Why should you use Rust for Serverless workloads?

Roman Boiko

Serverless Specialist Solutions Architect
AWS

What will we cover

- AWS Lambda introduction

- Writing Lambda functions with Rust

- AWS Lambda extensions

- Writing Lambda extensions with Rust

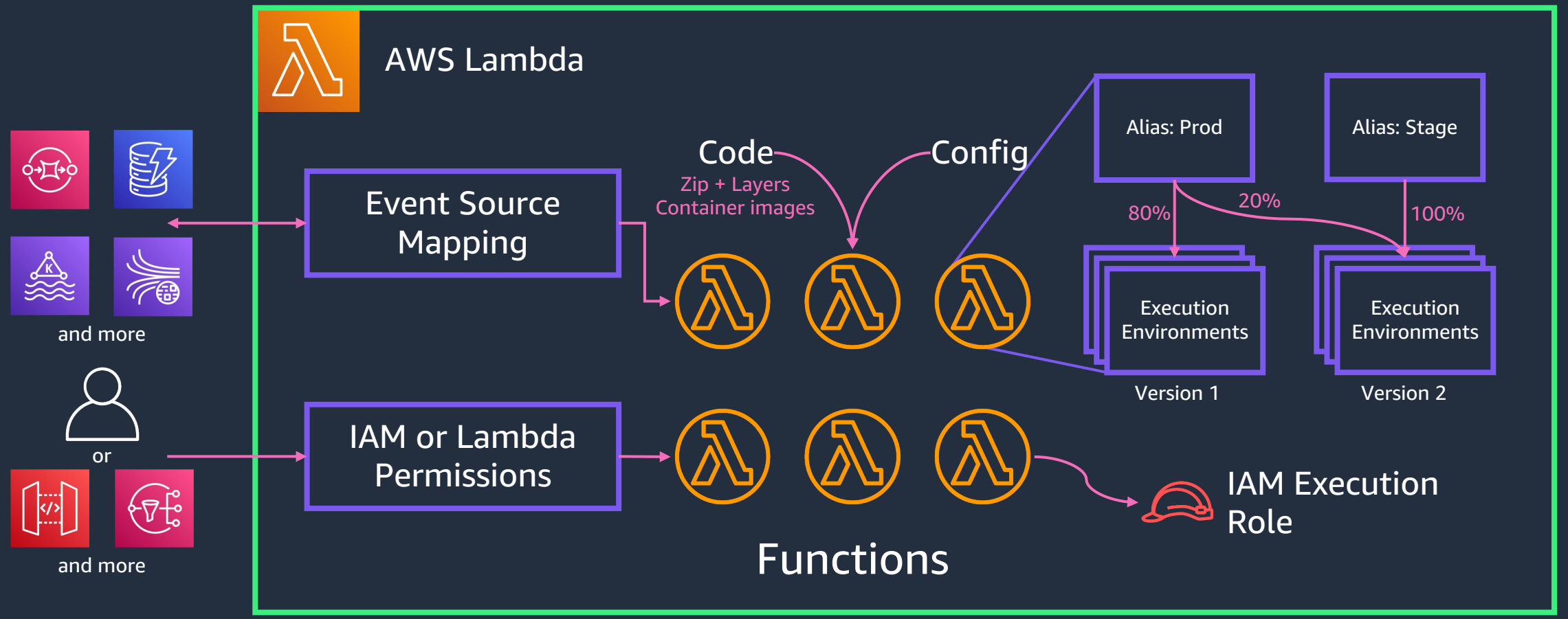
- Summary

AWS Lambda Overview

Overview of AWS Lambda

Python
Javascript
Java
C#
Golang
BYOL
Container images

Overview of AWS Lambda



Packaging code

Zip files

Function Code (/var/task)

Function Layer (/opt)

