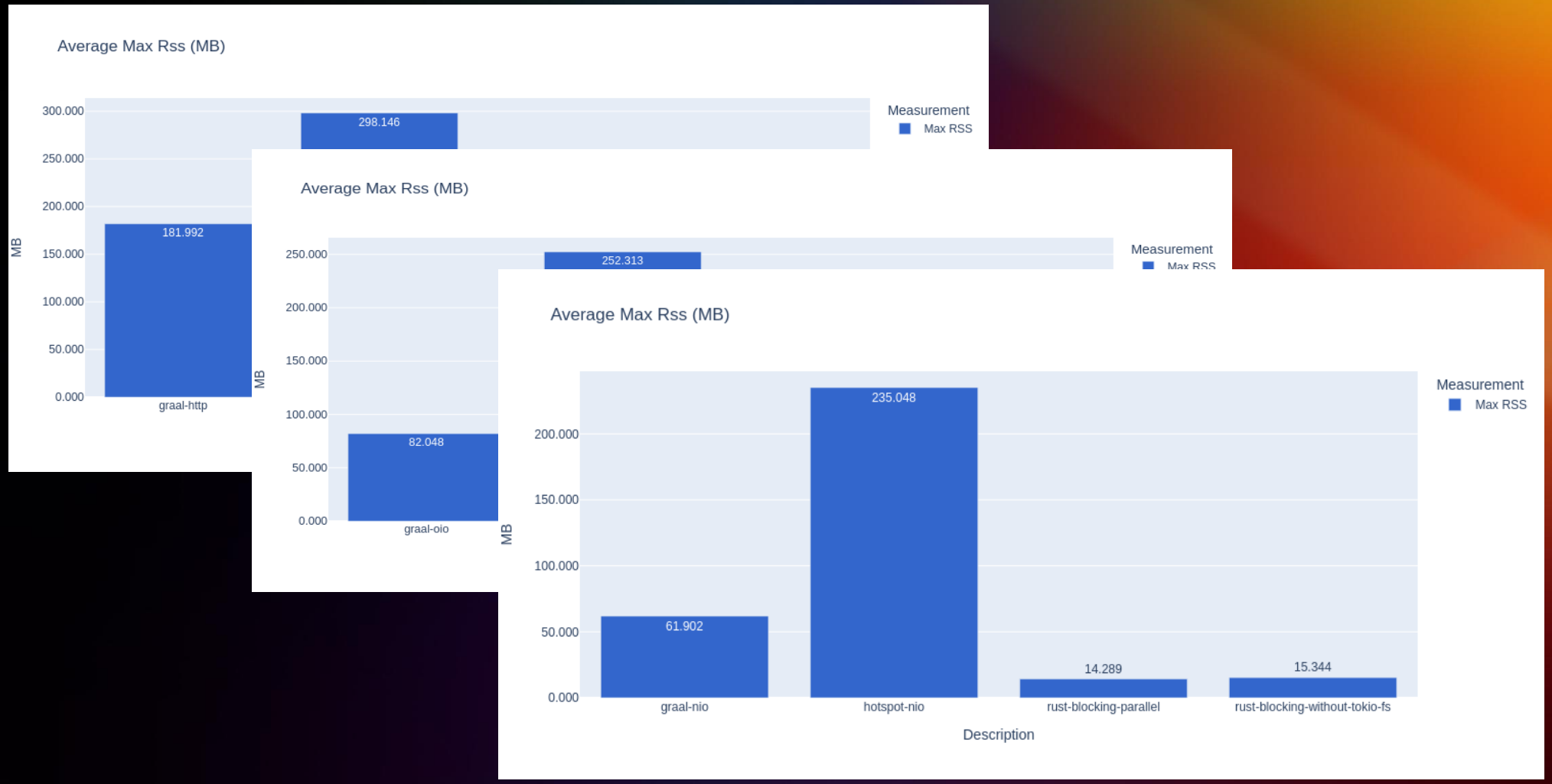
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Time




© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Memory




© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# By the end

## We built 3 versions of the Rust app

1. We dropped the Tokio runtime after interacting with the registry
2. We added support to optionally use sendfile
3. We added directory-sending support behind a thread pool

## And we built 4 versions of the Java app

1. Initial, HTTP-based version
2. stdlib I/O
3. NIO
4. App teardown was being slowed by OS disk syncing

   We switched to using tempfs for the receiver's target director

aws

© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Thinking of the overall ecosystem

