AWS re:Invent

NOV. 28 – DEC. 2, 2022 | LAS VEGAS, NV

CON314

Serverless high-concurrency containers on AWS

Nathan Peck (he/him)

Senior Developer Advocate
Amazon Web Services

What is concurrency?

Why should I care about concurrency?

Photo by Jud McCranie / CC BY-SA 4.0

Tandy 1000

No concurrency! If you wanted to run a different program, you had to turn the computer off, put a new disk in, and turn it back on

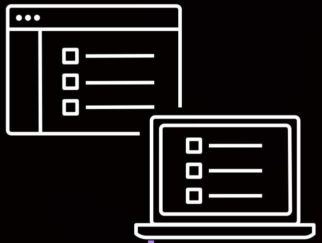
Photo by Jud McCranie / CC BY-SA 4.0

Cooperative multitasking

The foreground program is responsible for checking periodically to see if another program is asking for CPU; it can optionally yield to background programs

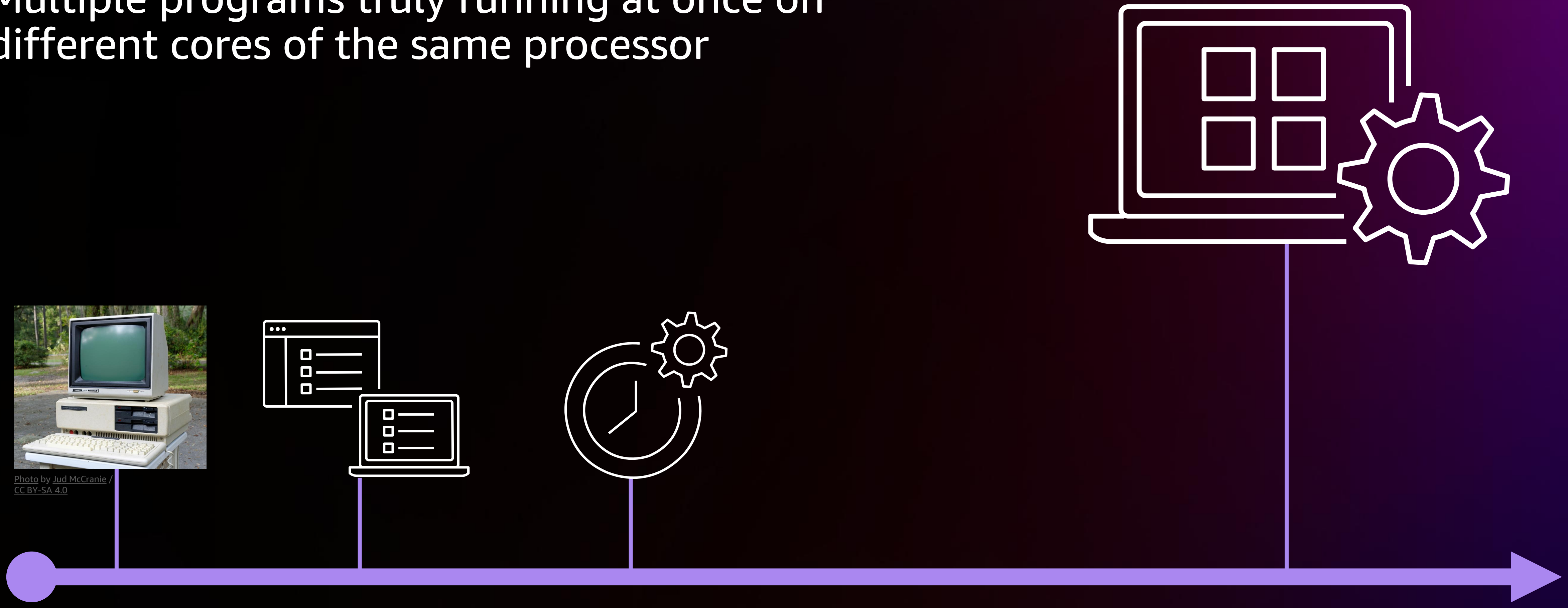
Preemptive multitasking

The operating system can proactively suspend a program and schedules which program should get a CPU time slice next