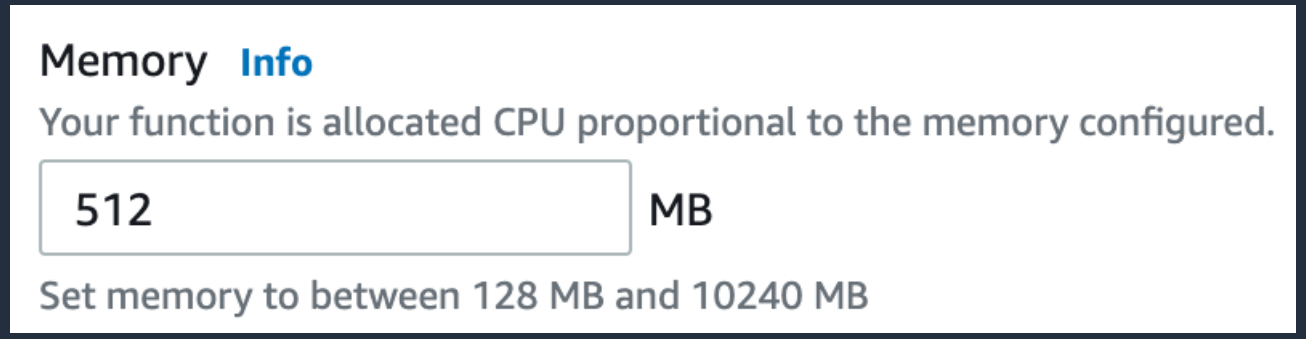
Function Layer (/opt)

Operating System (AL or AL2)

Container images

Function Container Image

Lambda Function Memory and CPU Allocation



CPU capacity is allocated proportionally to the amount of allocated memory

At ~1800MB of allocated memory your function will have the equivalent of one full vCPU

The pricing view

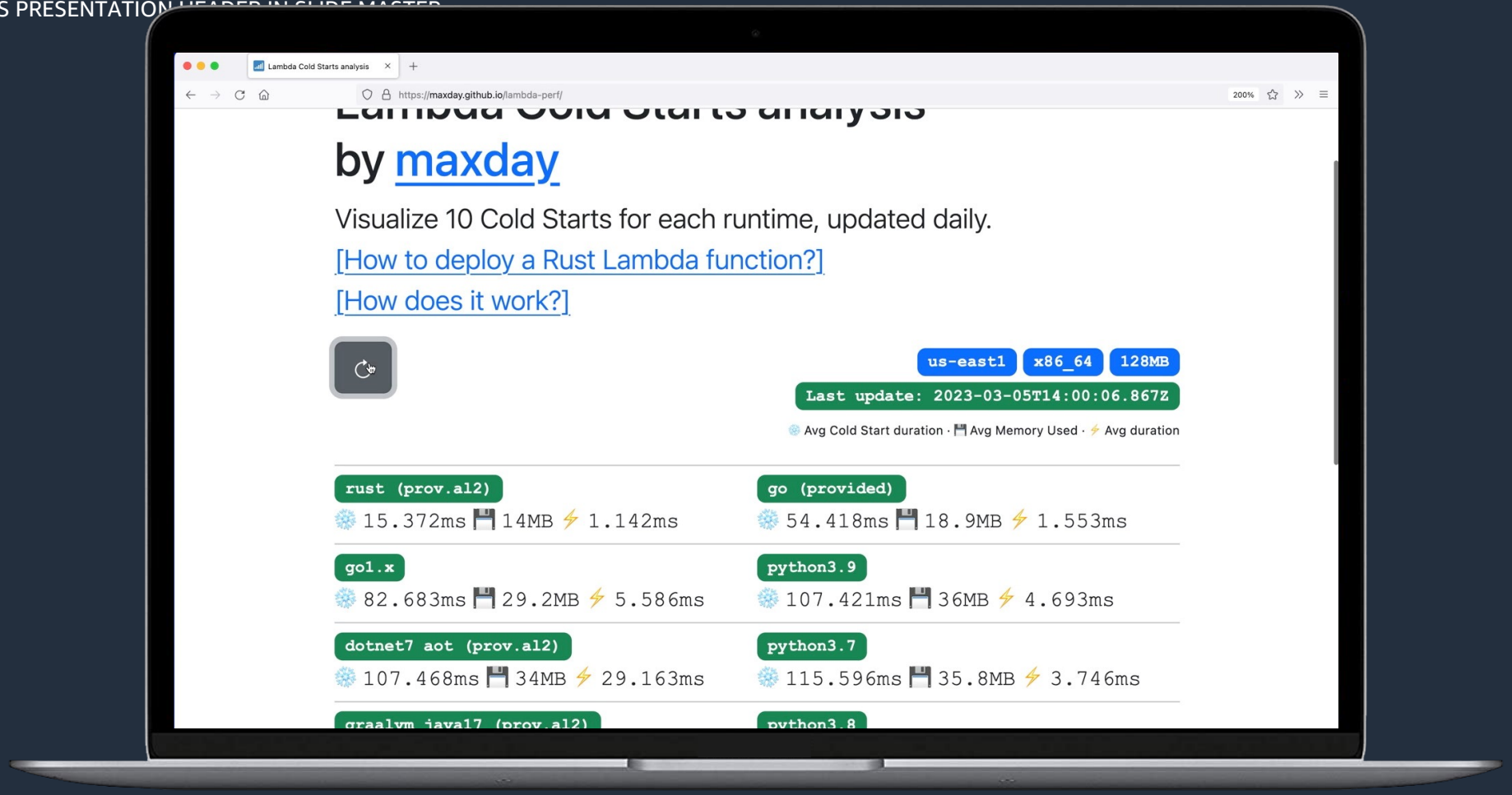
Two pricing components:

