AWS re:Invent

NOV. 29 – DEC. 3, 2021 | LAS VEGAS, NV

SVS401-R1

AWS Lambda function performance tuning

Chris Munns
Tech Lead/Advisor – Startup Solution Architects
Amazon Web Services

Agenda

- Measurement of AWS Lambda function performance
- Comparing with AWS Lambda Power Tuning
- Function configuration: CPU and memory
- Cold starts, warm starts, no starts?
- Application design patterns

munns@amazon.com
@chrismunns

Not in the agenda

- Comparisons of runtimes
- Tricks compiling/building runtimes for slimmed down performance
- Databases/downstream services

munns@amazon.com
@chrismunns

Three areas of performance

Latency

Throughput

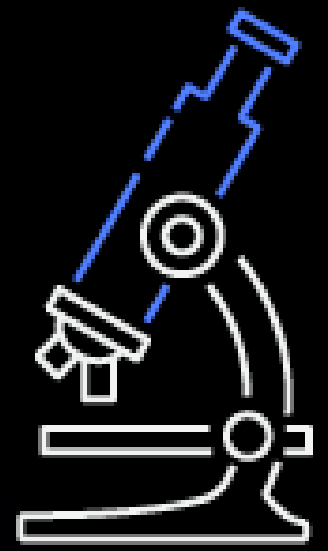
Cost

Measurement of AWS Lambda function performance

Measurement of AWS Lambda function performance

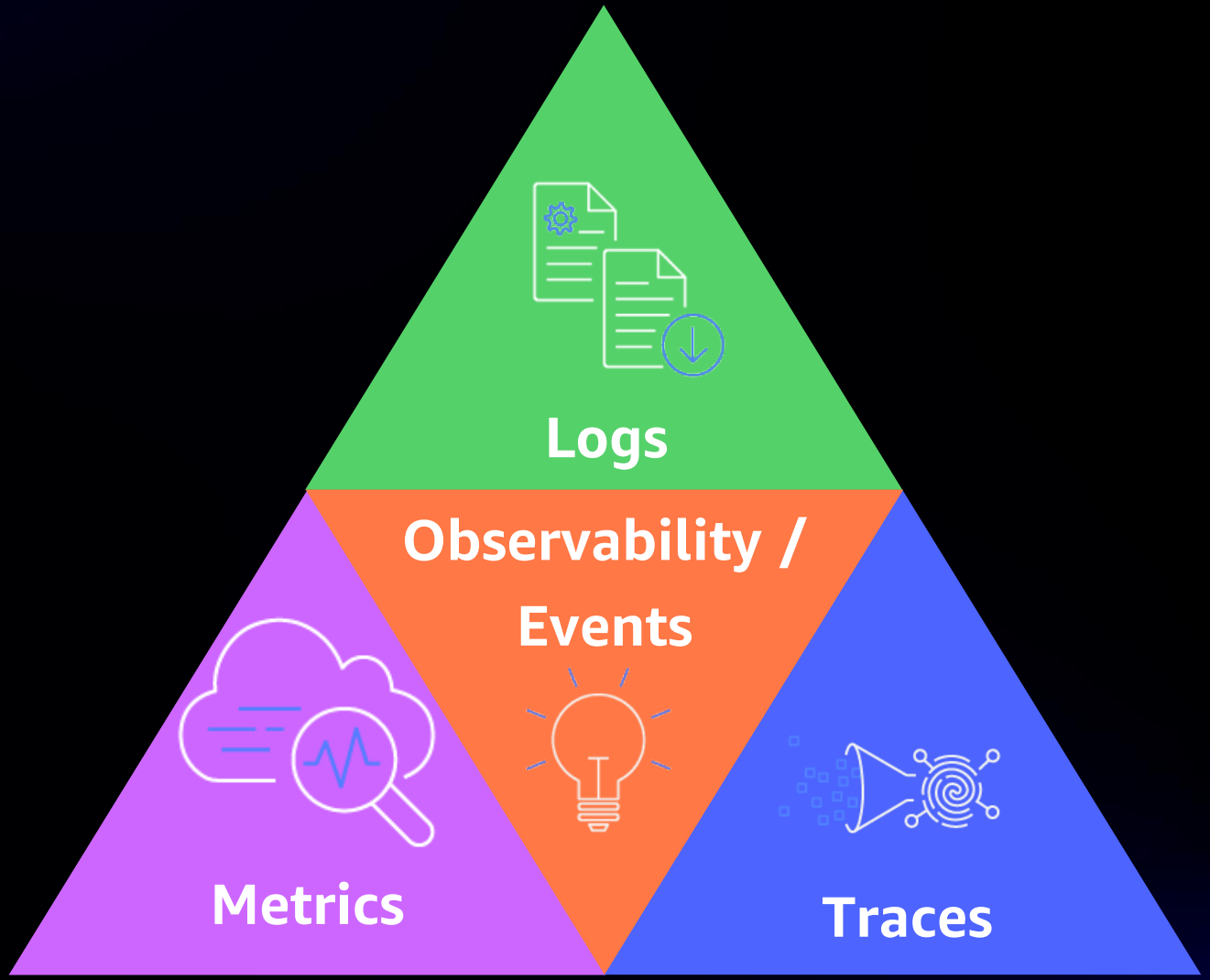
Without data, you can’t optimize your function performance:

- Data from inside the function
- Data from the invoke source
- Data to explain downstream service or dependency performance



You should have data match production workloads

Observability



Observability inputs

Logs	Metrics	Traces
Timestamped records of discrete events that happened within an application or system, such as a failure, an error, or a state transformation	Numeric data measured at various time intervals (time series data); SLIs (request rate, error rate, duration, CPU%, etc.)	A trace represents a single user's journey across multiple applications and systems (usually microservices)

Source: "Distributed Systems Observability" by Cindy Sridharan, available at https://s12d.com/dist-systems-obs

AWS services for observability

Amazon CloudWatch

AWS X-Ray

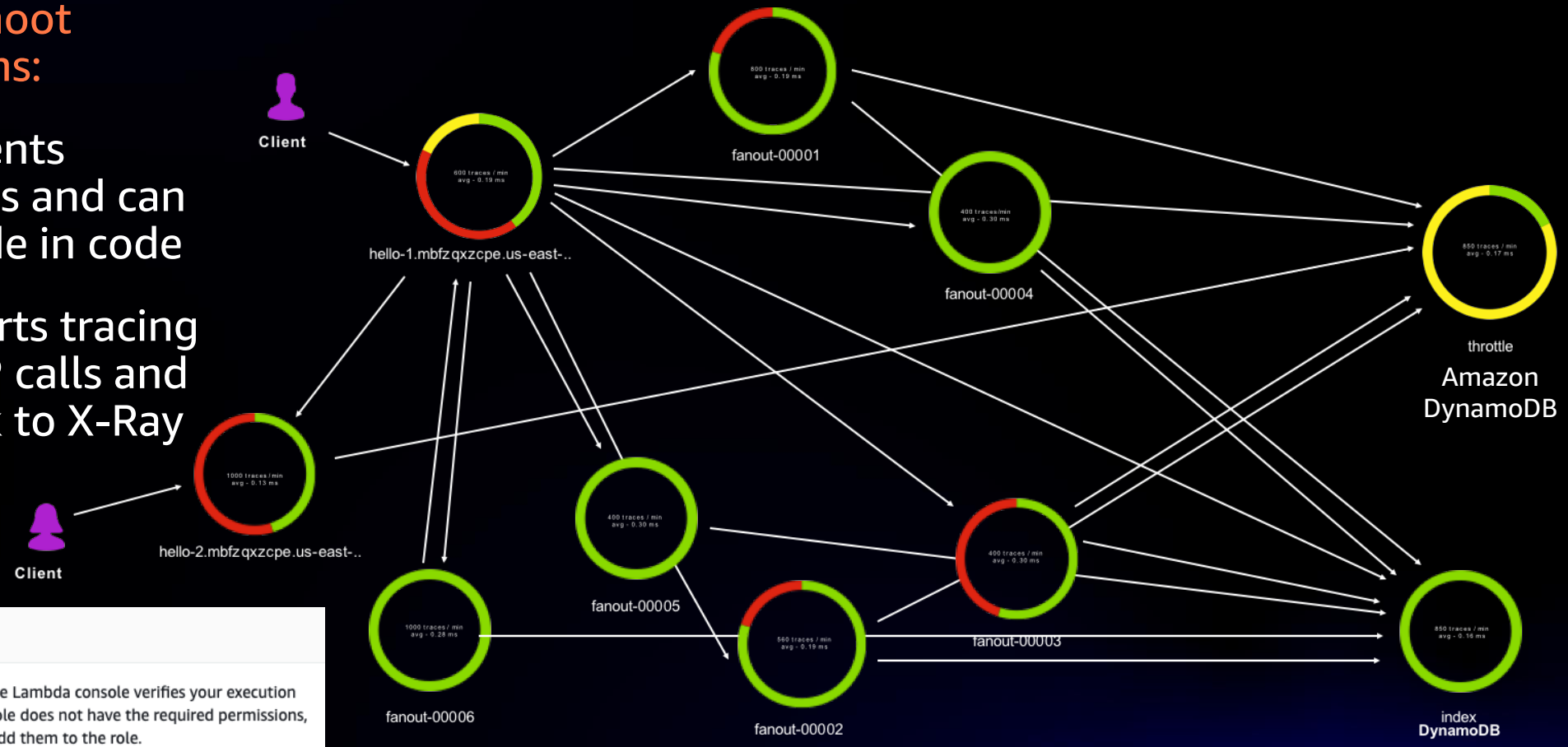
Amazon CloudWatch	AWS X-Ray
Dashboards	Traces
Logs	Analytics
Metrics	Service map
Alarms	
Events	

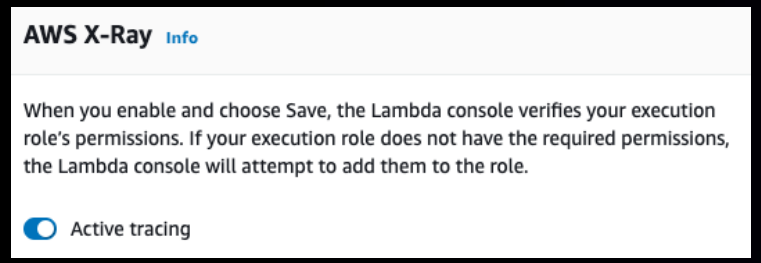
Measuring with AWS X-Ray

```
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))
AWSXRay.captureFunc('annotations', subsegment => {
    subsegment.addAnnotation('Name', name)
    subsegment.addAnnotation('UserID', event.userid)
})
```

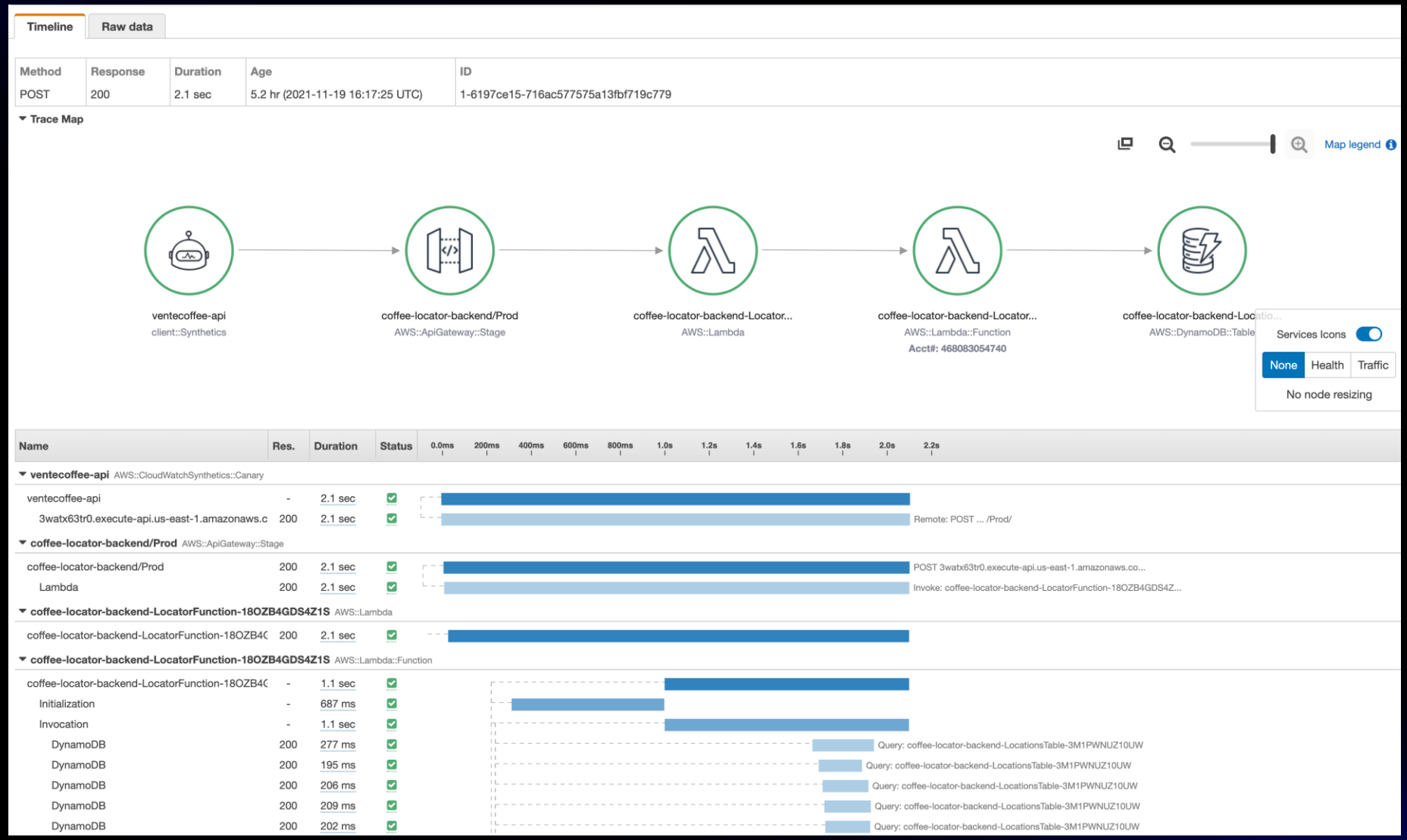
Profile and troubleshoot serverless applications:

- Lambda instruments incoming requests and can capture calls made in code
- API Gateway inserts tracing header into HTTP calls and reports data back to X-Ray

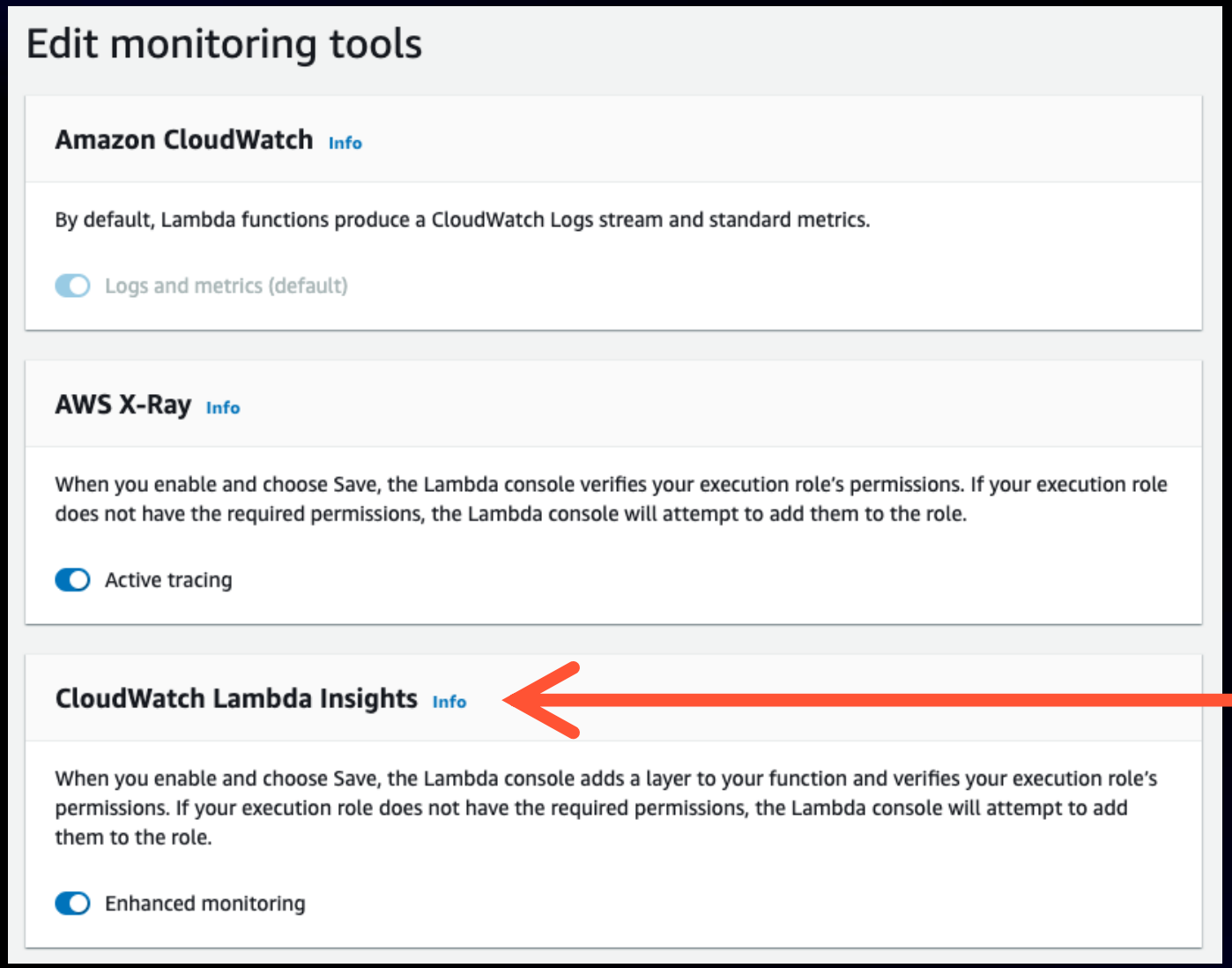




AWS X-Ray trace example



Enabling CloudWatch Lambda Insights



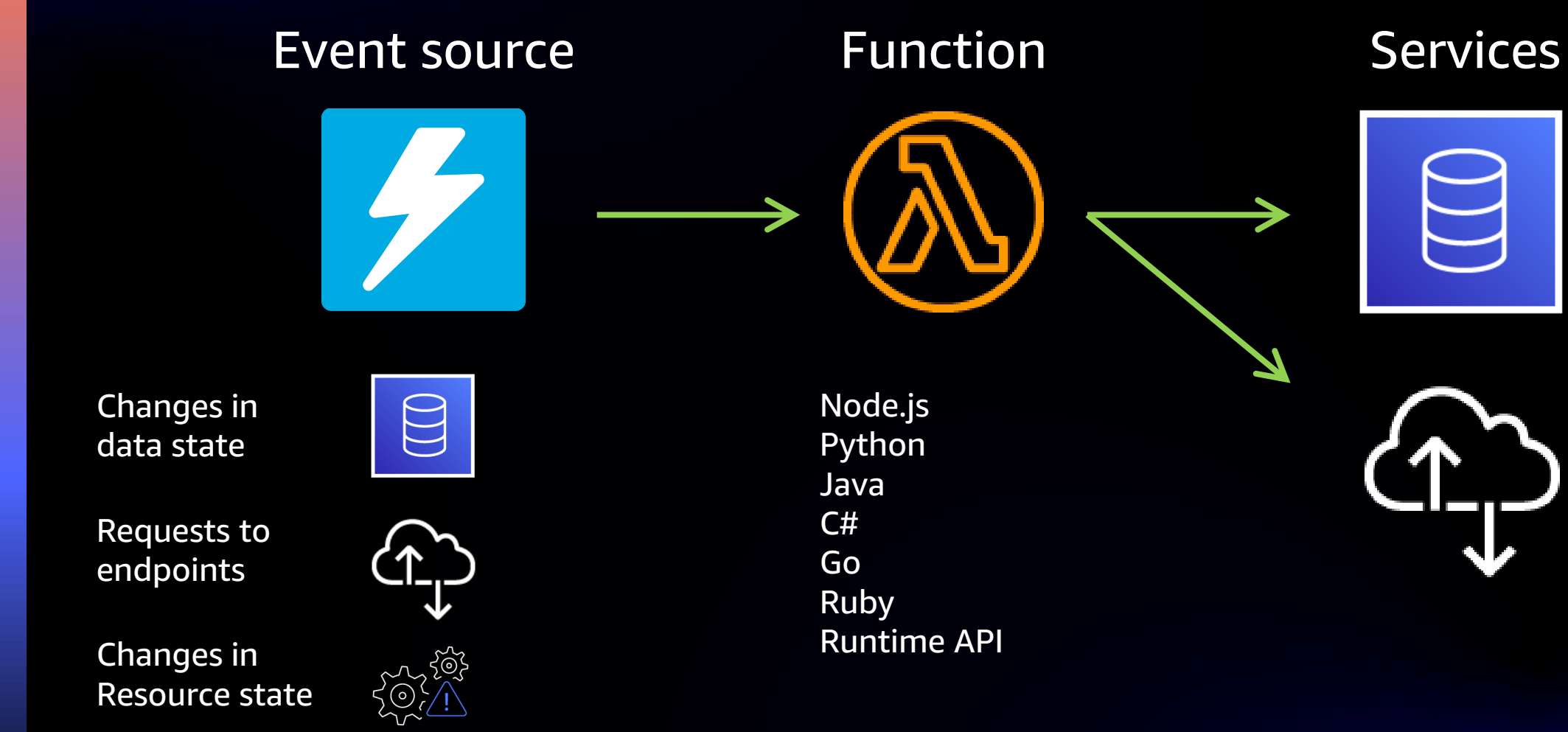
"Collects, aggregates, and summarizes system-level metrics, including CPU time, memory, disk, and network.

It also collects, aggregates, and summarizes diagnostic information, such as cold starts and Lambda worker shutdowns."

– Lambda Docs

But what are we measuring?

AWS Lambda-based applications



Anatomy of an AWS Lambda function

Handler() function

Function to be executed upon invocation

Event object

Data sent during Lambda function invocation

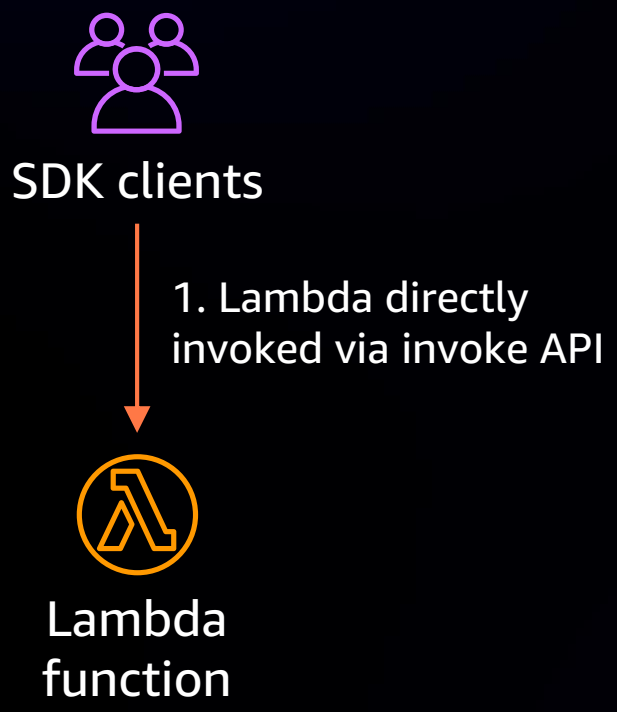
Context object

Methods available to interact with runtime information (request ID, log group, etc.)