- We found the application easy to write in both languages
- Using the hotspot JVM was slow to start but fast as it warmed up
- Don’t ignore deferred disk I/O impact
- Performance was faster than expected for Java relative to Rust
  - We were surprised by the FS performance with async Tokio, but it is an expected trade-off
- Rust was significantly better at memory utilization than Java, but we expect more amortization on a multithreaded app
- Java provided more mature libraries

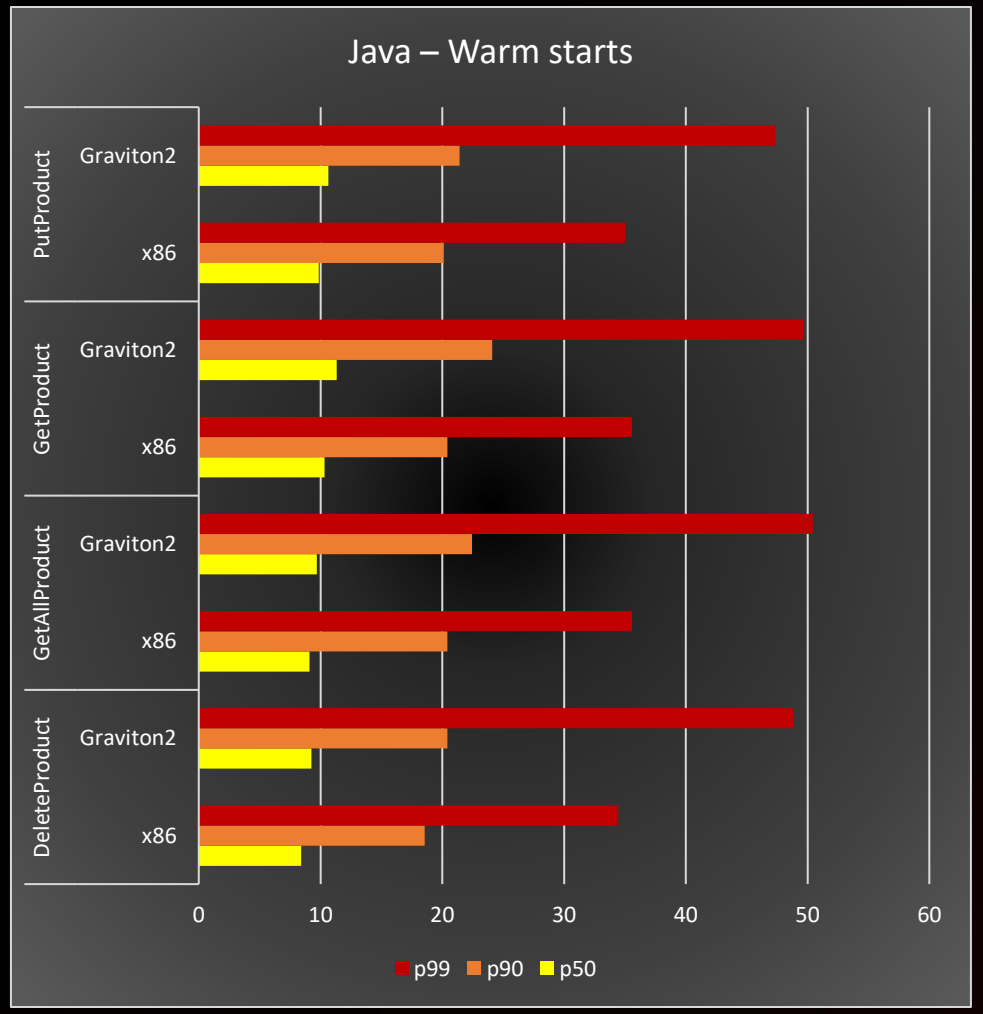
aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