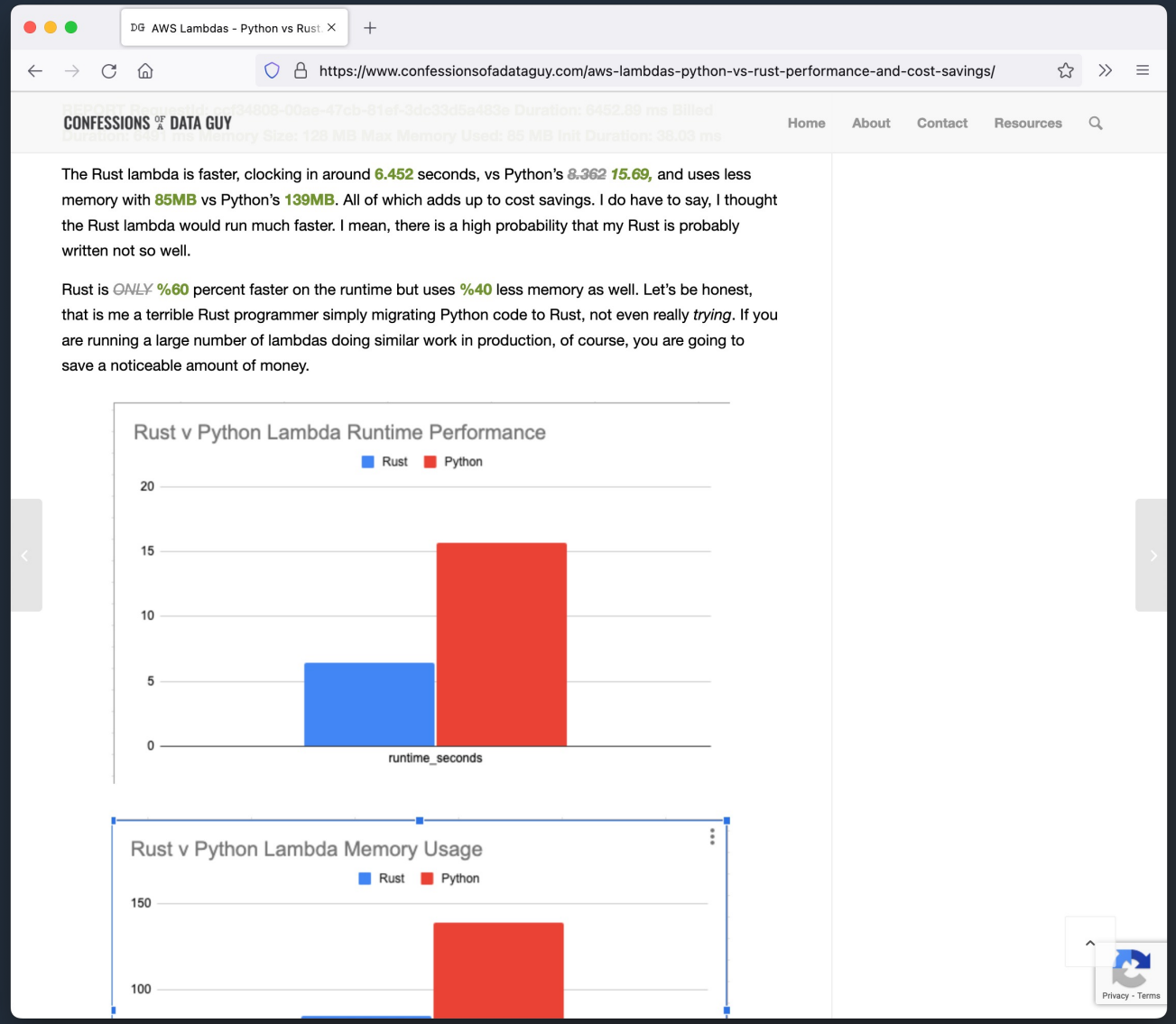
- Number of requests
- Duration (in GB-s)