Multicore processors

Multiple programs truly running at once on different cores of the same processor



Client/server architecture

Programs communicate over the internet to server clusters where work is done across many multicore computers

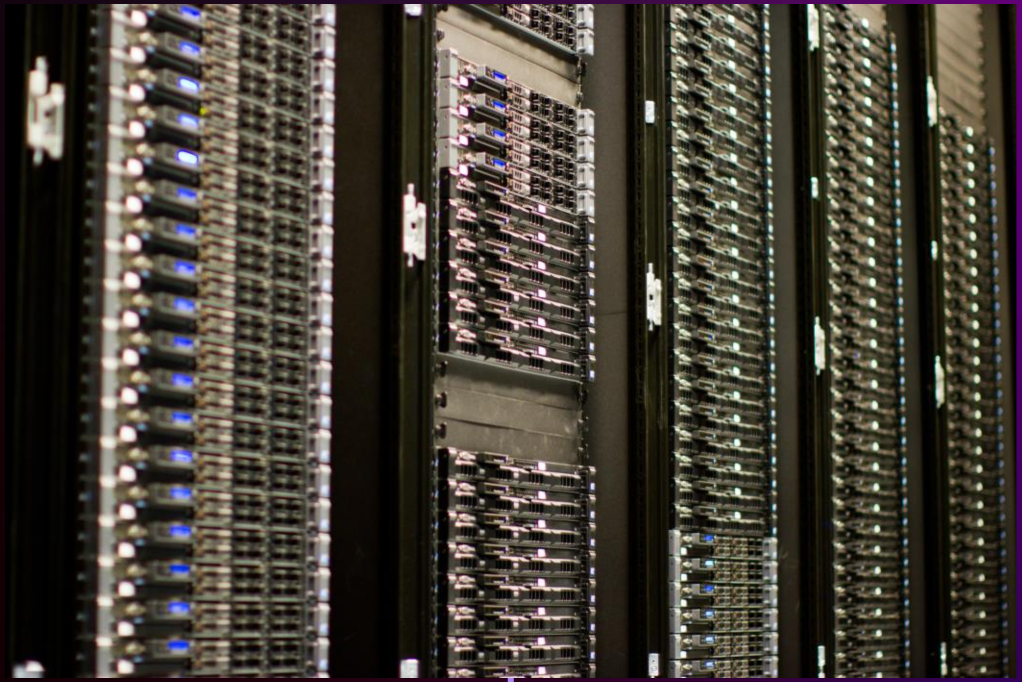
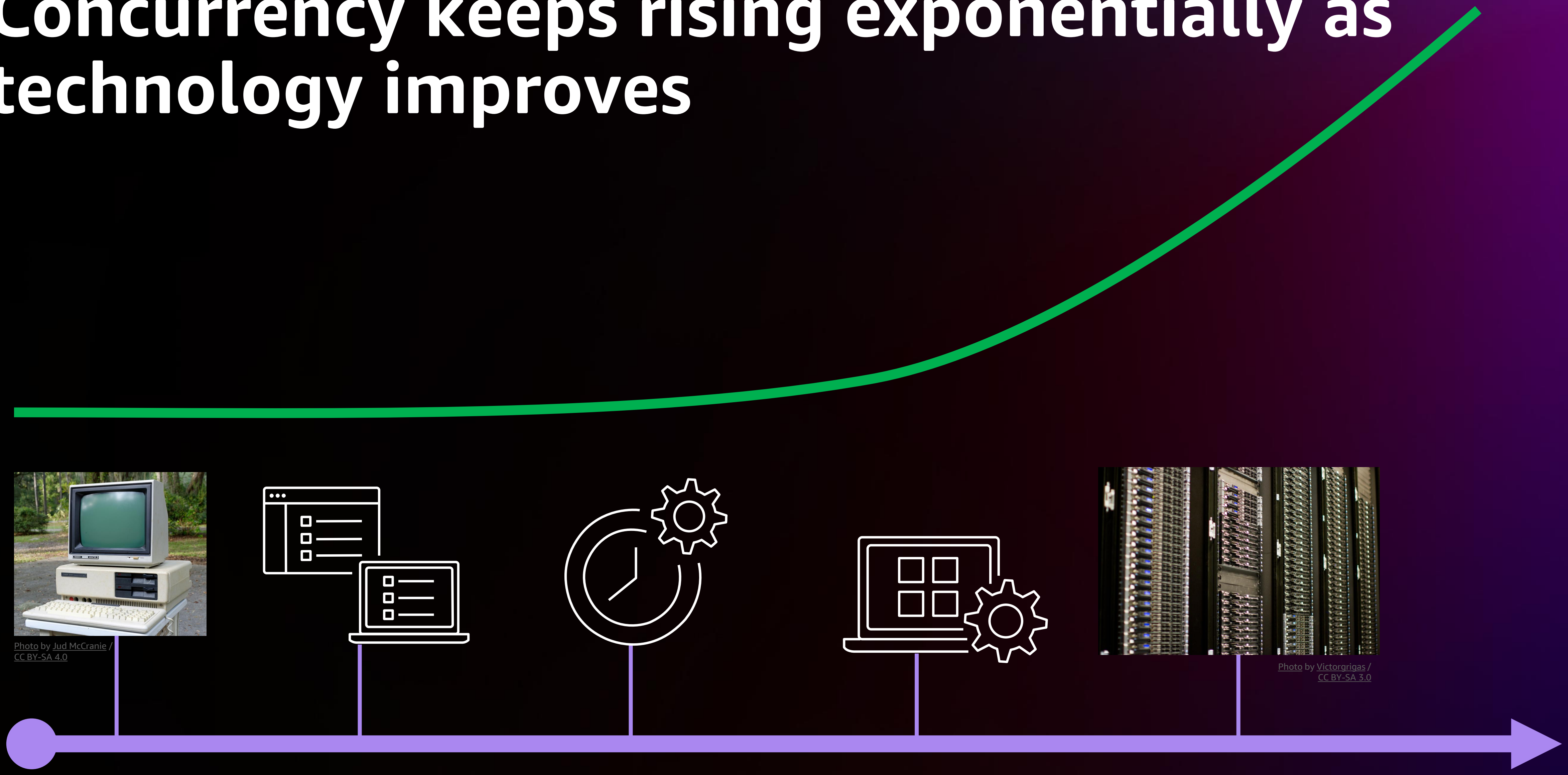