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello World!')
    }
```

aws

AWS Lambda API



API provided by the AWS Lambda service

Used by all other services that invoke Lambda across all models

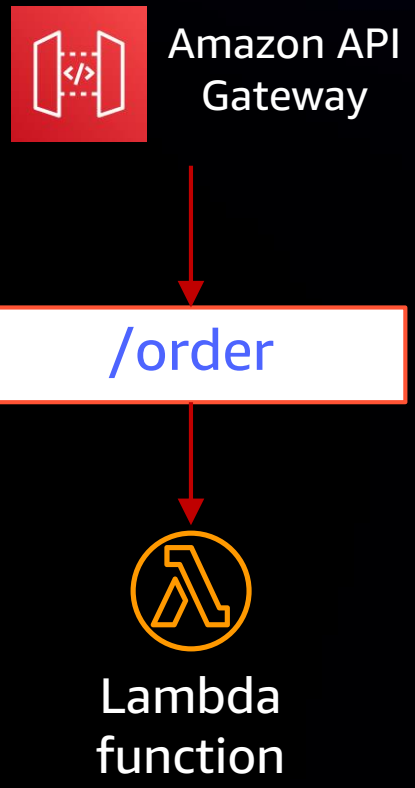
Supports sync and async

Can pass any event payload structure you want

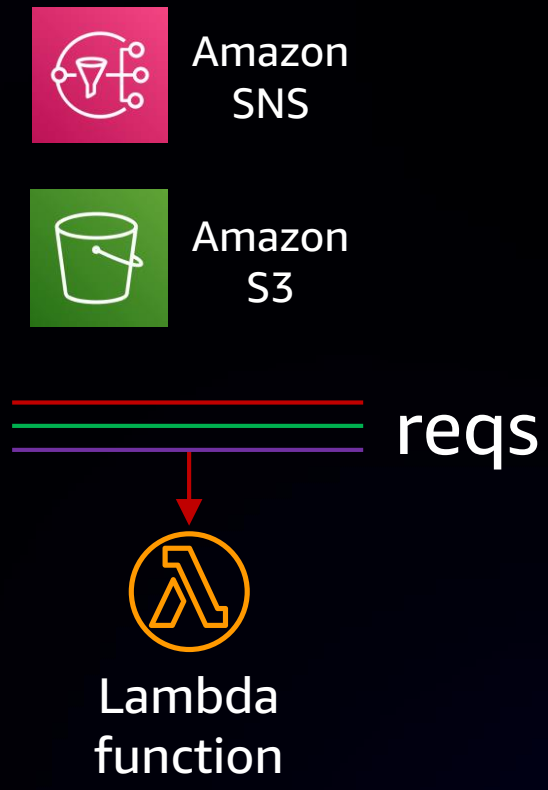
Client included in every SDK

AWS Lambda execution model

Synchronous (push)



Asynchronous (event)

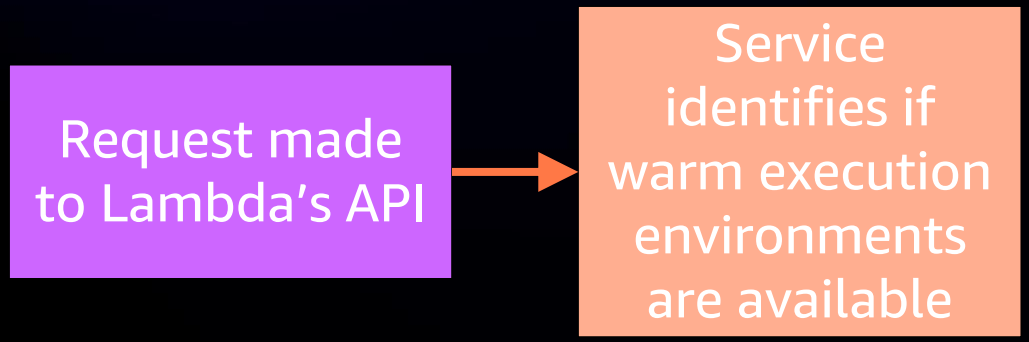


Stream (poll-based)

Function lifecycle

Request made to Lambda's API

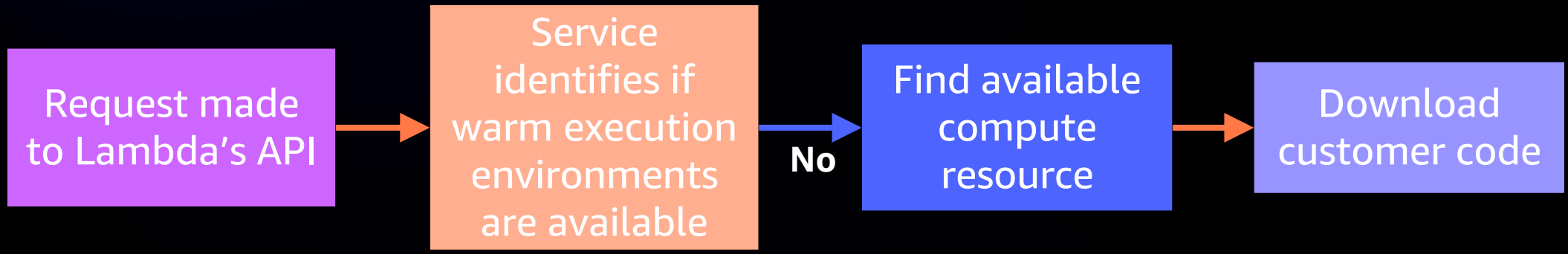
Function lifecycle



Function lifecycle

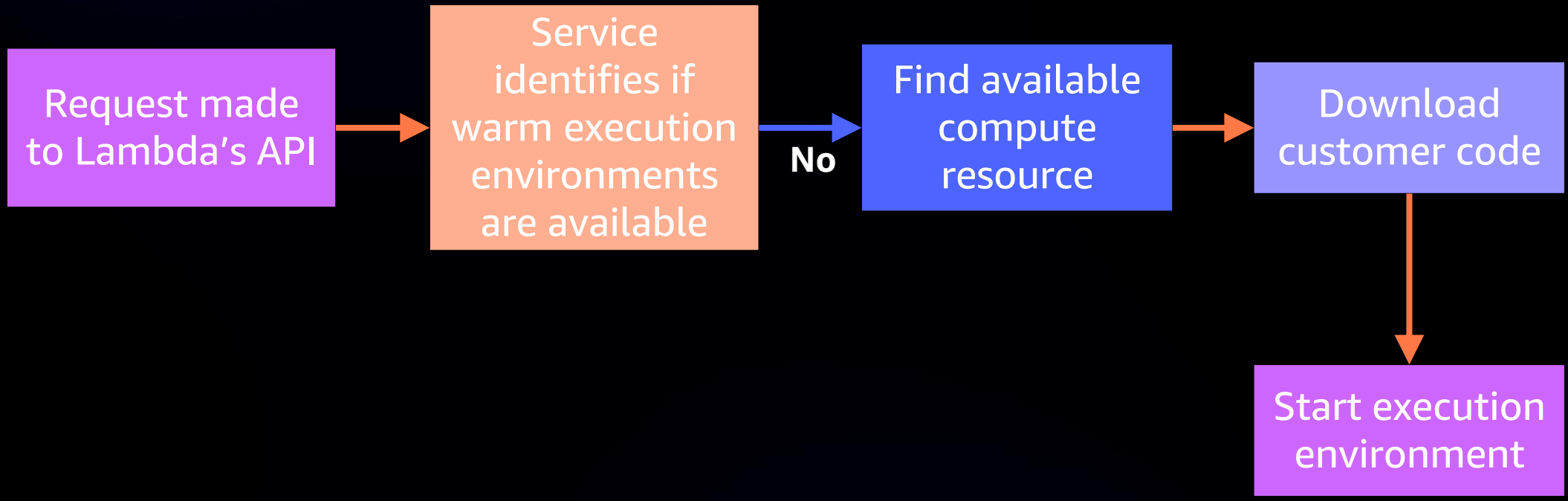
"Full" cold start

Function lifecycle



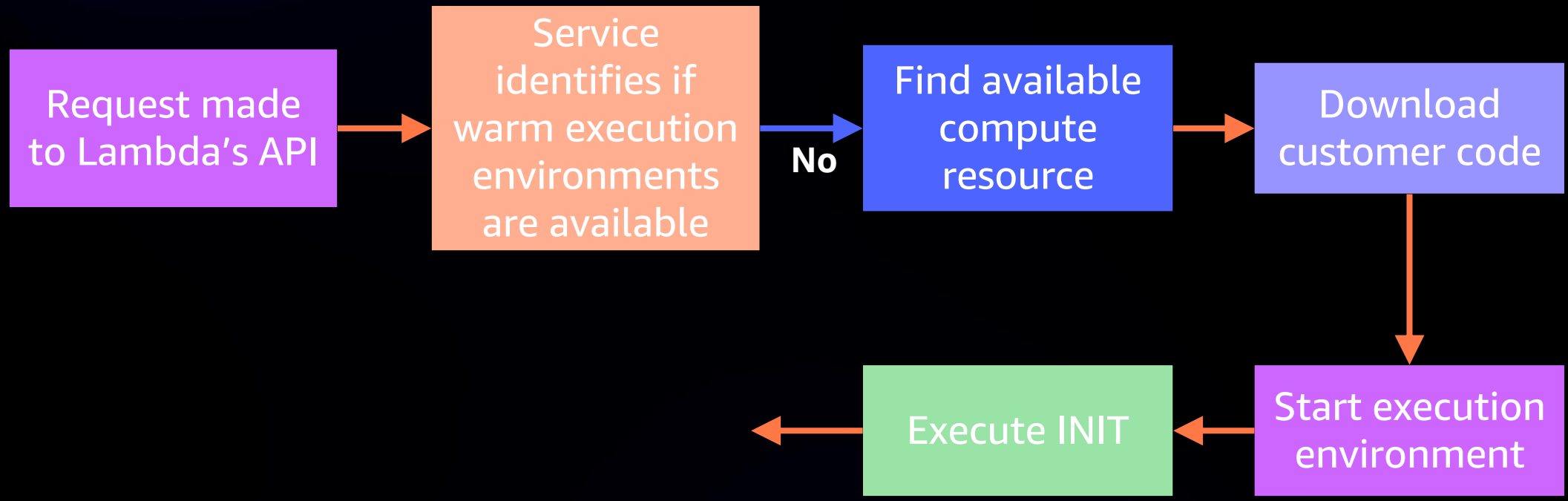
"Full" cold start

Function lifecycle



"Full" cold start

Function lifecycle



"Full" cold start

Anatomy of an AWS Lambda function

```
Function myhandler(event, context) {
    <Event handling logic> {
             result = SubfunctionA()
        }else {
                 result = SubfunctionB()

    return result;
}
```

Anatomy of an AWS Lambda function

```
Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}
```

Your handler

Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}
```