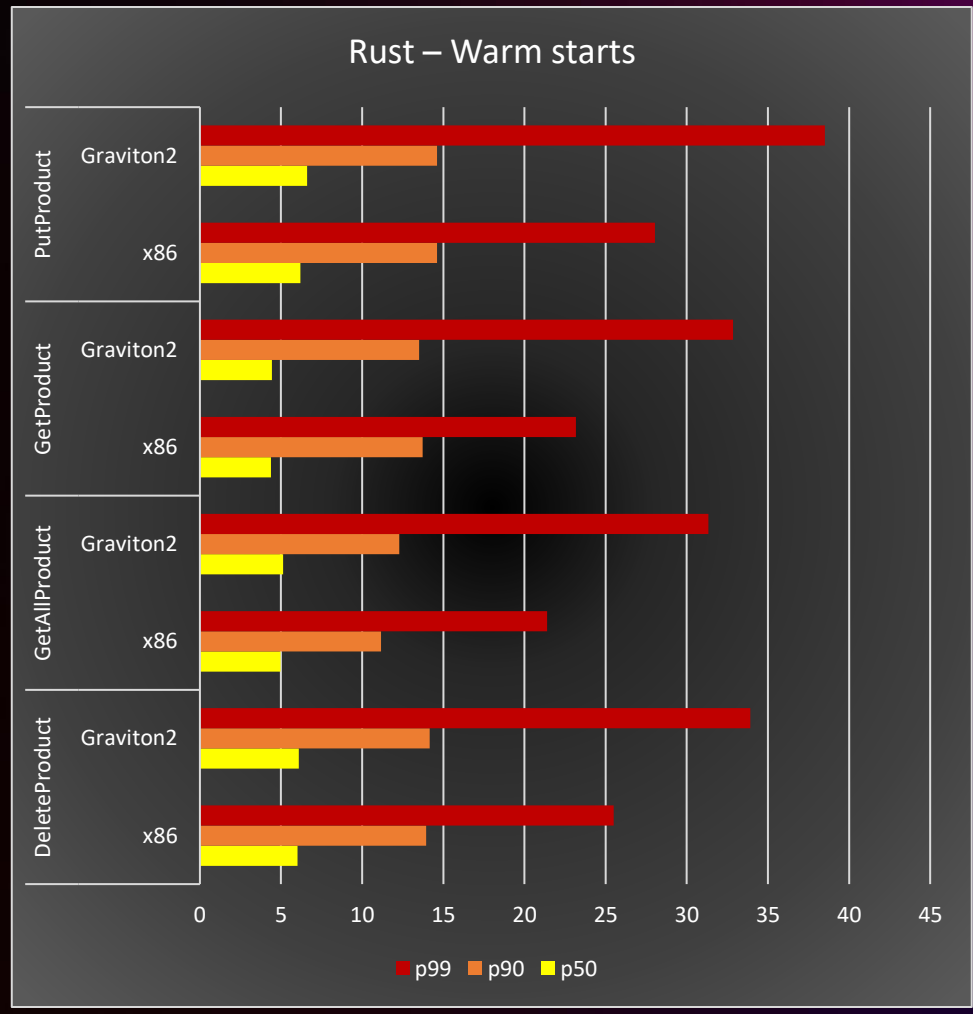
# Rust and Graviton in the cloud

- What differences do we see using Rust and Graviton on AWS Lambda?
  - Performance
  - Energy efficiency
  - Cost
- AWS sample application demonstrates identical application in several languages
- We used:
  - Java: https://github.com/aws-samples/serverless-graalvm-demo
  - Rust: https://github.com/aws-samples/serverless-rust-demo
- Java sample was changed to 128 MB functions to match Rust
- Both samples needed minor changes when switching between x86 and Graviton

aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

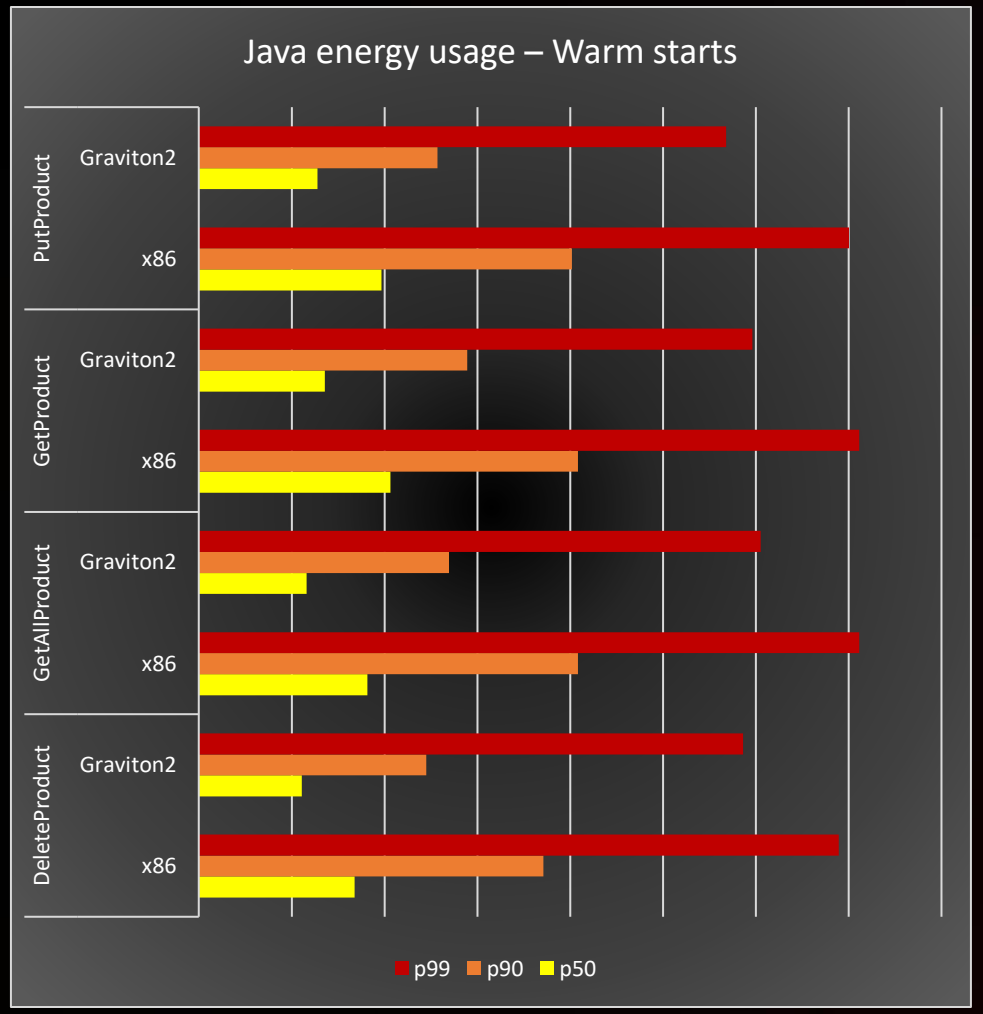
# Rust and Graviton in the cloud: Performance