Photo by Victorgrigas / CC BY-SA 3.0

Photo by Jud McCranie / CC BY-SA 4.0

Concurrency keeps rising exponentially as technology improves



How do I build an application that can handle concurrency?

From zero users to millions

As a software builder, you need the right tools at the right time

Completely new software needs rapid development of new features

Established software needs maintenance, support, and reliability

Builders need compute optimized for low concurrency and compute optimized for high concurrency

Low-concurrency applications

Still searching for product market fit; need easy development, low operational costs, and low baseline costs

High-concurrency applications

Wildly successful, viral adoption; need large amounts of compute at cheap sustained usage rates

Serverless container technologies help you start out small but then scale out when needed

AWS Lambda

AWS App Runner

AWS Fargate

AWS Lambda

Your containerized event handling function in the cloud

AWS Lambda runs your function in the cloud

Handler function

```
exports.handler = (event, context, callback) => {
  callback(null, "Hello world!");
}
```

Different ways to run the code: Events

AWS Lambda

Invoke code by calling the AWS Lambda API

Handler function

```
exports.handler = (event, context, callback) => {
  callback(null, "Hello world!");
}
```

Amazon EventBridge

Invoke code when a specific event happens – upload to an S3 bucket, for example

Different ways to run the code: Web requests

Application Load Balancer

Amazon API Gateway

Lambda function URL

Handler function

```
exports.handler = (event, context, callback) => {
  callback(null, "Hello world!");
}
```

Firecracker microVM

Handler function

```
exports.handler = (event, context, callback) => {
  callback(null, "Hello world!");
}
```

Firecracker microVM

Handler function

```
exports.handler = (event, context, callback) => {
  callback(null, "Hello world!");
}
```

Lambda spins up multiple isolated copies of your containerized code in microVMs that are strongly isolated from each other

Lifecycle of a Lambda function instance

Cold start	Init code	Invoke function code
Downloading your container, extracting it, and loading the code for launch in its own isolated runtime sandbox	Any setup code outside your handler function	Running your handler code, waiting for it to respond back with a result

You pay per millisecond for time spent in init code and invoking the handler function