Your handler

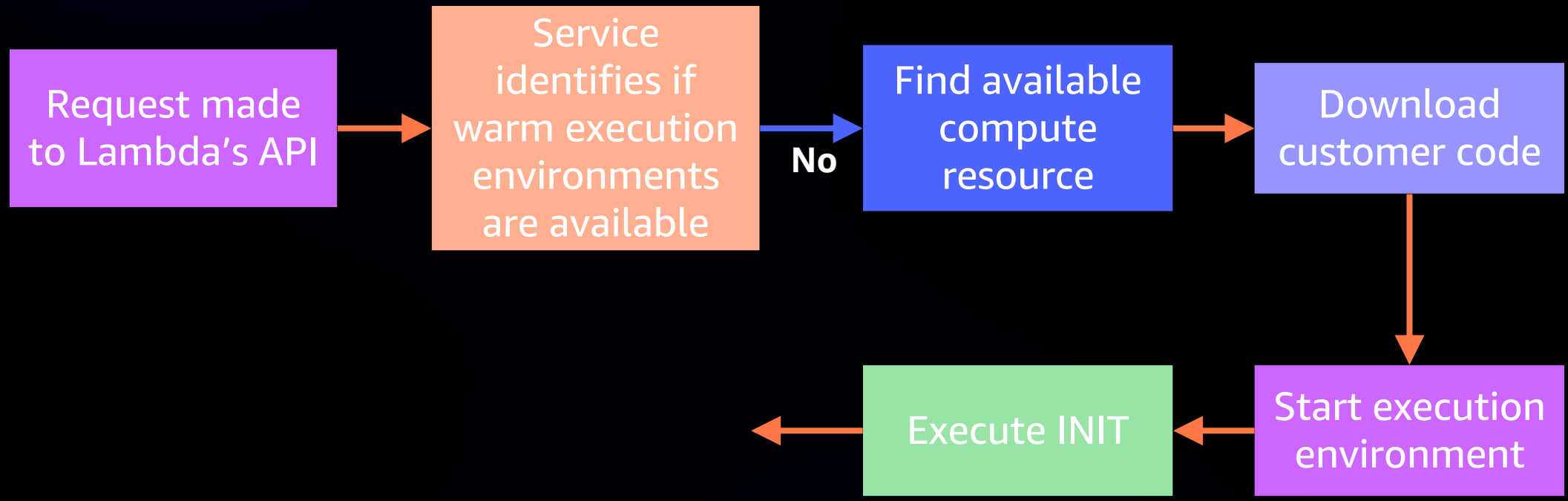
Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}
```

Dependencies, configuration information, common helper functions

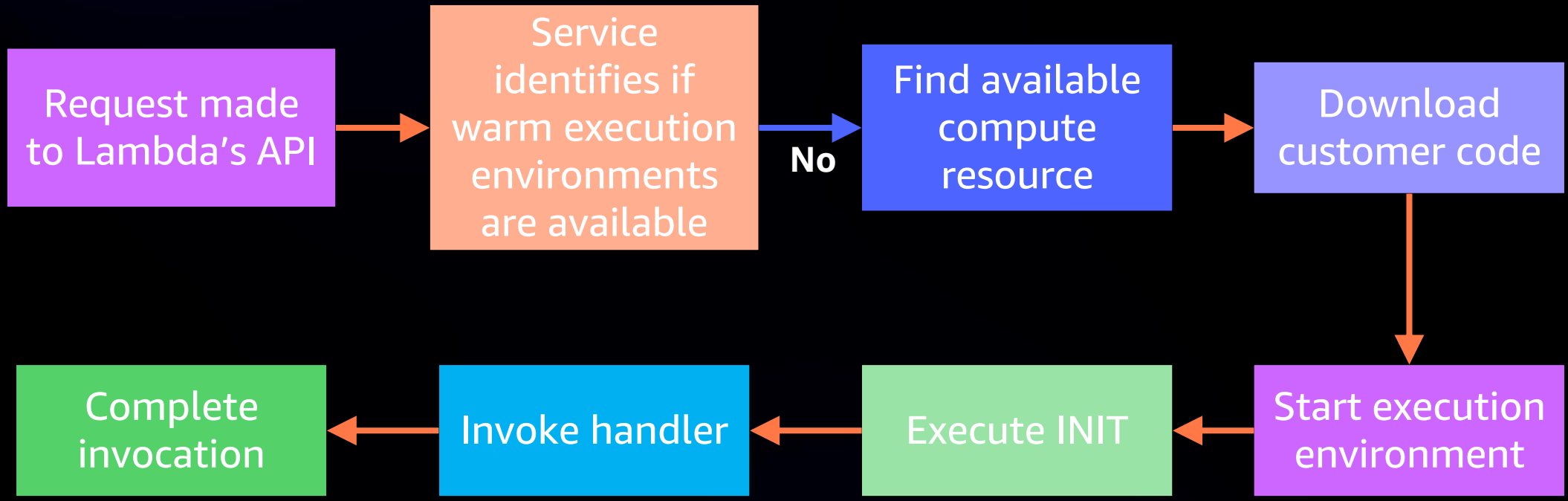
Your handler

Function lifecycle



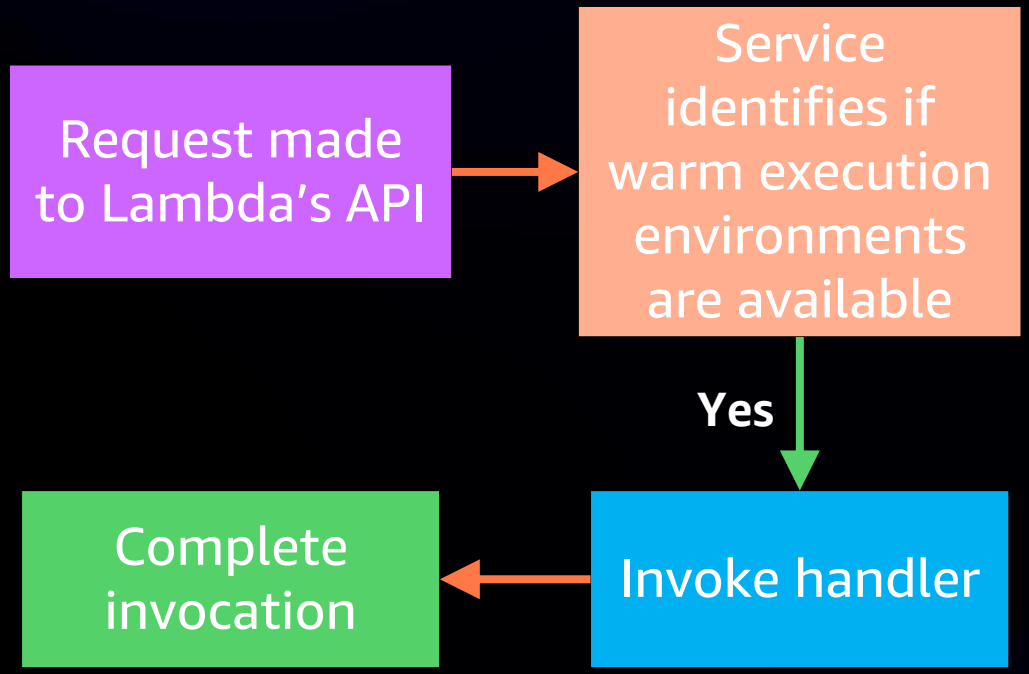
"Full" cold start

Function lifecycle



"Full" cold start

Function lifecycle



Subsequent invokes could then follow the warm path

Warm start

Measurement: How and what

You now know that you can use Amazon CloudWatch, AWS X-Ray, and AWS Partner tools to get in-depth information about how your functions and applications are performing

Q: How do you begin to test new configurations for best performance?

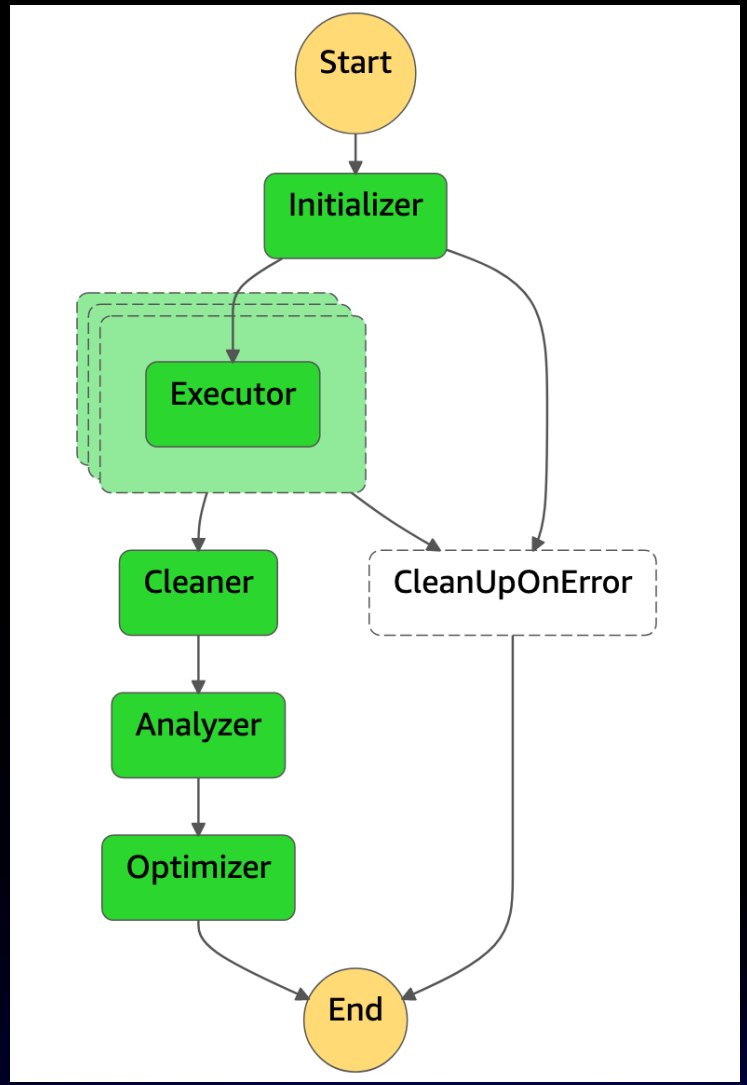
Comparing with Lambda Power Tuning

Meet AWS Lambda Power Tuning

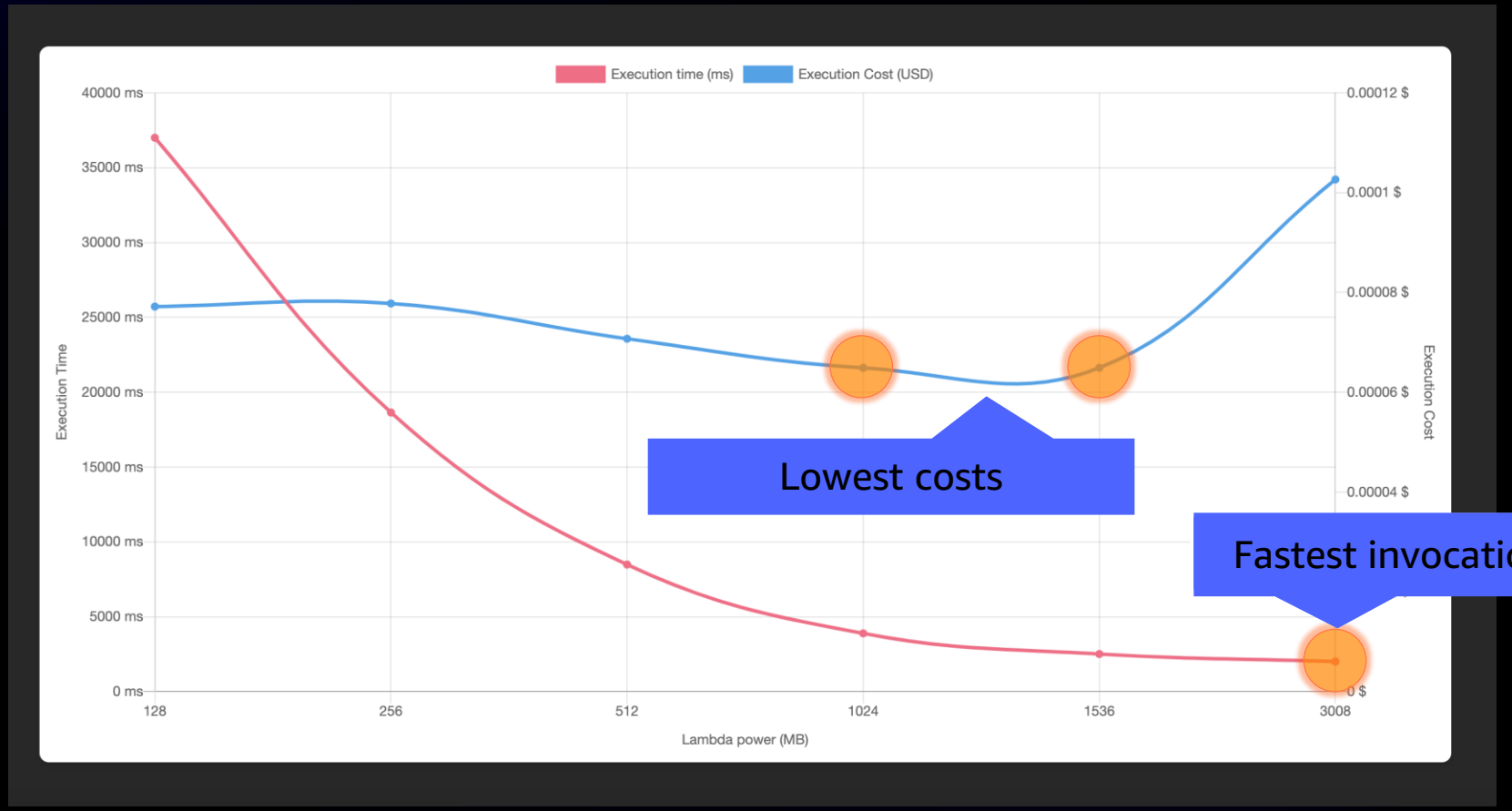
Data-driven cost and performance optimization for AWS Lambda

Features:

- Compare multiple configurations
- Uses AWS Step Functions to automate resource creation and testing
- Easy deployment with AWS SAM
- Can run pre-/post-flight scripts, pass different payloads, configure desired outcome strategy (i.e., lowest cost, fastest execution, best mix)

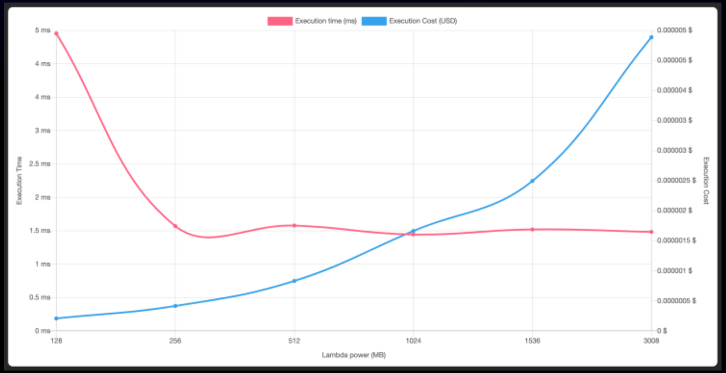


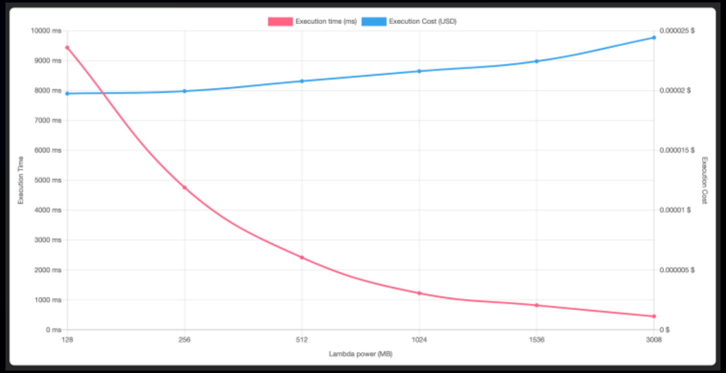
AWS Lambda Power Tuning (visualization)

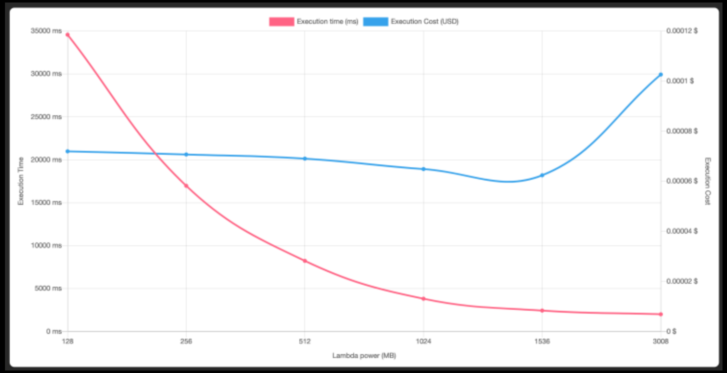


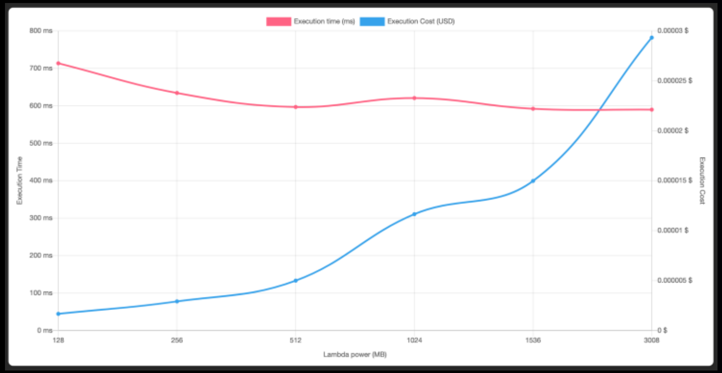
https://github.com/alexcasalboni/aws-lambda-power-tuning

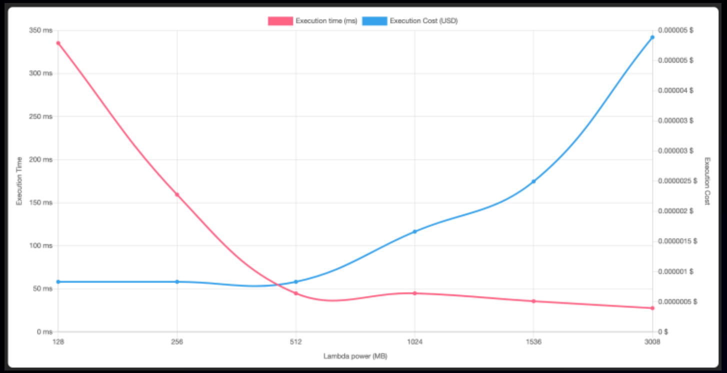
Cost/performance patterns

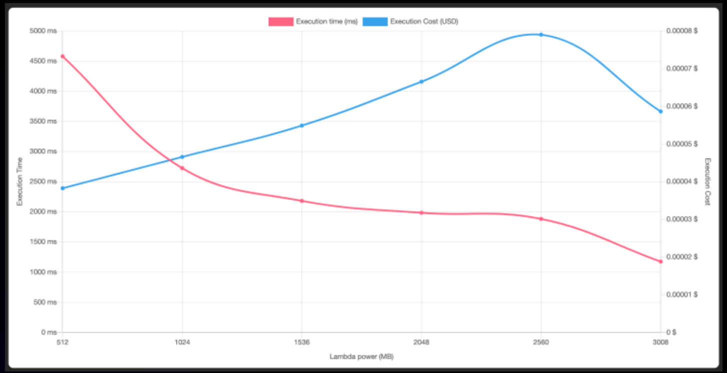












https://github.com/alexcasalboni/aws-lambda-power-tuning

“Using Lambda Power Tuning in coordination with the data from observability tools is the key to performance tuning smartly.”

Me, here today

(I just really like using the quote slide template once per talk)

aws

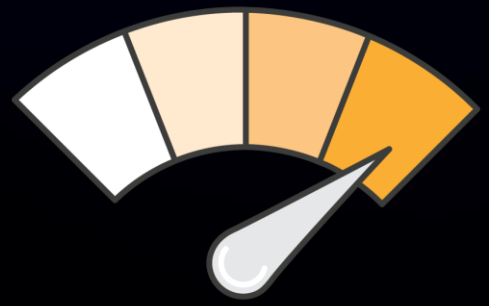
Comparing function configurations

You now know how to measure your performance of and test various configurations of your functions for the best outcome (speed, cost, or a mix)

Q: What knobs and levers exist to tune the performance of your functions?

Function configuration: CPU and memory

Tweak your function's computer power



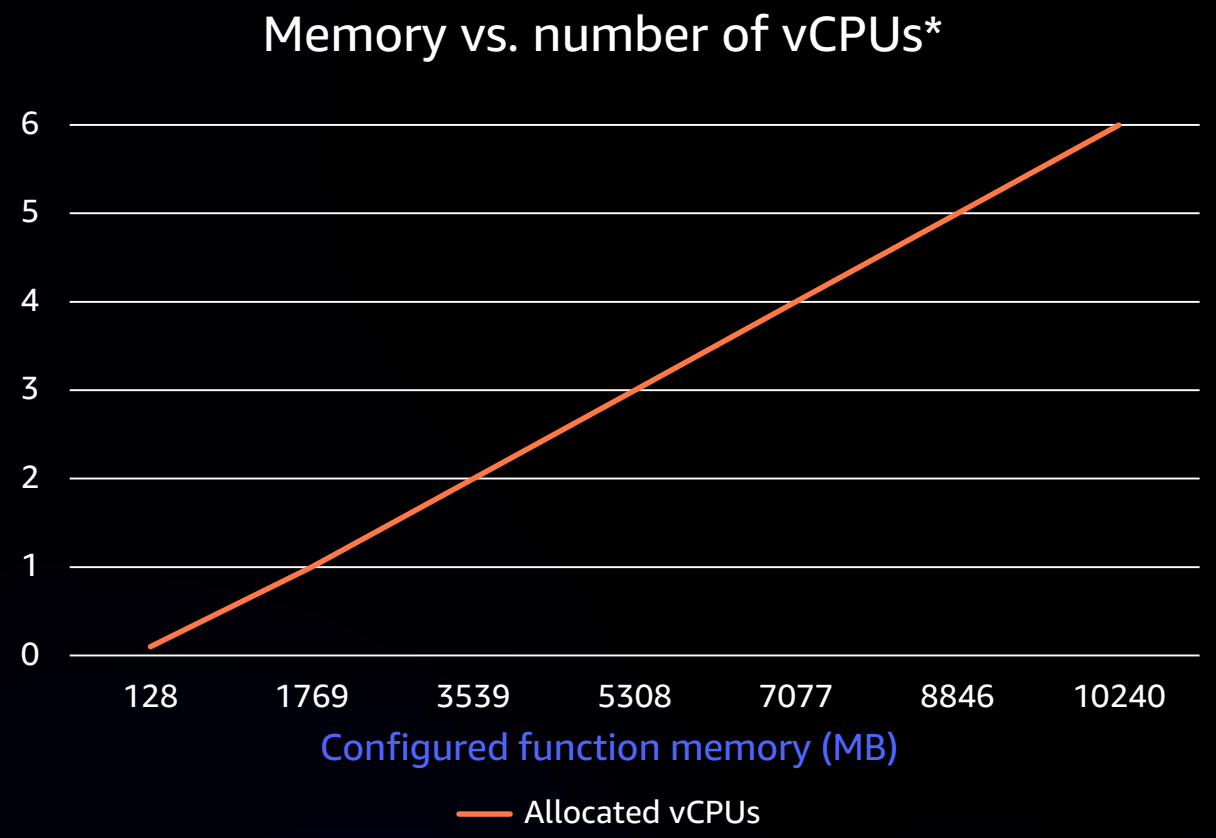
Lambda exposes only a memory control, with the **% of CPU core and network capacity** allocated to a function proportionally

Is your code CPU-, network-, or memory-bound?
If so, it could be **cheaper** to choose more memory

aws

Memory configuration for your function

- Developers can configure Lambda functions for: **10 GB** in memory with up to **6 vCPUs** proportional to memory configuration
- Build compute-intensive workloads: Machine learning, genomics, gaming, and HPC applications
- Build memory-intensive workloads: Batch, ETL, analytics, media processing

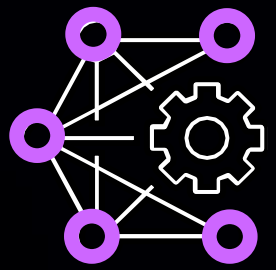


*Numbers are an approximation of vCPU power

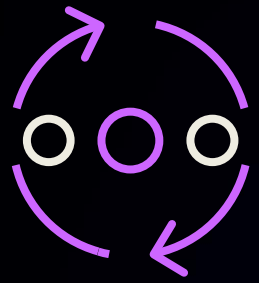
aws

AWS Graviton processors

Custom AWS silicon with 64-bit Arm Neoverse cores



