TALK

Blazing fast serverless with Rust

Conf42: Rustlang 2022

NEOSPERIENCE

Who am I?

Luca Bianchi, PhD

Chief Technology Officer @ **Neosperience**

AWS Hero, passionate about serverless and machine learning

github.com/aletheia

https://it.linkedin.com/in/lucabianchipavia

@bianchiluca

https://speakerdeck.com/aletheia

www.ai4devs.io

NEOSPERIENCE

What is serverless?

“Serverless architecture replaces long-running virtual machines with ephemeral compute power that **comes into existence on request and disappears immediately after use**.

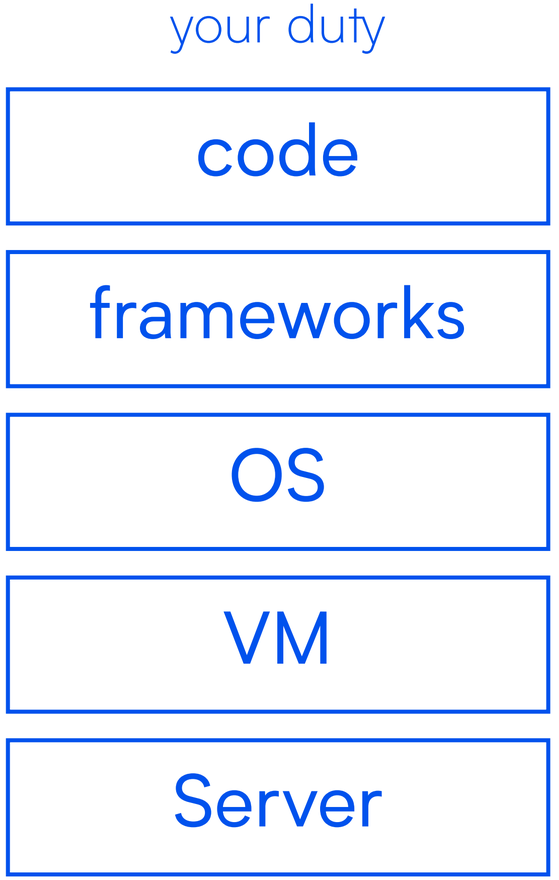
Use of this architecture can **mitigate some security concerns** such as security patching and SSH access control, and can make much more **efficient use of compute resources**. These systems cost very little to operate and can have **inbuilt scaling features**.”

— ThoughtWorks, 2016

Serverless means **no servers**

Serverless means **no servers**

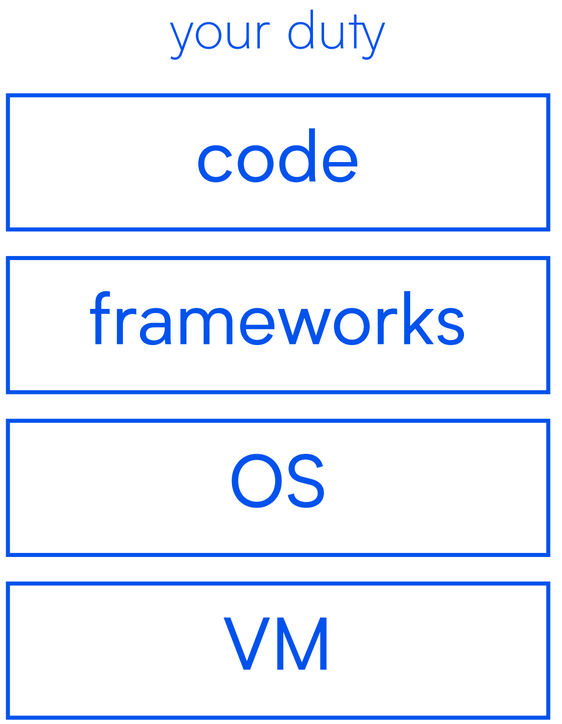
- No hardware to provision or manage
- No IT service team installing hardware
- But still it’s someone else server



Serverless means **no VMs**

Serverless means **no VMs**

- No under or over provisioning
- Never pay for idle
- No VM disaster recovery



Serverless means **no OS to patch**

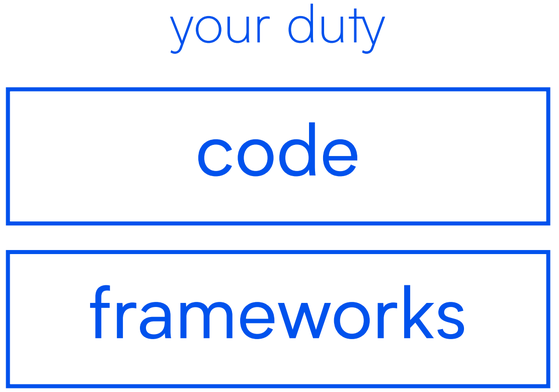
Serverless means **no OS to patch**

- OS is provisioned automatically
- Patches are installed by vendor
- Built-in best practices