Pay per millisecond

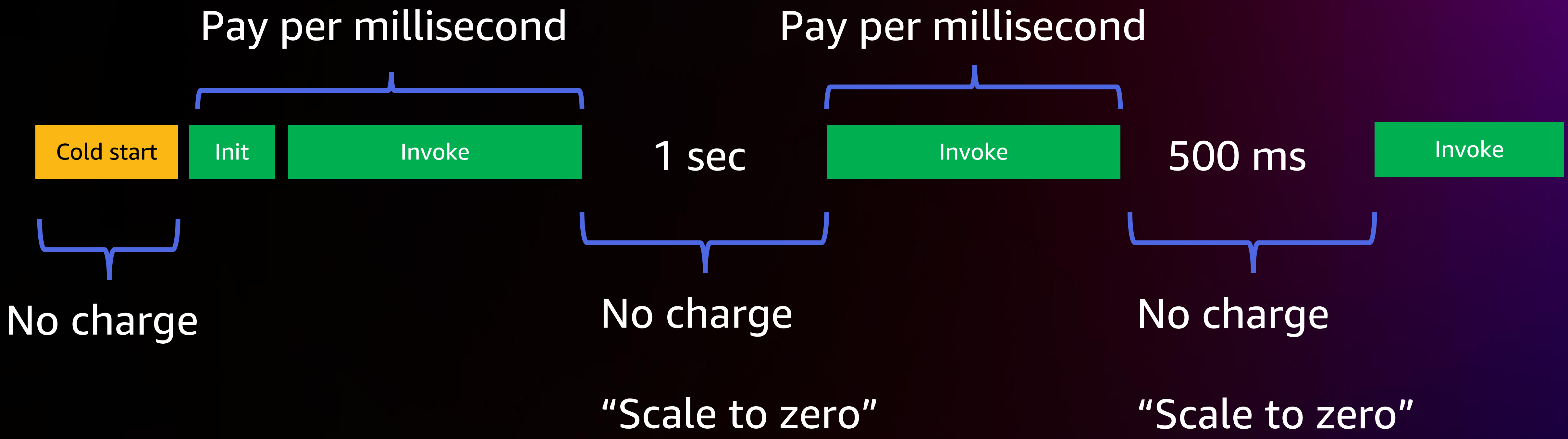
No charge

Cold start

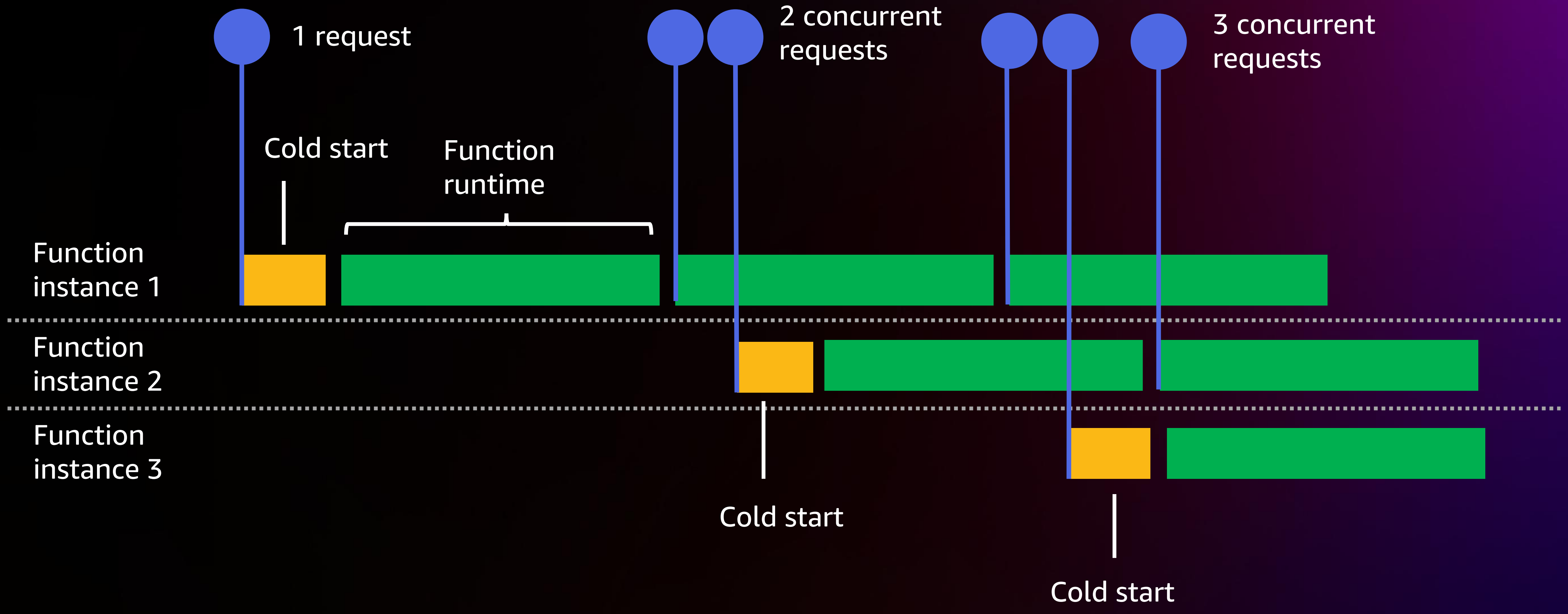
Init code