Targeted optimizations for cloud-native workloads



Rapidly innovate, build, and iterate on behalf of customers

Lambda functions powered by AWS Graviton2

Arm/Graviton = up to **34% better priced performance** over x86-based Lambda

Highest performance
vs. x86

20% lower cost
vs. same-sized
Lambda functions

Up to 34% better priced performance

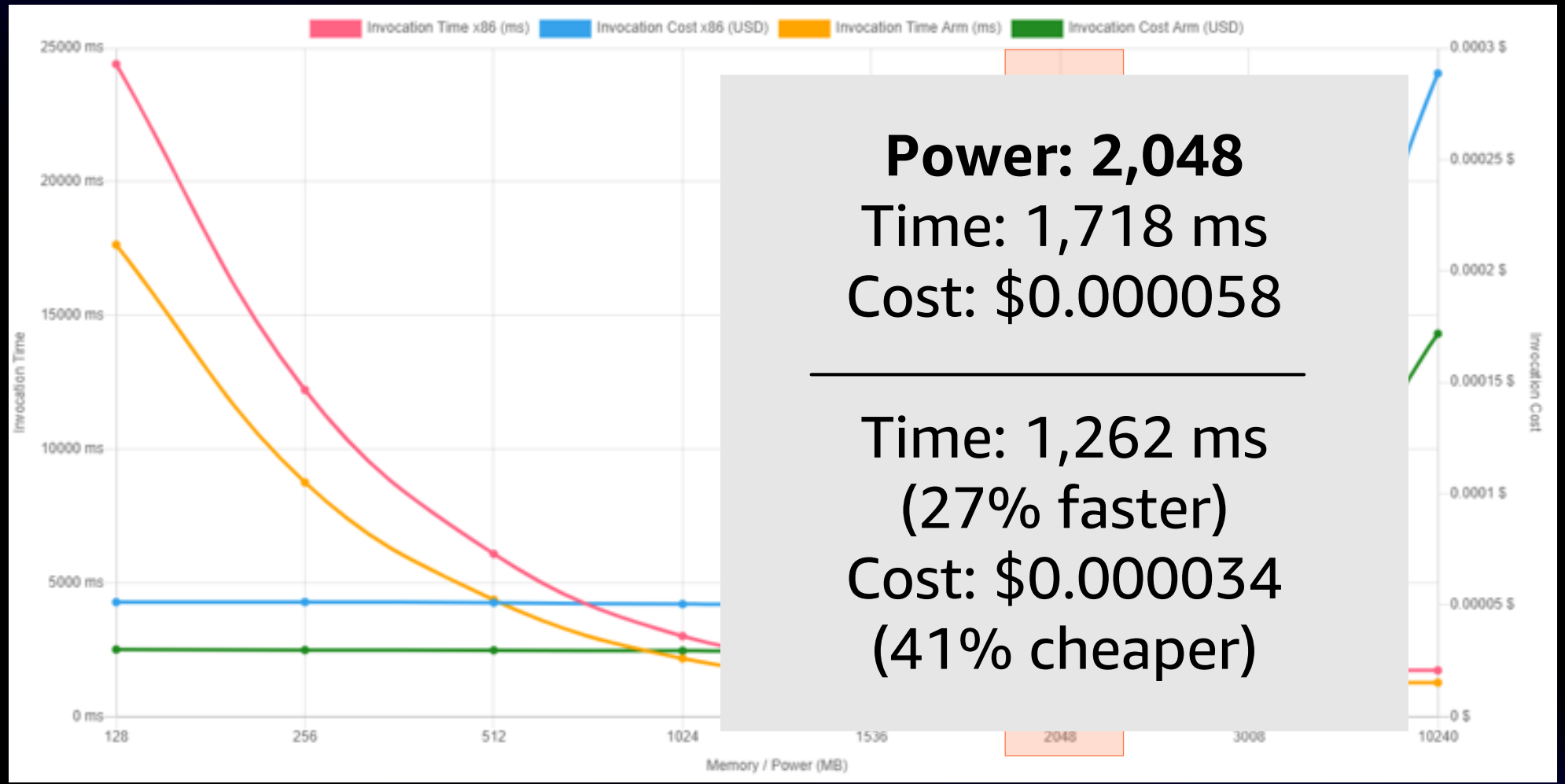
aws

Arm64: Languages, toolkits, and runtimes

Interpreted and compiled bytecode languages can run without modification

- Compiled languages need to be recompiled for arm64
- Lambda container images need to be rebuilt for Arm
- Most AWS tools and SDKs support Graviton2 transparently:
  - AWS CLI v1, AWS CLI v2
  - SDKs for C/C++, Node.js, Python, Go, Java, .NET

Comparing x86 to Arm/Graviton2



AWS Lambda – Advanced Vector Extensions 2 (AVX2) support in Intel x86

Allows CPUs to perform a higher number of integer and floating-point operations per clock cycle

Python – libraries can be compiled with the AVX2 flag or linked with MKL to take advantage of AVX2

Java – Java's JIT compiler can auto-vectorize code to run with AVX2 instructions

Golang – use the GCC compiler for Go, GCCGO

Node – use the AVX2-enabled or MKL-enabled versions of libraries

aws

AWS Lambda – AVX2 support

For vectorizable algorithms, this can enhance performance, resulting in lower latencies and higher throughput

Filter	No AVX2	With AVX2	Performance improvement
Bilinear	105 ms	71 ms	32%
Bicubic	122 ms	73 ms	40%
Lanczos	136 ms	77 ms	43%

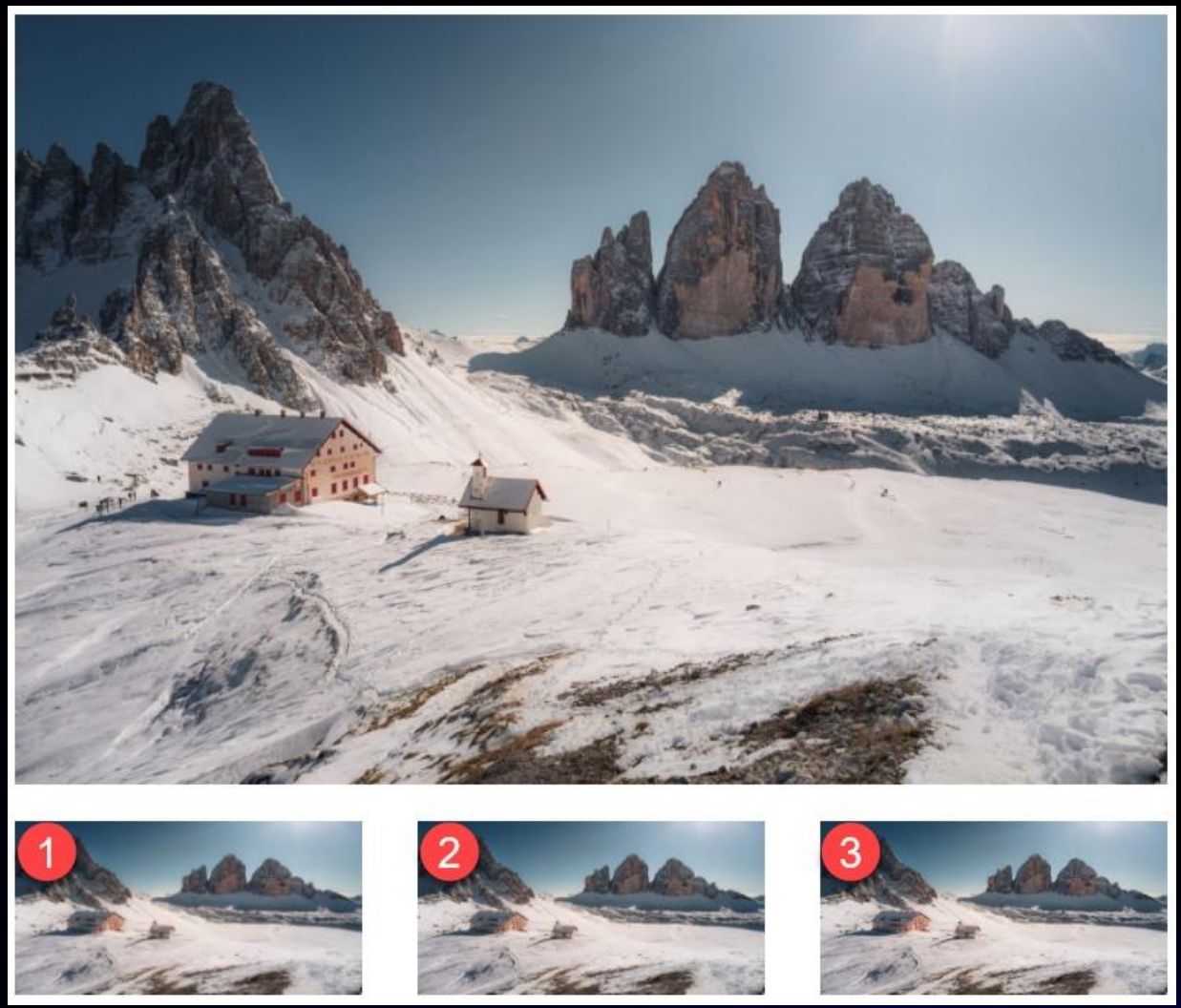


Image source: https://unsplash.com/photos/IMXhx6qhvf0. Photo credit: Daniel Seßler.

Function configuration: CPU and memory

You now know that turning up the memory knob gives you a proportional increase in CPU power

You can also select either Intel x86 or AWS Graviton2 processors

Lastly, specific capabilities of x86 processors can bring their own benefits

Q: What are the other aspects of my function that I can tune for performance?

Cold starts, warm starts, no starts?

Cold starts and you

A cold start is what occurs when a system needs to create a new resource in response to an event/request

For Lambda:

- Happens when new execution environments are needed to handle requests
- Typically <1% of all invokes for “production workloads”
  - As measured by functions with consistent invokes over a period of time (aka, not dev), excludes rarely invoked
  - Can vary from <100 ms to >1 second

Cold starts and you

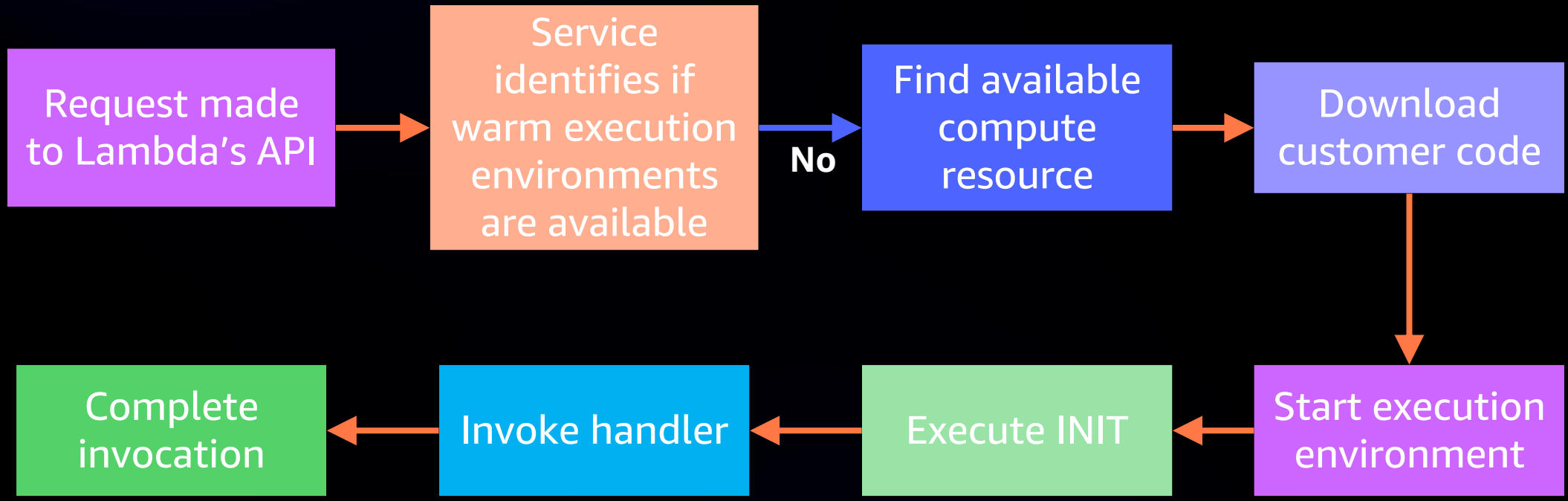
A cold start is what occurs when a system needs to create a new resource in response to an event/request

For Lambda:

- Happens when new execution environments are needed to handle requests
- Typically <1% of all invokes for “production workloads”
  - As measured by functions with consistent invokes over a period of time (aka, not dev), excludes rarely invoked
  - Can vary from <100 ms to >1 second

Why is there variance?

Function lifecycle



"Full" cold start

Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}
```