Duration is metered by 1ms increment, based on the function memory setting.

100ms with 2GB of RAM
costs the same as
200ms with 1GB of RAM

Motivating Rust – performance



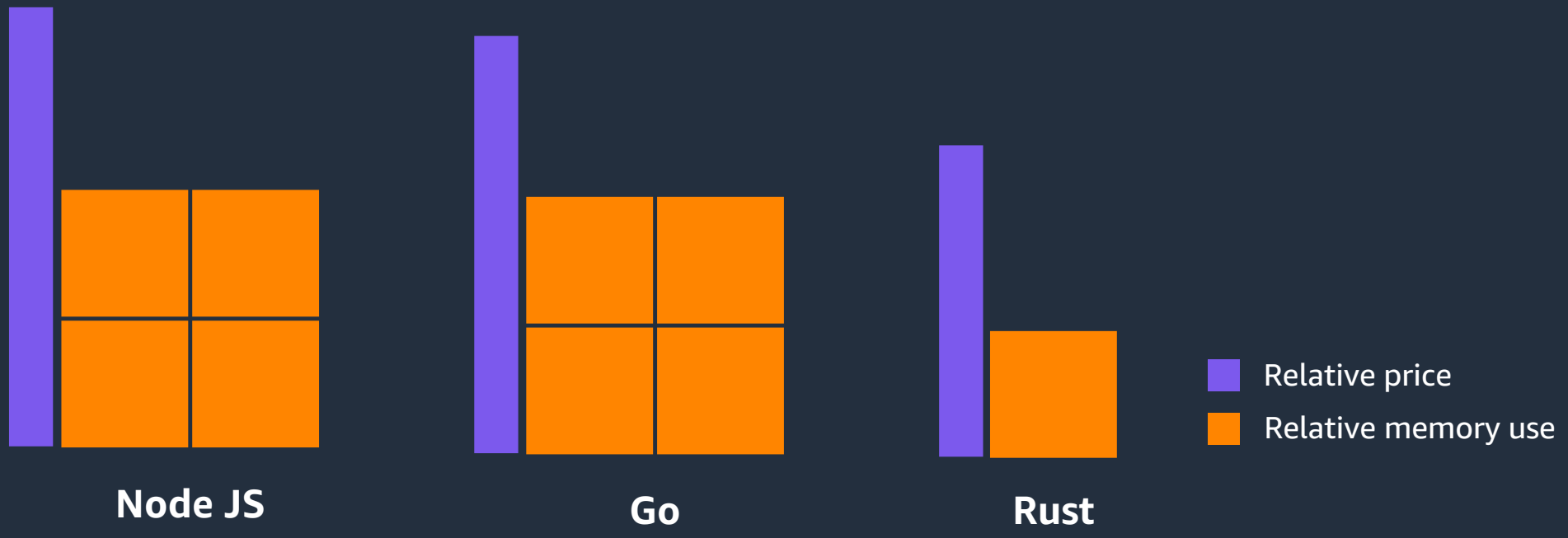


Motivating Rust – cost efficiency

“[With Rust], we saw a 75% reduction in CPU usage and a 95% reduction in memory usage in production. ”

Alan Ning

Tenable.io



Price of running equivalent workloads running 2.5B times using different AWS Lambda runtimes

Motivating Rust - sustainability

“Broad adoption of C and Rust could reduce energy consumption of compute by 50% – even with a conservative estimate.”

Shane Miller & Carl Lerche
AWS

Writing Lambda functions with Rust

How to bootstrap a new project – cargo lambda

```
brew tap cargo-lambda/cargo-lambda
brew install cargo-lambda
```

```
cargo lambda new new-lambda-project \
    && cd new-lambda-project
```

Cargo Lambda provides tools and workflows to help you get started building Rust functions for AWS Lambda from scratch.