Serverless means **no schedulers**

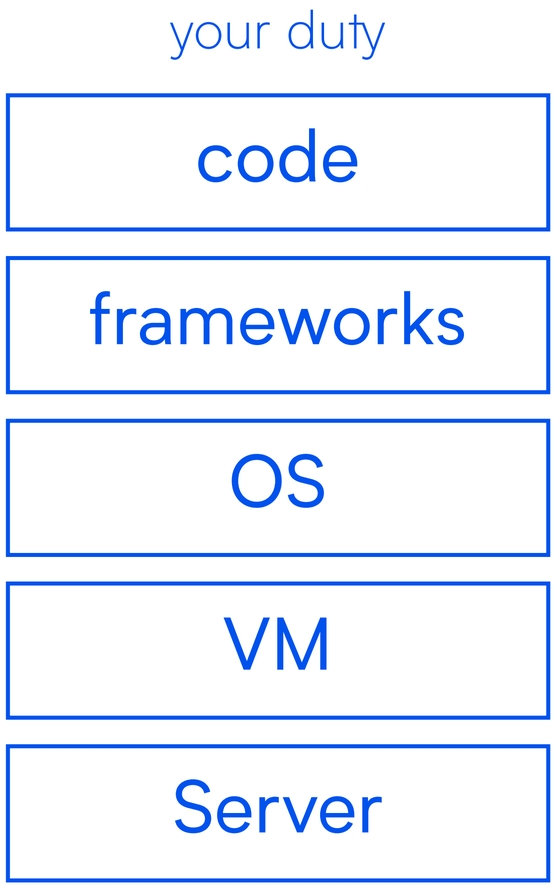
Serverless means **no schedulers**

- Code is invoked by platform
- Language support is packed within runtime
- Analytics are provided out of the box