Dependencies, configuration information, common helper functions

Your handler

Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}

Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}
```

Dependencies, configuration information, common helper functions

Your handler

Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich
Pre-handler-secret-getter()
Pre-handler-db-connect()
```

Dependencies, configuration information, common helper functions

```
Function myhandler(event, context) {
    <Event handling logic> {
            result = SubfunctionA()
        }else {
                result = SubfunctionB()

    return result;
}
```

Your handler

```
Function Pre-handler-secret-getter() {
}

Function Pre-handler-db-connect(){
}
```

Common helper functions

Anatomy of an AWS Lambda function

```
Import sdk
Import http-lib
Import ham-sandwich
Pre-handler-secret-getter()
Pre-handler-db-connect()
```

Dependencies, configuration information, common helper functions

Data shows that the longest cause of delay before handler execution comes from INIT/pre-handler code

INIT – Pre-handler code, dependencies, and variables

Executes at the initial start of an execution environment

Import only what you need

- Where possible trim down SDKs and other libraries to the specific bits required

Pre-handler code is great for establishing connections, but be prepared to then handle reconnections in further executions

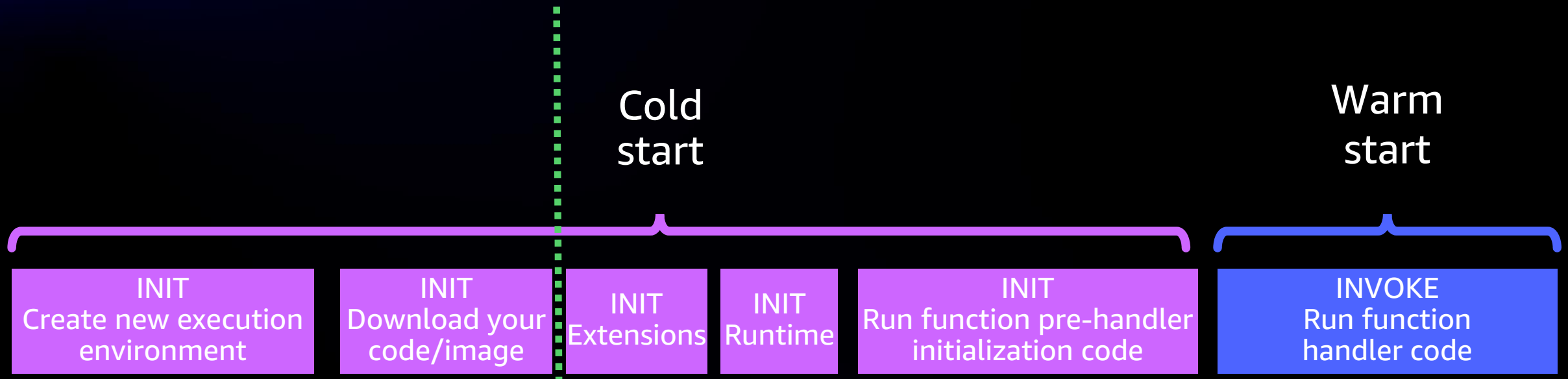
Remember that execution environments are reused