Invoke function code

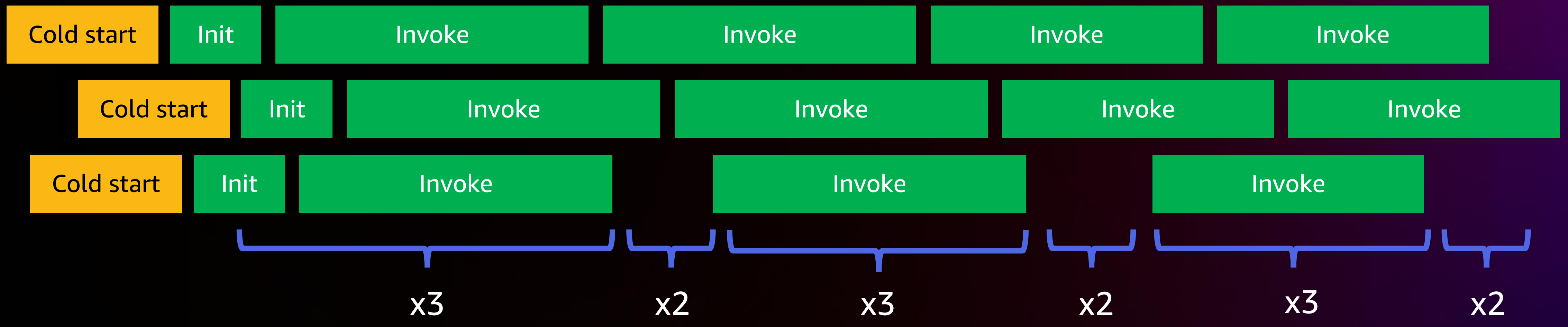
AWS Lambda is extremely cost-effective when there are gaps between work to do