When you're ready to put your work in production, Cargo Lambda helps you build and deploy your functions on AWS Lambda in an efficient manner.

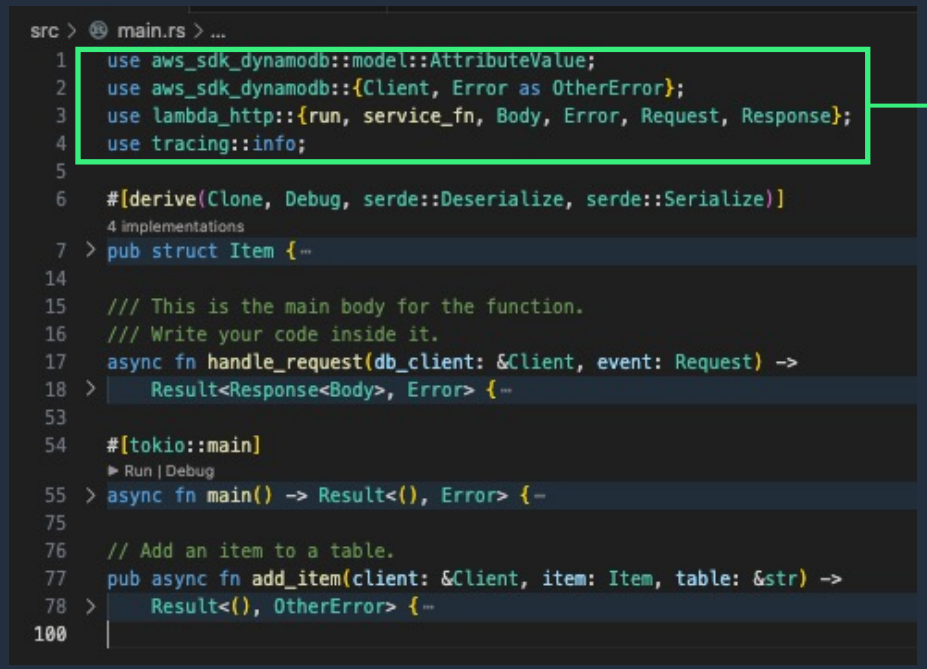
Sample Lambda function

```
src > main.rs > ...
1   use aws_sdk_dynamodb::model::AttributeValue;
2   use aws_sdk_dynamodb::{Client, Error as OtherError};
3   use lambda_http::{run, service_fn, Body, Error, Request, Response};
4   use tracing::info;
5
6   #[derive(Clone, Debug, serde::Deserialize, serde::Serialize)]
    4 implementations
7   pub struct Item {…
14
15  /// This is the main body for the function.
16  /// Write your code inside it.
17  async fn handle_request(db_client: &Client, event: Request) ->
18      Result<Response<Body>, Error> {…
53
54  #[tokio::main]
    ▶ Run | Debug
55  async fn main() -> Result<(), Error> {…
75
76  // Add an item to a table.
77  pub async fn add_item(client: &Client, item: Item, table: &str) ->
78      Result<(), OtherError> {…
100
```