- Lazily load variables in the global scope
- Don’t load it if you don’t need it – cold starts are affected
- Clear out used variables so you don’t run into leftover state

```
Import sdk
Import http-lib
Import ham-sandwich

Pre-handler-secret-getter()
Pre-handler-db-connect()

Function myhandler(event, context) {
....
```

aws

AWS Lambda execution environment lifecycle

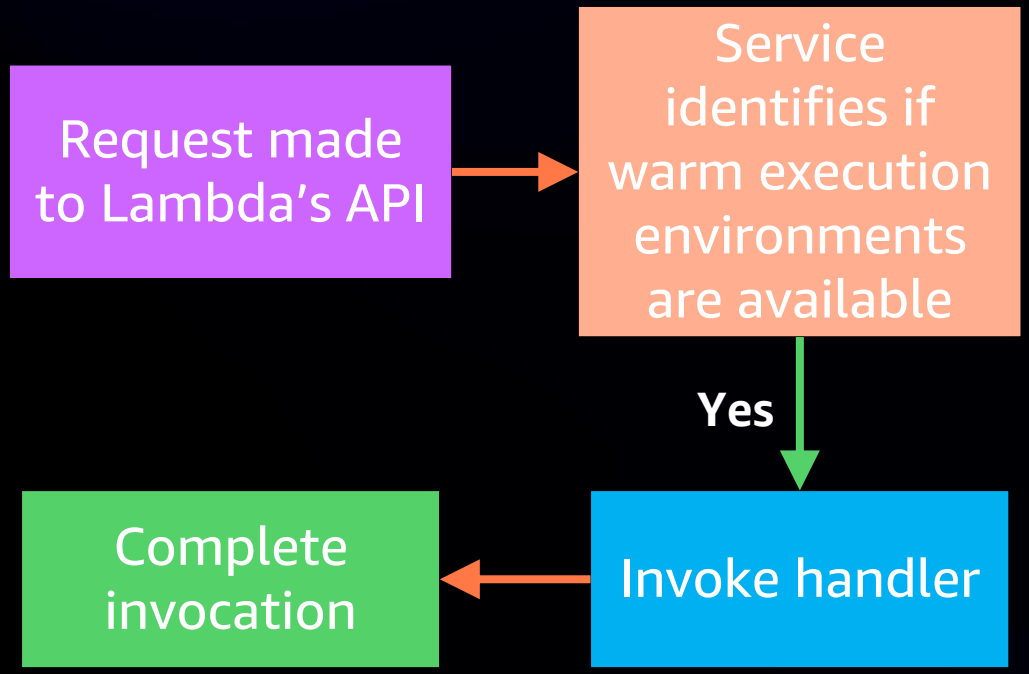


AWS optimization

Your optimization

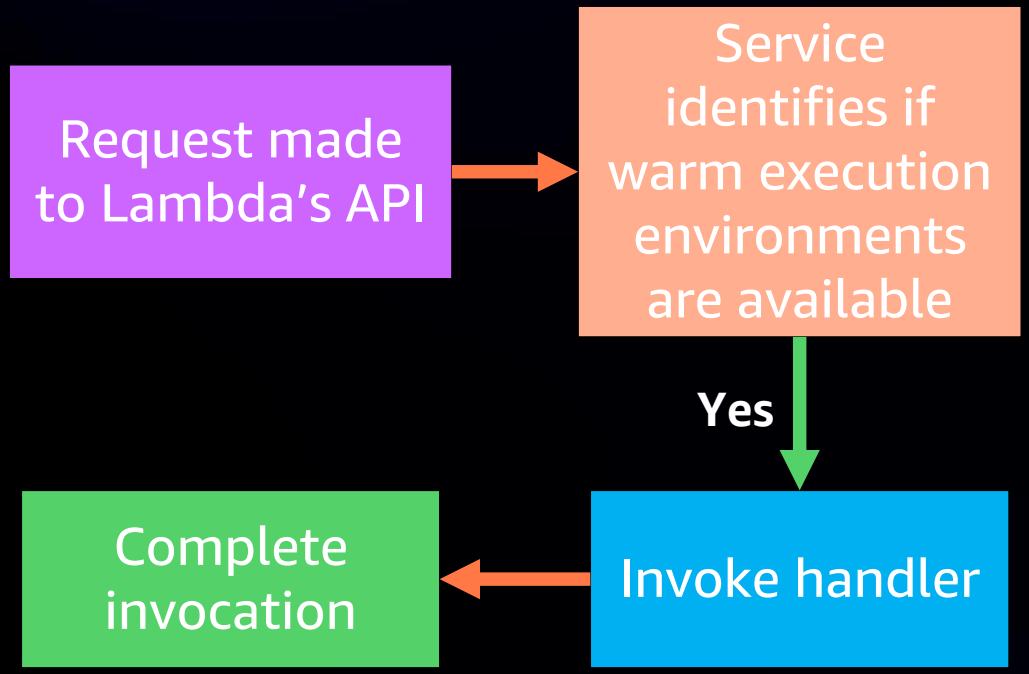
When using extensions/runtime modifications

Function lifecycle



Warm start

Function lifecycle



This is the path that most invocations end up taking in applications with somewhat consistent traffic patterns – a small <1% of requests would cause a cold-start and then ~99% would end on warmed environments

Warm start

Other warm/cold factors

Once warm, environments do not stick around forever

- Regularly reaped every few hours to keep execution environments fresh
- Run on resources that could encounter failures
- On occasion, warm functions are ignored in order to rebalance load across Availability Zones

No affinity for repeat requests – no concept of sticky sessions or ability to target a warmed environment

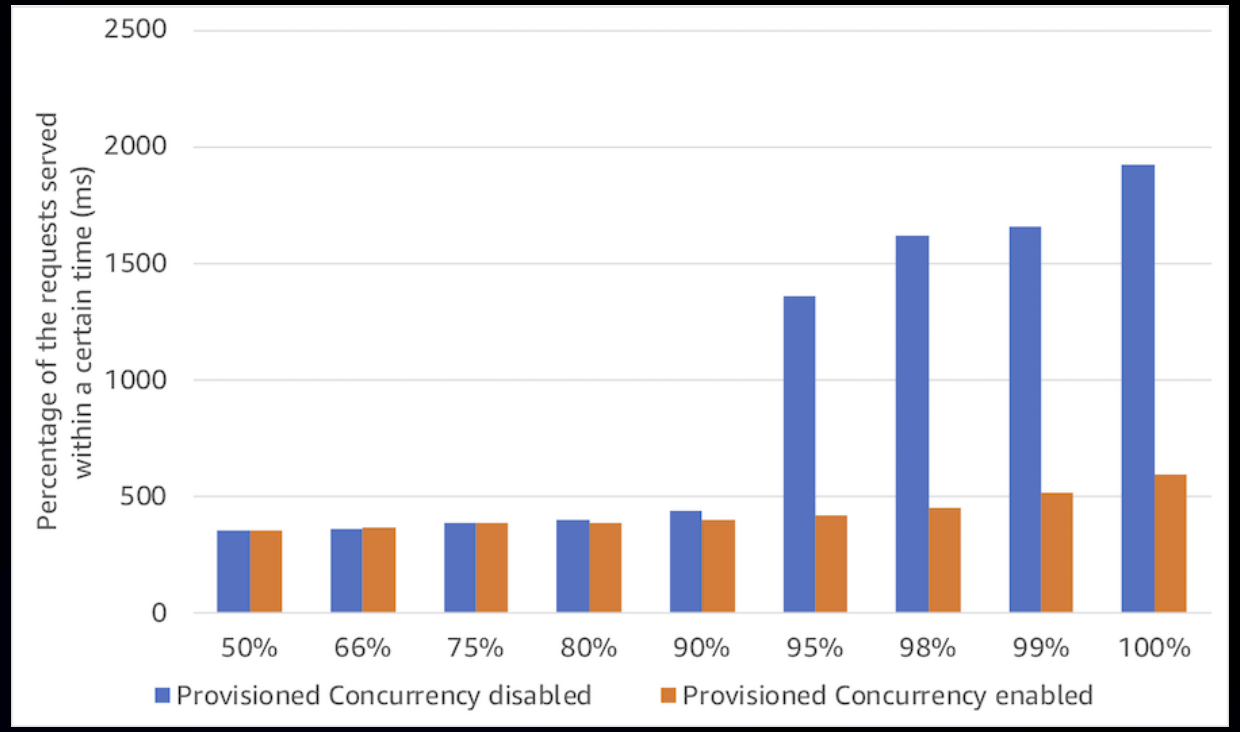
Updating your application code or changing your function configuration will cause the existing environments to be flushed and cause cold starts

Provisioned concurrency on AWS Lambda

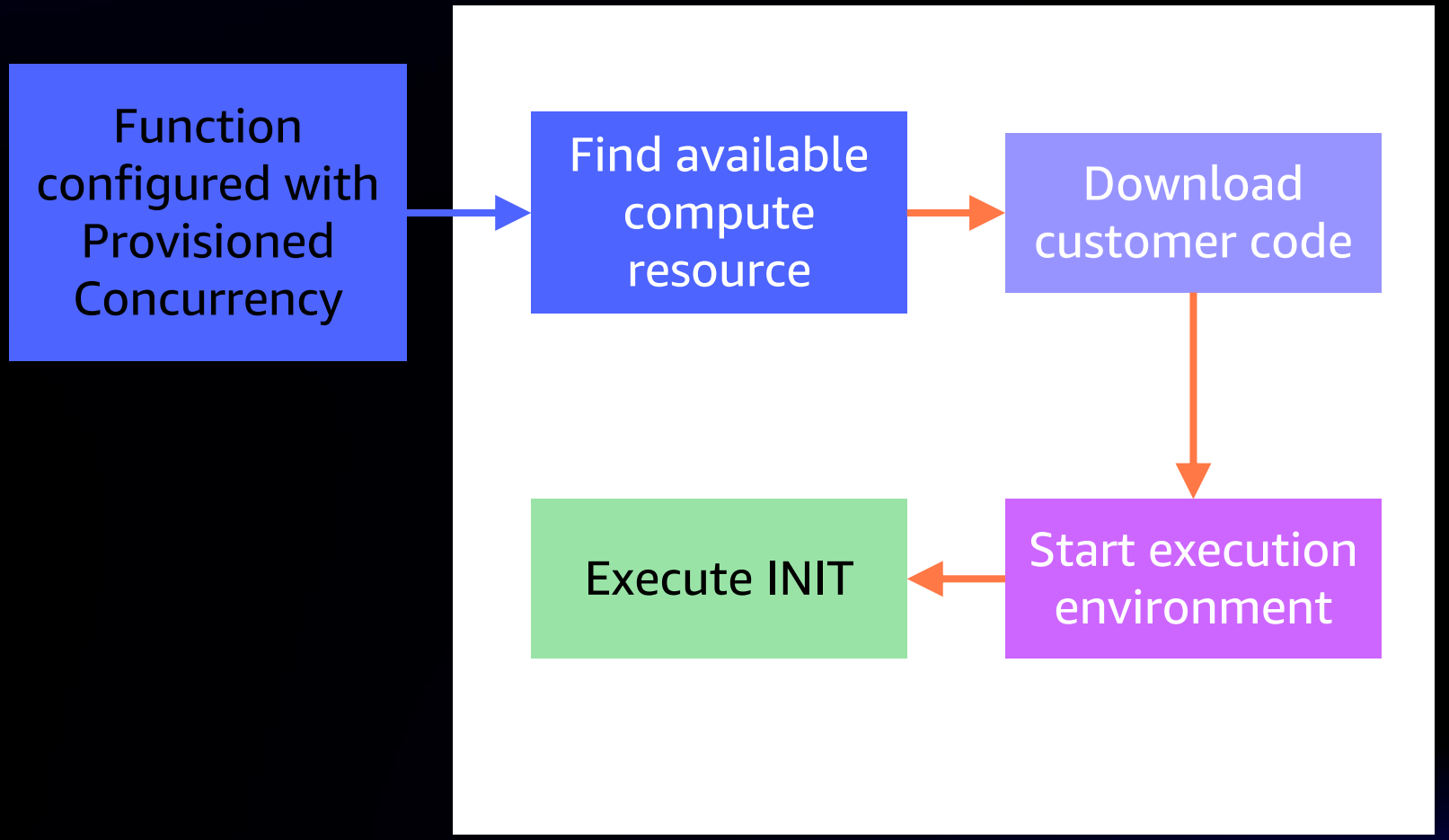
Pre-creates execution environments, running INIT code

- Mostly for latency-sensitive, interactive workloads
- Improved consistency across the long tail of performance
- Minimal changes to code or Lambda usage
- Runs on AWS Auto Scaling
- Adds a cost factor for per concurrency provisioned but a lower duration cost for invocation

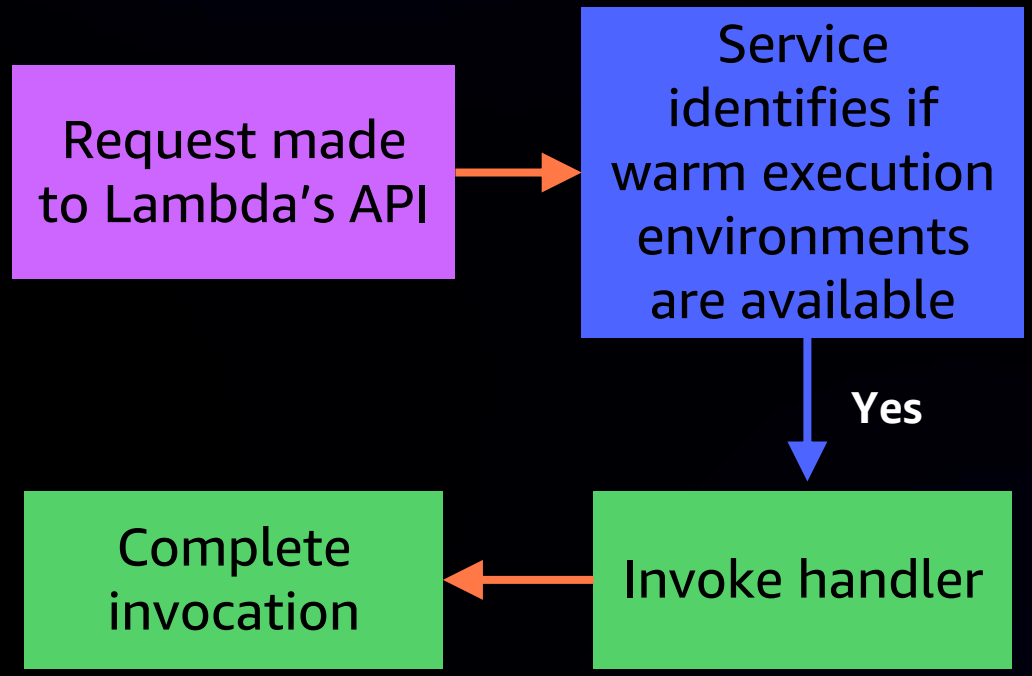
  This could save you money when heavily utilized