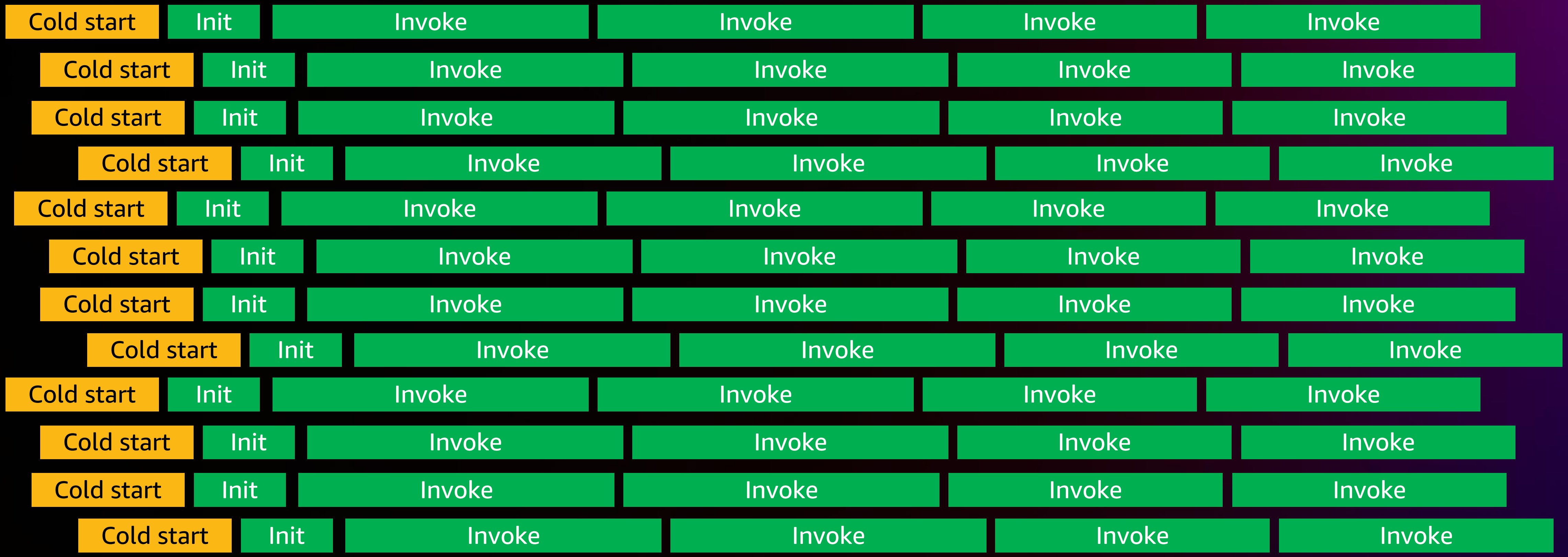
A function instance only handles one invoke at a time



When there are multiple concurrent Lambda function instances running, each one charges concurrently



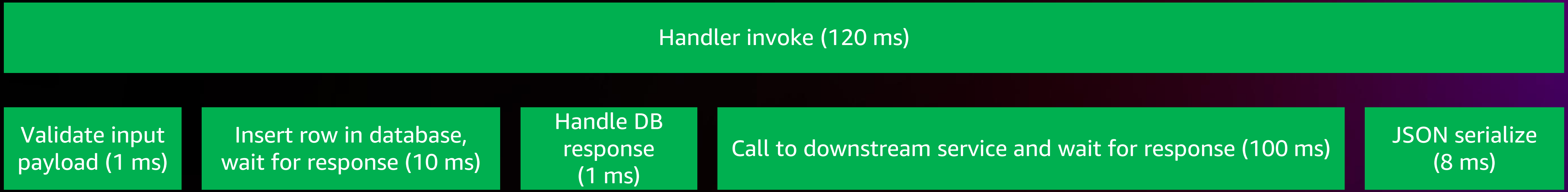
What about when there are many concurrent Lambda functions?