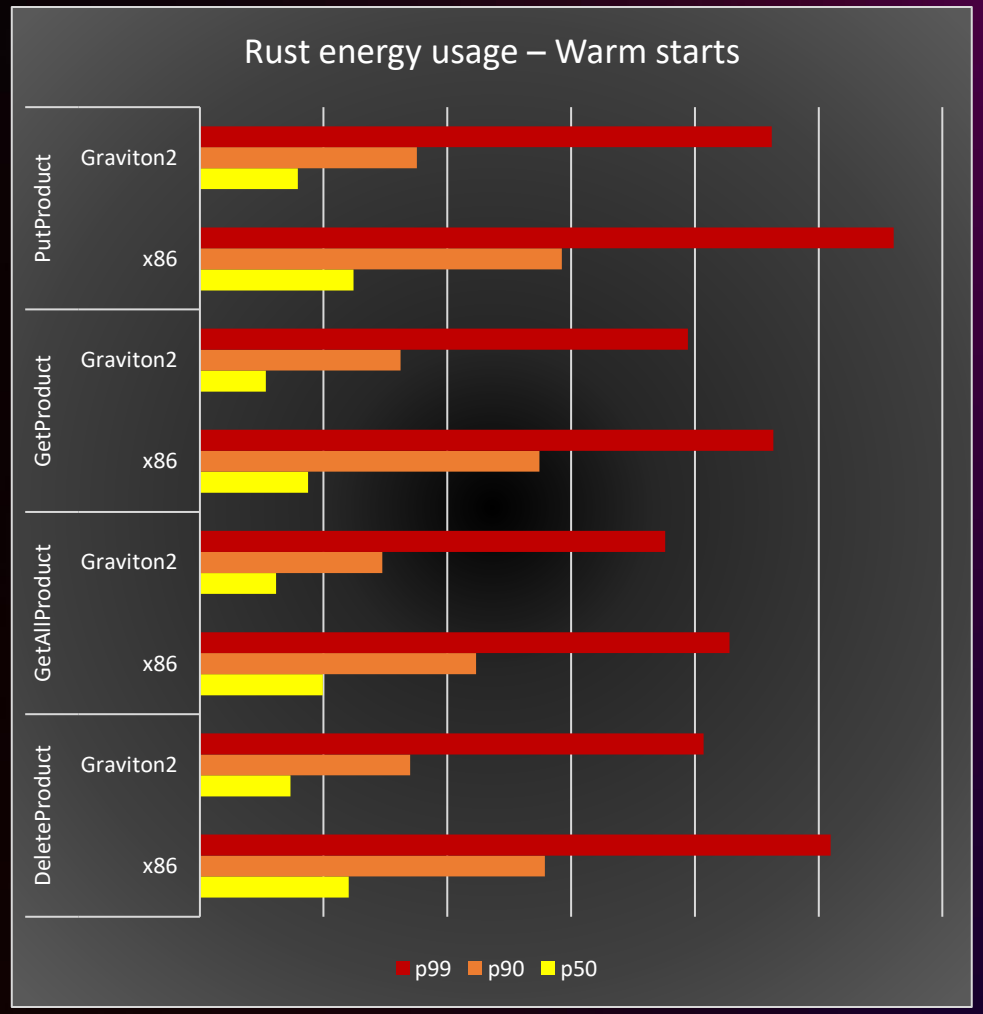



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

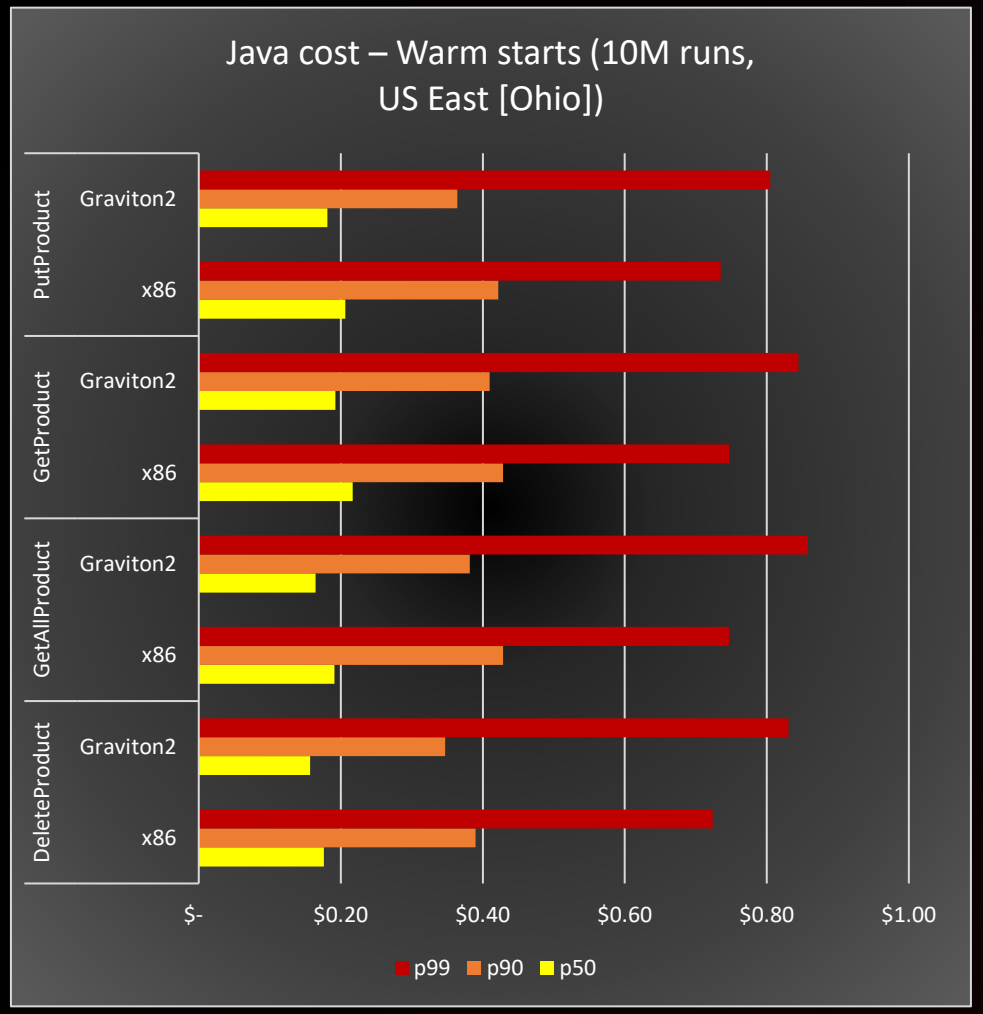
# Rust and Graviton in the cloud: Energy usage