Function lifecycle: Provisioned concurrency start



Function lifecycle: Provisioned concurrency invoke

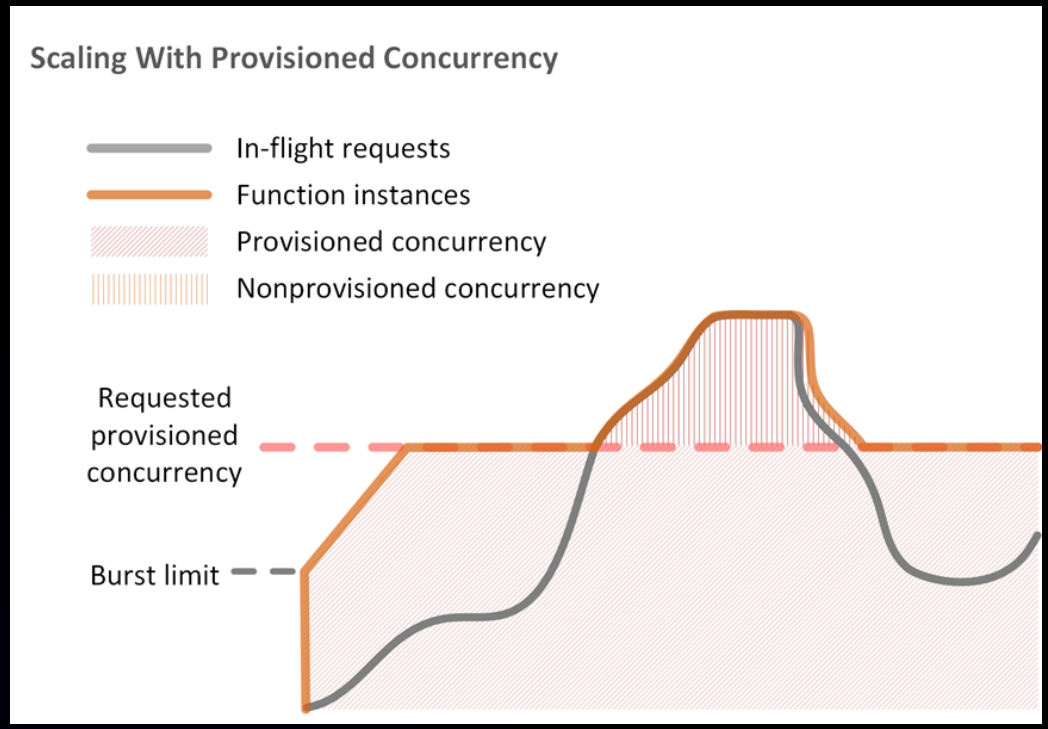


This becomes the default for all provisioned concurrency worker environments

Provisioned concurrency: Things to know

- Reduces the start time to <100 ms
- Can’t configure for $LATEST

  Use versions/aliases
- Provisioning ramp-up of 500 per minute
- No changes to function handler code performance
- Requests previously mentioned provisioned concurrency follow on-demand Lambda limits and behaviors for cold-starts, bursting, and pricing
- Still overall account concurrency per limit Region
- Wide support in serverless tools

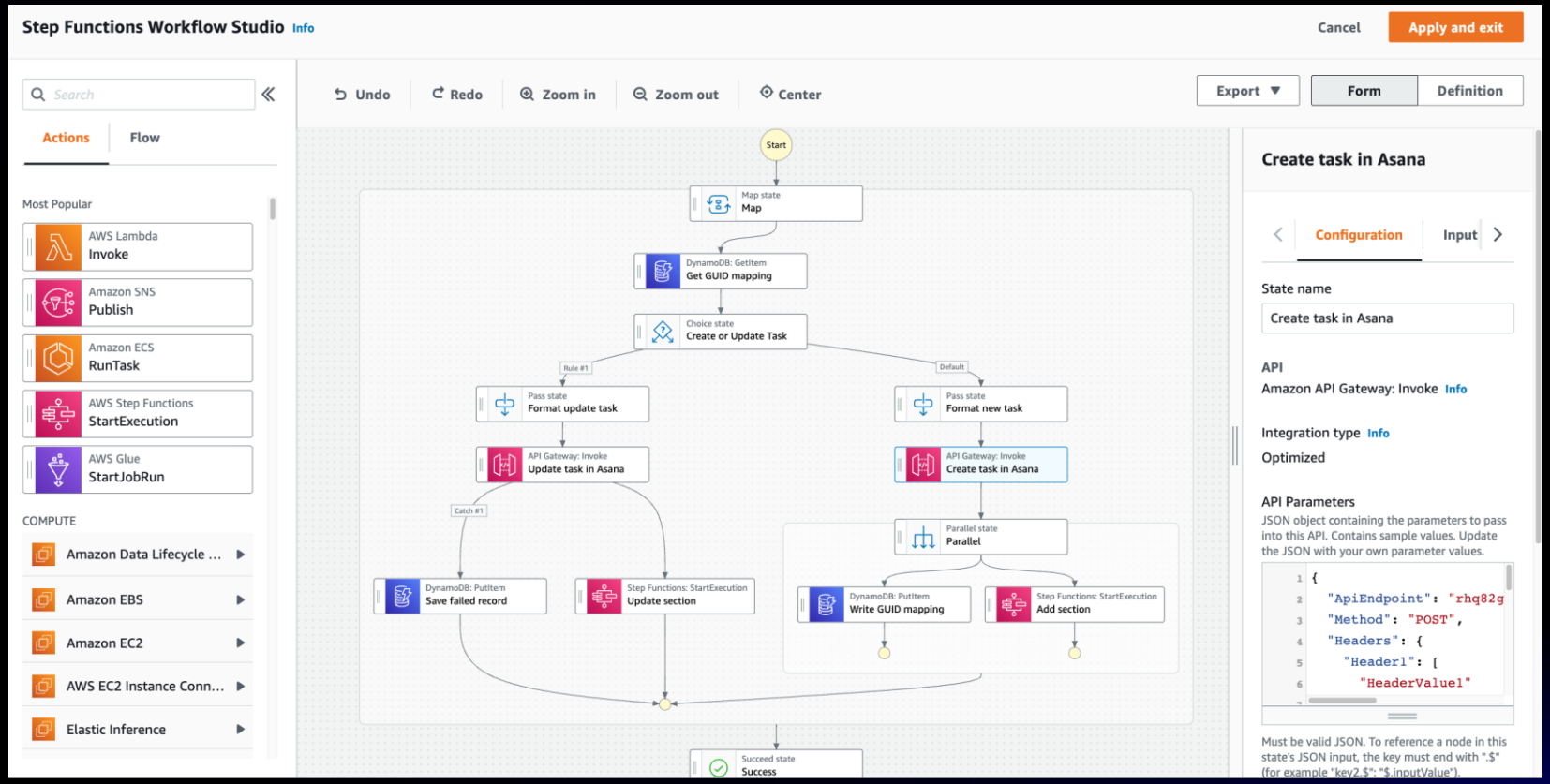


The fastest and lowest cost function is the one you don’t write

AWS Step Functions SDK integration

SUPPORT FOR OVER 200 AWS SERVICES

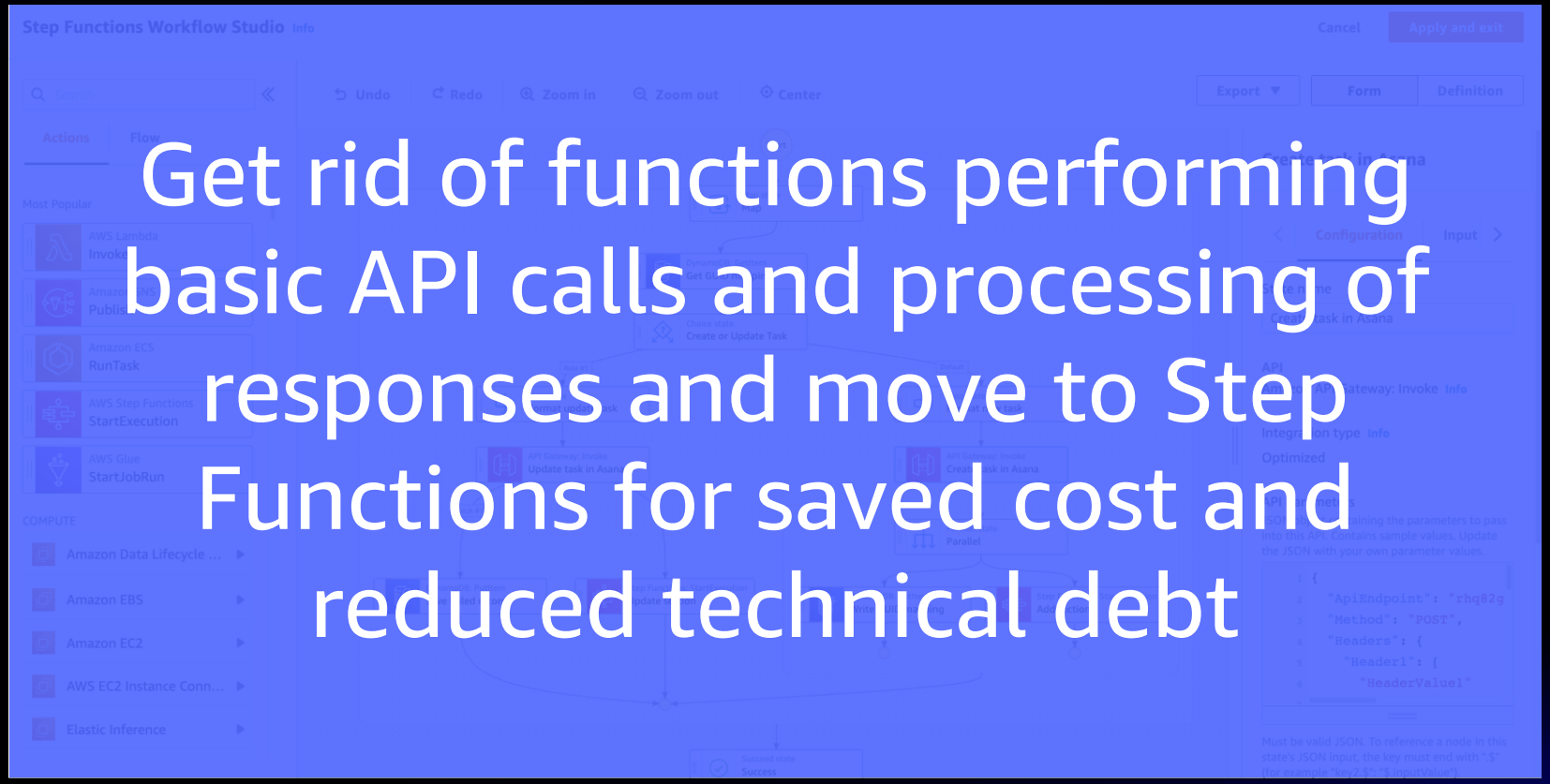
On top of the previous integrations with over 46 services you can now call almost any AWS service API from Step Functions



AWS Step Functions SDK integration

SUPPORT FOR OVER 200 AWS SERVICES

On top of the previous integrations with over 46 services you can now call almost any AWS service API from Step Functions



Cold starts, warm starts, no starts?

You now know that by using provisioned concurrency you can avoid the impact of cold starts on your function latency

Most of the pain of cold starts comes from code run in INIT, so minimize and use smartly

Lastly, don’t write the functions you don’t need!

Use services like Step Functions to reduce your code to just business logic

FIN/ACK – closing

AWS Lambda performance comes down to a few key points:

- Measurement – you can't tune what you don't have data for
- Select the right CPU for your workload
  - Utilize special CPU features if you can
- Configure the best memory for your latency/cost needs
- Don't pre-warm yourself, use provisioned concurrency

munns@amazon.com
@chrismunns

serverlesspresso

A re:Invent coffee bar built on serverless technologies by the serverless DA team

---

Dev lounge at the Venetian

Tuesday **10 AM – 6 PM**
Wednesday **10 AM – 6 PM**
Thursday **10 AM – 4 PM**

aws

Thank you!

Chris Munns

munns@amazon.com
@chrismunns