Concurrency deep dive

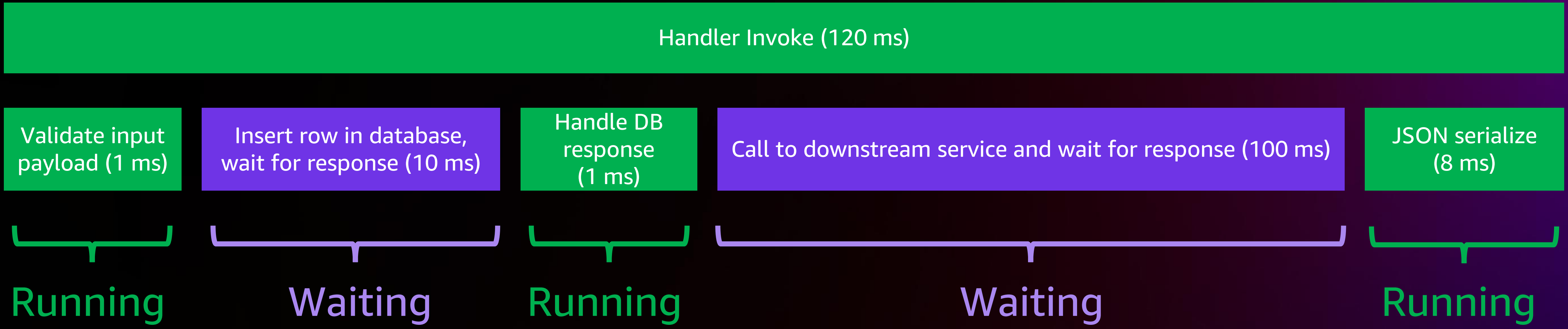
What is actually happening during a request?

What is my application actually doing?



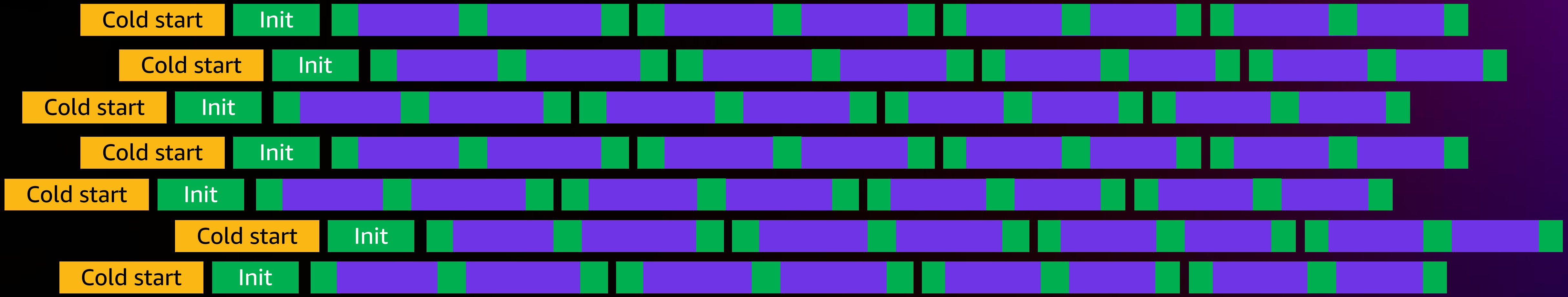
Many modern workloads are full of input/output (I/O) operations

What is my application actually doing?