A sample Lambda function

- Lambda acts as a backend for HTTP requests
- Amazon API Gateway forwards requests to Lambda function
- Event object contains data from HTTP request
- Lambda function processes the data and uses the AWS SDK for Rust to save data into DynamoDB
- If the put operation is successful Lambda function returns success
- API Gateway converts the return object into a response to the API call

Sample Lambda function



- We use Rust runtime for AWS Lambda which provides a Lambda runtime for applications written in Rust
- As our function gets events from Amazon API Gateway we use lambda-http a library that makes it easy to write API Gateway proxy event focused Lambda functions in Rust. It automatically adds Rust runtime.
- We also use AWS SDK for Rust to make requests to DynamoDB

Handler method

src > main.rs > handle_request

```rust
/// This is the main body for the function.
/// Write your code inside it.
async fn handle_request(db_client: &Client, event: Request) ->
    Result<Response<Body>, Error> {
    // Extract some useful information from the request
    let body: &Body = event.body();
    let s: &str = std::str::from_utf8(body).expect(msg: "invalid utf-8 sequence");
    //Log into Cloudwatch
    info!(payload = %s, "JSON Payload received");

    //Serialze JSON into struct.
    //If JSON is incorrect, send back 400 with error.
    let item: Item = match serde_json::from_str::<Item>(s) {
        Ok(item: Item) => item,
        Err(err: Error) => {
            let resp: Response<Body> = Response::builder() Builder
                .status(400) Builder
                .header(key: "content-type", value: "text/html") Builder
                .body(err.to_string().into()) Result<Response<Body>, Error>
                .map_err(op: Box::new)?;
            return Ok(resp);
        }
    };

    //Insert into the table.
    add_item(db_client, item.clone(), table: "lambda_dyno_example").await?;

    //Deserialize into json to return in the Response
    let j: String = serde_json::to_string(&item)?;

    //Send back a 200 - success
    let resp: Response<Body> = Response::builder() Builder
        .status(200) Builder
        .header(key: "content-type", value: "text/html") Builder
        .body(j.into()) Result<Response<Body>, Error>
        .map_err(op: Box::new)?;
    Ok(resp)
} fn handle_request
```

The Lambda function *handler* is the method in your function code that processes events.

When your function is invoked, Lambda runs the handler method.

Method gets an event which abstracts data from API Gateway or other HTTP sources. Event also have context object which includes data about Lambda runtime.

The function returns a Result type. If the function is successful, the result is a JSON value. If the function is not successful, the result is an error.

Your function runs until the handler returns a response, exits, or times out.

Handler method

src > main.rs > handle_request

```rust
/// This is the main body for the function.
/// Write your code inside it.
async fn handle_request(db_client: &Client, event: Request) ->
    Result<Response<Body>, Error> {
    // Extract some useful information from the request
    let body: &Body = event.body();
    let s: &str = std::str::from_utf8(body).expect(msg: "invalid utf-8 sequence");
    //Log into Cloudwatch
    info!(payload = %s, "JSON Payload received");

    //Serialze JSON into struct.
    //If JSON is incorrect, send back 400 with error.
    let item: Item = match serde_json::from_str::<Item>(s) {
        Ok(item: Item) => item,
        Err(err: Error) => {
            let resp: Response<Body> = Response::builder() Builder
                .status(400) Builder
                .header(key: "content-type", value: "text/html") Builder
                .body(err.to_string().into()) Result<Response<Body>, Error>
                .map_err(op: Box::new)?;
            return Ok(resp);
        }
    };

    //Insert into the table.
    add_item(db_client, item.clone(), table: "lambda_dyno_example").await?;

    //Deserialize into json to return in the Response
    let j: String = serde_json::to_string(&item)?;

    //Send back a 200 - success
    let resp: Response<Body> = Response::builder() Builder
        .status(200) Builder
        .header(key: "content-type", value: "text/html") Builder
        .body(j.into()) Result<Response<Body>, Error>
        .map_err(op: Box::new)?;
    Ok(resp)
} fn handle_request
```