Serverless means **Servicefull**

Patrick Debois - 2016



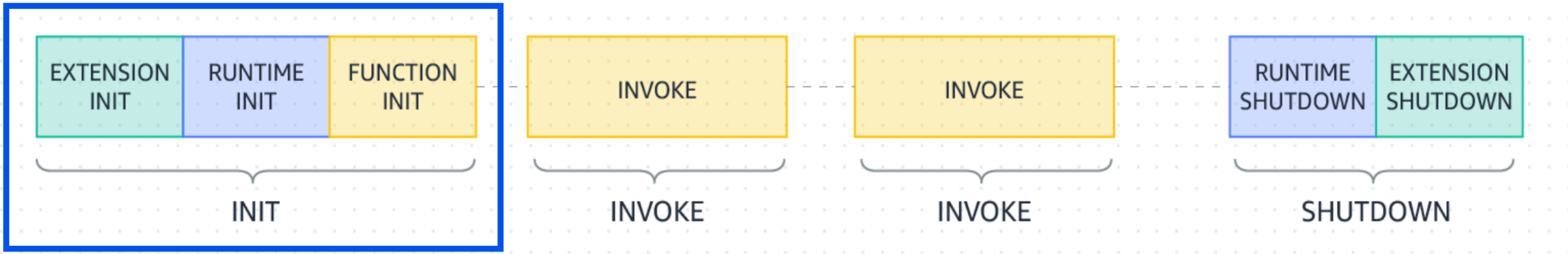
Serverless means **Servicefull**

Patrick Debois - 2016

Serverless means **Servicefull**

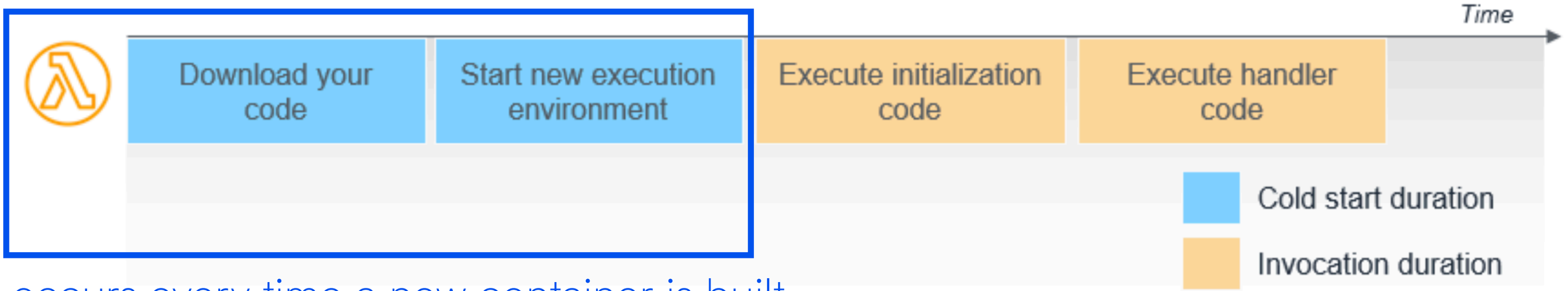
- Managed services offer storage, databases, queues, and AI through API calls
- API endpoints are secured and managed using services such as Amazon API Gateway (REST, websocket) and AWS AppSync (graphQL)
- business logic is handled through **AWS Lambda** following the Function-as-a-Service (FaaS) paradigm
  - Lambda functions are code units which happen to be packaged and deployed on the fly by AWS Lambda service upon invocation
  - AWS Lambda supports a variety of languages: Java, Node/Typescript, C# (.NET Core), PowerShell, Python, Go, and **Rust**

Lambda Lifecycle

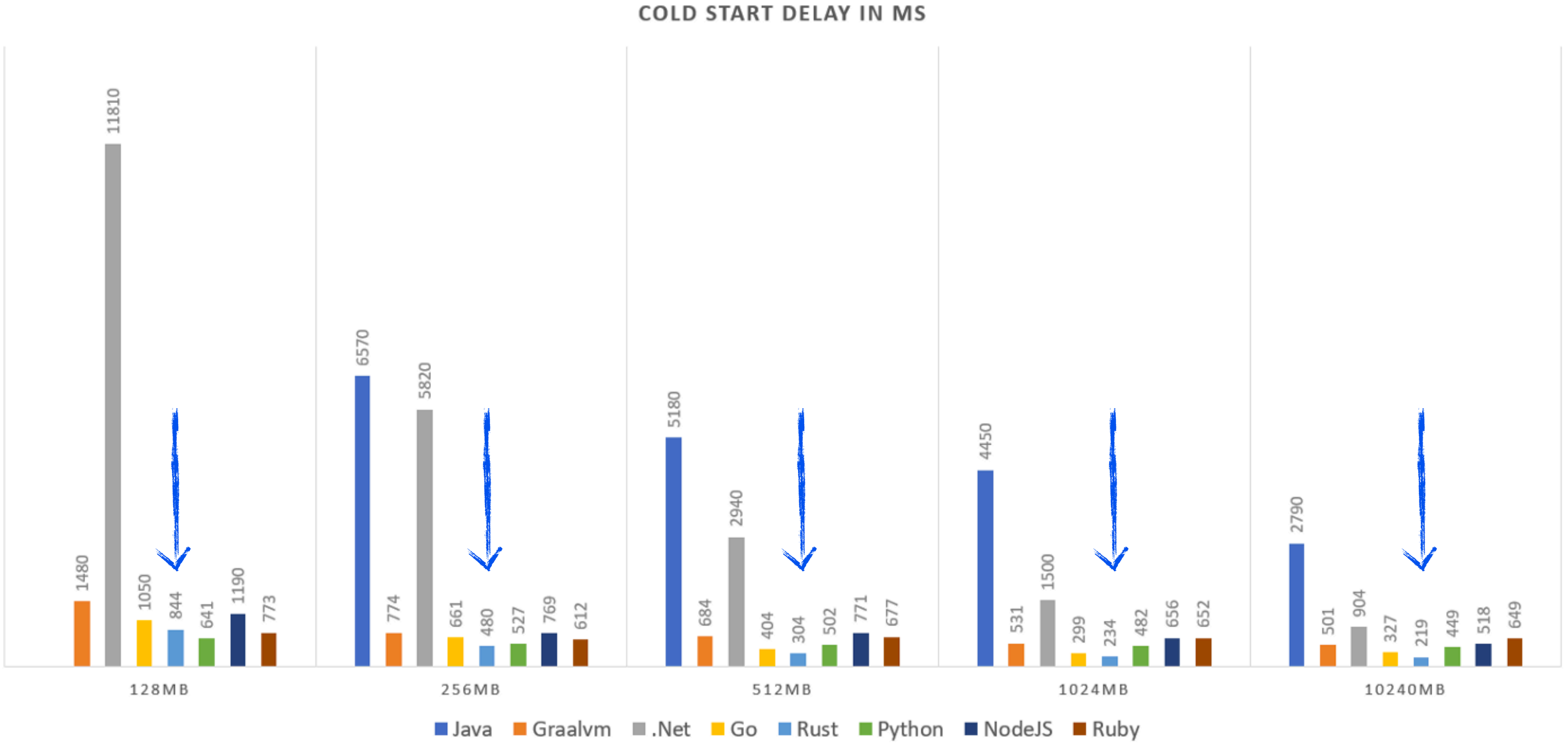


this is called “cold start”