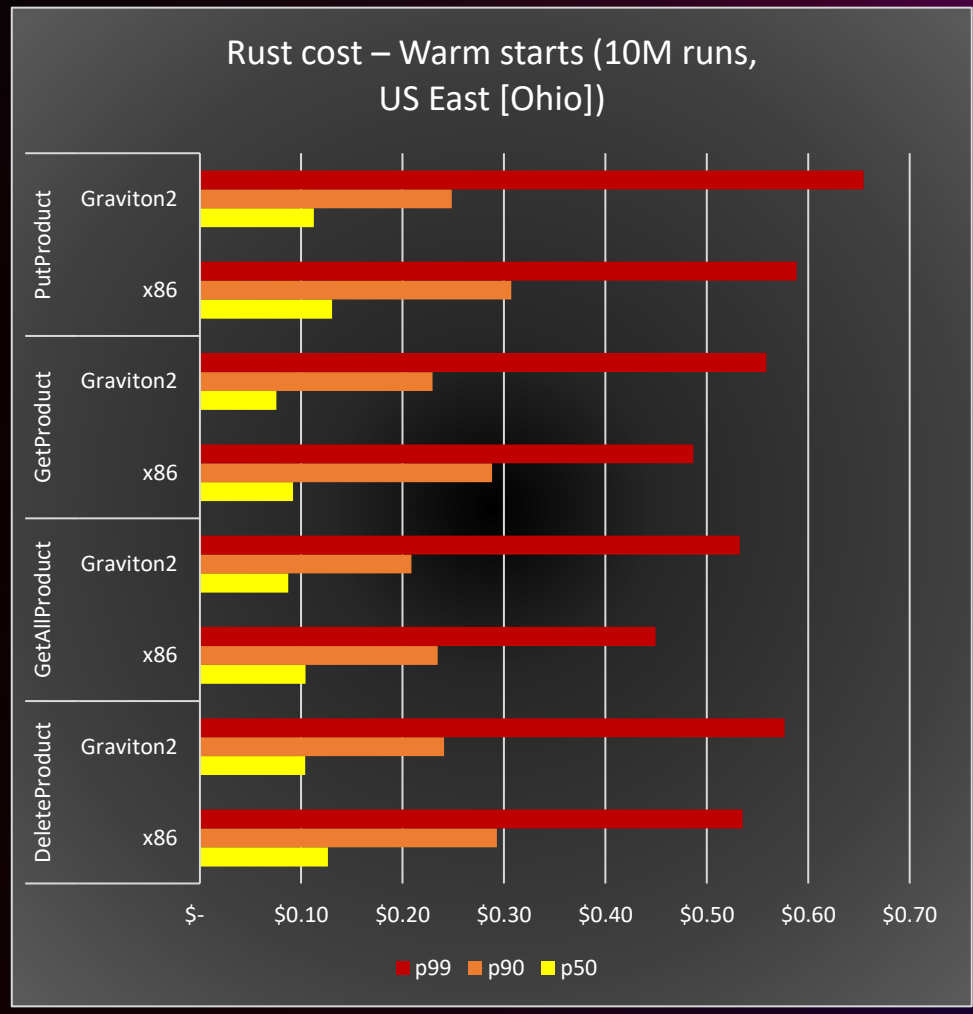



© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust and Graviton in the cloud: Cost




© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Rust and Graviton in the cloud: Summary

- Rust (today, on this application) is:
  - Faster than Java
  - More economical than Java
- Lambda/Graviton (today, on this application):
  - Has similar performance to x86
  - Is more energy efficient than x86
  - Is cheaper than x86
- Caveats
  - Your applications may vary!
  - Don’t ignore the cost of development; rewrites in particular may not net positive ROI
  - Lambda internals constantly evolve to improve performance

aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

# Thank you!


Emil Lerch

@elerch

@elerch

@emil@mstdn.io

Esteban Küber

@ekuber

@estebank




Please complete the session survey in the **mobile app**


© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.