- Within the handler, you may run an arbitrary Rust
- You can define other Rust methods outside of the handler scope and use them in your handler method
- You can programmatically interact with the Lambda environment, e.g.
  - println!(..) to log into Amazon CloudWatch or use Tracing crate for advanced logging
  - Access the filesystem at /tmp
  - Make network requests to external services

Main method

src > main.rs > ...

```
53
54  #[tokio::main]
    ▶ Run | Debug
55  async fn main() -> Result<(), Error> {
56      // required to enable CloudWatch error logging by the runtime
57      tracing_subscriber::fmt()
58          .with_max_level(tracing::Level::INFO)
59          // disable printing the name of the module in every log line.
60          .with_target(false)
61          // disabling time is handy because CloudWatch will add the ingestion time.
62          .without_time()
63          .init();
64
65      //Get config from environment.
66      let config: SdkConfig = aws_config::load_from_env().await;
67      //Create the DynamoDB client.
68      let client: Client = Client::new(&config);
69
70      run(service_fn(|event: Request| async {
71          handle_request(db_client: &client, event).await
72      }))
73      .await
74  }
```

main function is the entry point that runs the Lambda function code. The Rust runtime client uses Tokio as an async runtime, so you must annotate the main function with #[tokio::main].

Main method

src > main.rs > ...

```rust
#[tokio::main]
▶ Run | Debug
async fn main() -> Result<(), Error> {
    // required to enable CloudWatch error logging by the runtime
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        .init();

    //Get config from environment.
    let config: SdkConfig = aws_config::load_from_env().await;
    //Create the DynamoDB client.
    let client: Client = Client::new(&config);

    run(service_fn(|event: Request| async {
        handle_request(db_client: &client, event).await
    }))
    .await
}
```

You can declare shared variables that are independent of your Lambda function's handler code. These variables can help you load state information during the Init phase, before your function receives any events.

Handler method

src > main.rs > handle_request

```rust
/// This is the main body for the function.
/// Write your code inside it.
async fn handle_request(db_client: &Client, event: Request) ->
    Result<Response<Body>, Error> {
    // Extract some useful information from the request
    let body: &Body = event.body();
    let s: &str = std::str::from_utf8(body).expect(msg: "invalid utf-8 sequence");
    //Log into Cloudwatch
    info!(payload = %s, "JSON Payload received");

    //Serialze JSON into struct.
    //If JSON is incorrect, send back 400 with error.
    let item: Item = match serde_json::from_str::<Item>(s) {
        Ok(item: Item) => item,
        Err(err: Error) => {
            let resp: Response<Body> = Response::builder() Builder
                .status(400) Builder
                .header(key: "content-type", value: "text/html") Builder
                .body(err.to_string().into()) Result<Response<Body>, Error>
                .map_err(op: Box::new)?;
            return Ok(resp);
        }
    };

    //Insert into the table.
    add_item(db_client, item.clone(), table: "lambda_dyno_example").await?;

    //Deserialize into json to return in the Response
    let j: String = serde_json::to_string(&item)?;

    //Send back a 200 - success
    let resp: Response<Body> = Response::builder() Builder
        .status(200) Builder
        .header(key: "content-type", value: "text/html") Builder
        .body(j.into()) Result<Response<Body>, Error>
        .map_err(op: Box::new)?;
    Ok(resp)
} fn handle_request
```

- An Ok statement ends the function call with the provided object, otherwise null will be returned
- For synchronous Lambda invocations, the result is returned to the caller
- For asynchronous Lambda invocations, the result is discarded
- The example returns an object as expected by the API Gateway proxy integration. This will result in an HTTP200 response with the given headers and body