Lambda cold start optimization



occurs every time a new container is built


COLD START DELAY IN MS
1480
11810
1050
844
641
1190
773
6570
774
5820
661
480
527
769
612
5180
684
2940
404
304
502
771
677
4450
531
1500
299
234
482
656
652
2790
501
904
327
219
449
518
649
128MB
256MB
512MB
1024MB
10240MB
Java
Graalvm
.Net
Go
Rust
Python
NodeJS
Ruby

Why Rust for Lambda?

- Rust is designed to be a safe and highly performant language.
- Rust as statically-typed language, it helps to prevent errors at compile time.
- Rust also has great tooling and a thriving community.

How Rust for Lambda?

The AWS Serverless Application Model (AWS SAM) is an open-source framework that you can use to build a serverless application.

```
> sam init

You can preselect a particular runtime or package type when using the `sam init` experience.
Call `sam init --help` to learn more.

Which template source would you like to use?
        1 - AWS Quick Start Templates
        2 - Custom Template Location
Choice: 2

Template location (git, mercurial, http(s), zip, path): gh:aws-samples/cookiecutter-aws-sam-rust

-----------------------
Generating application:
-----------------------
Location: gh:aws-samples/cookiecutter-aws-sam-rust
Output Directory: .

project_name [My Project]: myapp
project_slug [myapp]:
Select architecture:
1 - x86_64
2 - arm64
Choose from 1, 2 [1]: 2
Select template:
1 - hello-world
2 - put-dynamodb
Choose from 1, 2 [1]: 2
 [SUCCESS]: Project initialized successfully! You can now jump to myapp folder
 [INFO]: myapp/README.md contains instructions on how to proceed.
→
```

NSP

```rust
use aws_sdk_dynamodb::{model::AttributeValue, Client};
use lambda_http::{ext::RequestExt, handler, lambda_runtime::{self, Context, Error},  Body, IntoResponse,
Request, Response };
use std::env;

#[tokio::main]
async fn main() -> Result<(), Error> {
    let config = aws_config::load_from_env().await;
    let table_name = env::var("TABLE_NAME").expect("TABLE_NAME must be set");
    let dynamodb_client = Client::new(&config);

    lambda_runtime::run(handler(|request: Request, context: Context| {
        put_item(&dynamodb_client, &table_name, request, context)
    }))
    .await?;

    Ok(())
}

async fn put_item(client: &Client, table_name: &str, request: Request, _context: Context) -> Result<impl
IntoResponse, Error> {
    // Extract path parameter from request
    let path_parameters = request.path_parameters();
    let id = match path_parameters.get("id") {
        Some(id) => id,
        None => return Ok(Response::builder().status(400).body("id is required")?),
    };

    // Extract body from request
    let body = match request.body() {
        Body::Empty => "".to_string(),
        Body::Text(body) => body.clone(),
        Body::Binary(body) => String::from_utf8_lossy(body).to_string(),
    };

    // Put the item in the DynamoDB table
    let res = client
        .put_item()
        .table_name(table_name)
        .item("id", AttributeValue::S(id.to_string()))
        .item("payload", AttributeValue::S(body))
        .send()
        .await;

    // Return a response to the end-user
    match res {
        Ok(_) => Ok(Response::builder().status(200).body("item saved")?),
        Err(_) => Ok(Response::builder().status(500).body("internal error")?),
    }
```