Lambda Extensions

Extensions Use Cases

A NEW WAY TO INTEGRATE WITH THE LAMBDA EXECUTION ENVIRONMENT

Capture diagnostic information before, during, and after function invocation

Automatically instrument your code without needing code changes

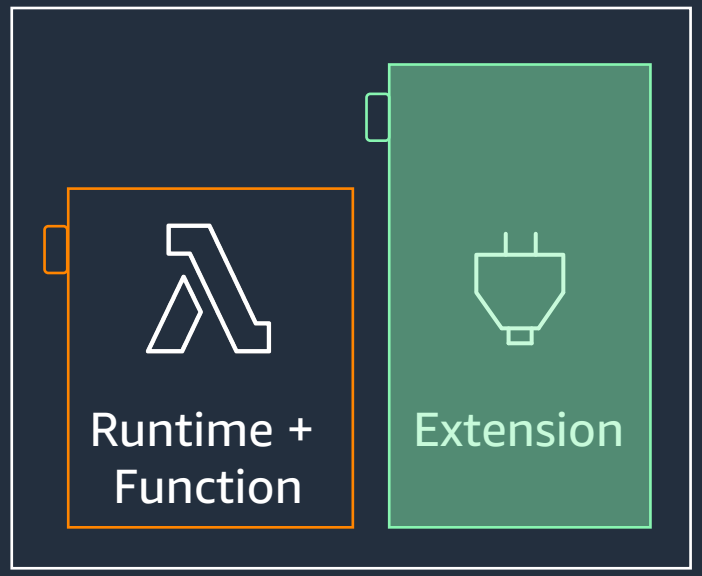
Fetch configuration settings or secrets before the function invocation

Detect and alert on function activity through hardened security agents

External Extensions

External Lambda extensions are companion processes that run within Lambda's execution environment that easily integrate Lambda with your favorite tools for monitoring, observability, security, and governance.

External Extensions



External Extensions that run within the sandbox, but outside the runtime as a separate process.

- Lambda function shares resources such as memory, CPU, and disk I/O, with the extension
- Extension has access to environment variables (some exceptions apply) and uses the same IAM Role as the Lambda function
- Extensions start before the runtime process and finishes after the runtime shuts down
- Extensions can be written in different language to the function (need compatible environment or can be delivered as an executable)

Extensions can impact the performance of your function because they share function resources such as CPU, memory, and storage.

Performance Considerations

Adding more memory to the Lambda function can increase extension performance

Deliver extensions as executables (minimize the runtime overhead)

Use non GC languages such as Rust

Lambda Extensions Example

Sample Lambda Extensions code in Rust

src > main.rs > main

```rust
use lambda_extension::*;
use tracing::info;

async fn events_extension(event: LambdaEvent) -> Result<(), Error> {
    match event.next {
        NextEvent::Shutdown(e: ShutdownEvent) => {
            info!(event_type = "shutdown", event = ?e, "shutting down");
        }
        NextEvent::Invoke(e: InvokeEvent) => {
            info!(event_type = "invoke", event = ?e, "invoking function");
        }
    }
    Ok(())
}

#[tokio::main]
▶ Run | Debug
async fn main() -> Result<(), Error> {
    // The runtime logging can be enabled here by initializing `tracing` with `tracing-subscriber`
    // While `tracing` is used internally, `log` can be used as well if preferred.
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        .init();


    Extension::new()
        .with_events_processor(service_fn(events_extension))
        .run()
        .await
}
```

- Lamba Extensions Invoke method is triggered by Lambda service during invocation of main Lambda function
- Lambda Extensions Shutdown method is triggered by Lambda service

Summary

- Using Rust for AWS Lambda functions allows you to get the best cost/performance characteristics
- You can use Rust both for writing business logic and Lambda extensions.
- AWS provides a lot of tooling to enable local Lambda function development, testing, and debugging for Rust.
- There’s a variety of packaging and deployment options that are available and they can be integrated with your existing CI/CD pipeline to automate the end to end process.

Thank you!

Roman Boiko

@romannboiko