Imports the libraries

Make the main function asynchronous

It is the idiomatic way to handle errors

Lambda runtime is initialized and context passed as parameter

Block until the results come back. The ? handles the error propagation

Tell the caller that things were successful

This is the handler

Builder pattern to construct the request

Matching pattern to handle the response

X

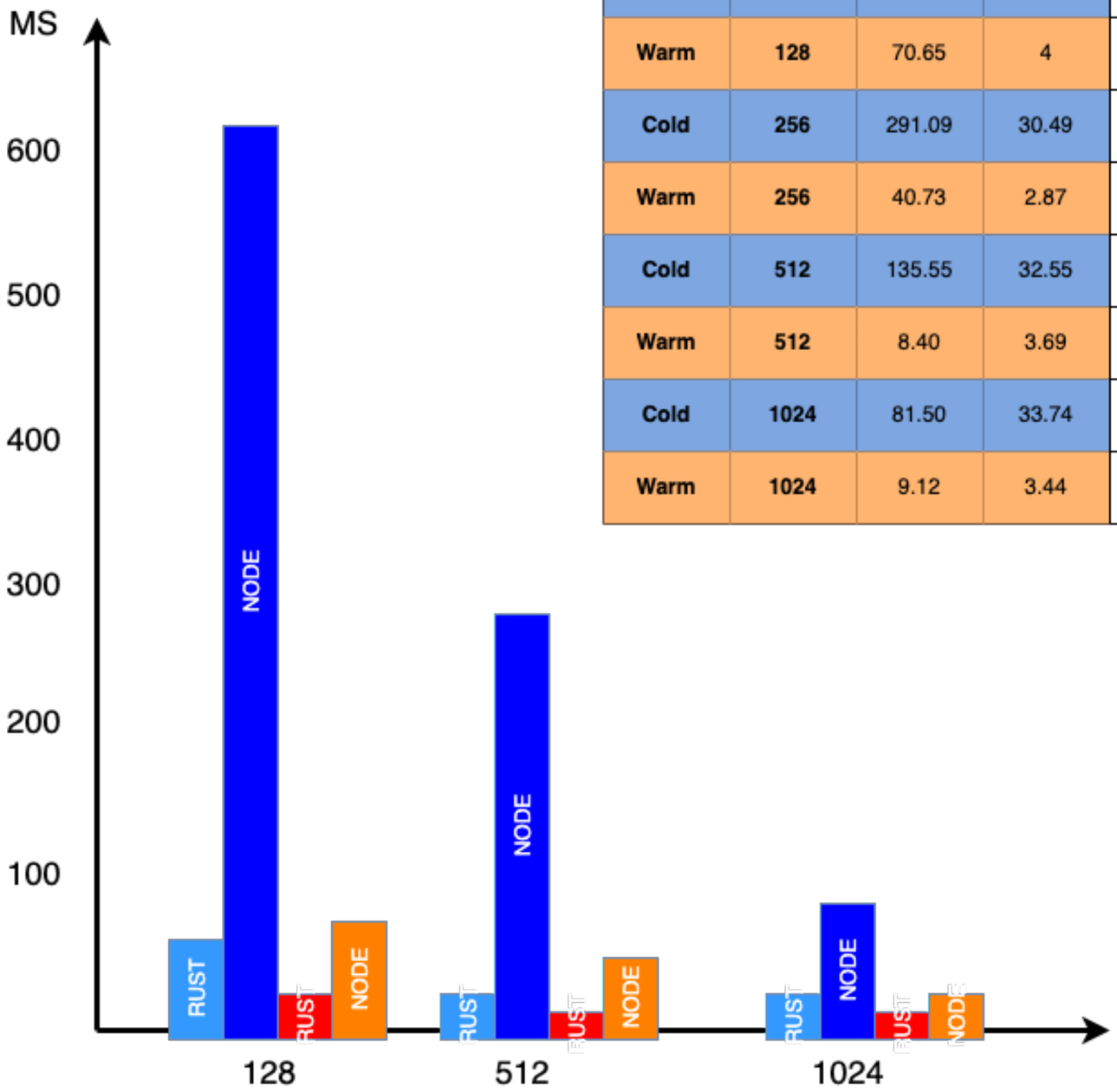
Comparison

Rust is at least 2x faster than equivalent code written in NodeJs

Lambda per millisecond billing translates this into significant saving when functions run at scale

Lambda FunctionUrl that query DynamoDB

	MB	Node	Rust	Node is slower of
Cold	128	615.18	64.55	+ 853.029%
Warm	128	70.65	4	+ 1666.25%
Cold	256	291.09	30.49	+ 854.706%
Warm	256	40.73	2.87	+ 1319.16%
Cold	512	135.55	32.55	+ 316.436%
Warm	512	8.40	3.69	+ 127.642%
Cold	1024	81.50	33.74	+ 141.553%
Warm	1024	9.12	3.44	+ 165.116%



NSP

x

Thank you!

WWW.NEOSPERIENCE.COM

INFO@NEOSPERIENCE.COM

NEOSPERIENCE

25125 BRESCIA, VIA ORZINUOVI, 20
20137 MILANO, VIA PRIVATA DECEMVIRI, 20