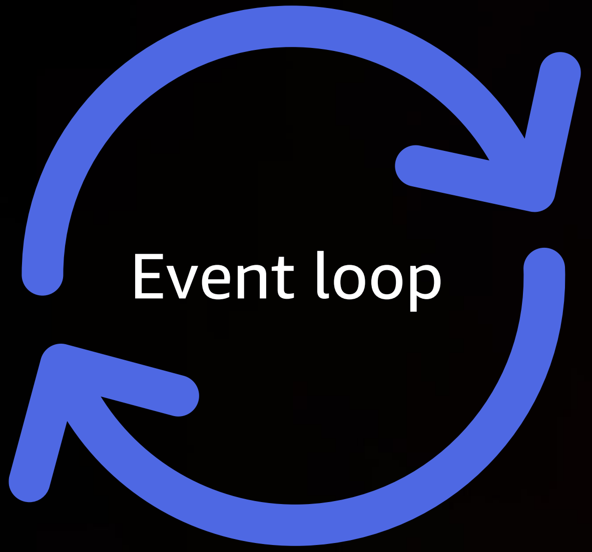
I/O heavy workloads often spend more time waiting than running CPU instructions

The more concurrent activity I have, the more milliseconds of waiting there are compared to milliseconds of running CPU instructions



What alternatives are there?

Modern application frameworks make it possible to handle multiple concurrent requests in a single application process



Validate input payload (1 ms)

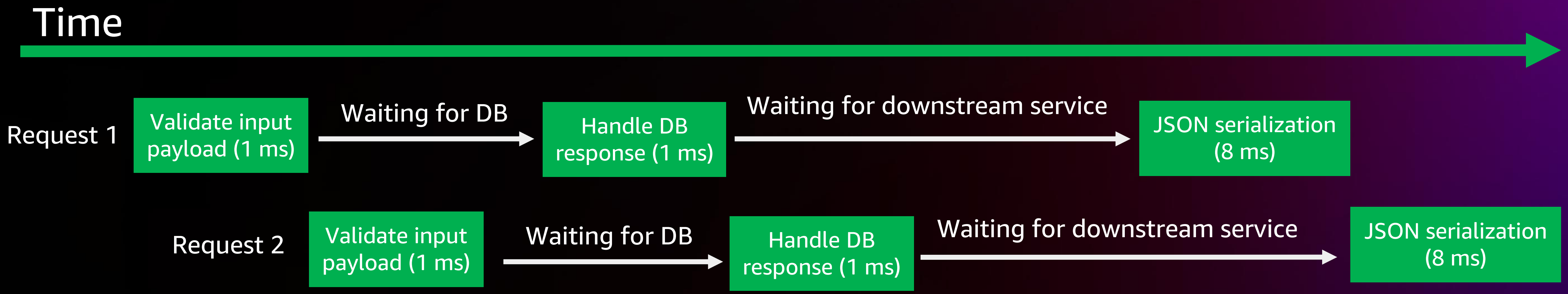
Handle DB response (1 ms)

JSON serialization (8 ms)

Each second has 1,000 ms for a CPU core to do work

Grab more work off the event loop whenever we are waiting on I/O

Instead of doing nothing while waiting, grab another event and do some work while we wait



The event loop schedules work into all 1,000 ms of time per second

A single application process can keep the CPU core busy by doing other work (like answering another HTTP request) while it waits on I/O

Node.js async/await

The await keyword yields back to the event loop to let the process start handling another request while it waits for the I/O to complete

```
exports.handler = async function (event, context, callback) {
  // Validate the event

  let dbResponse = await db.insert(); // Wait for IO from DB

  // Handle DB response

  let downstreamResponse = await downstreamService.put();

  // Serialize a JSON response
}
```

The amount of possible concurrency for a function depends on the CPU time consumed by its instructions

1 ms

10 ms

1 ms

100 ms

8 ms

```
exports.handler = async function (event, context, callback) {
  // Validate the event

  let dbResponse = await db.insert(); // Wait for IO from DB

  // Handle DB response

  let downstreamService = await downstreamService.put();

  // Serialize a JSON response
}
```

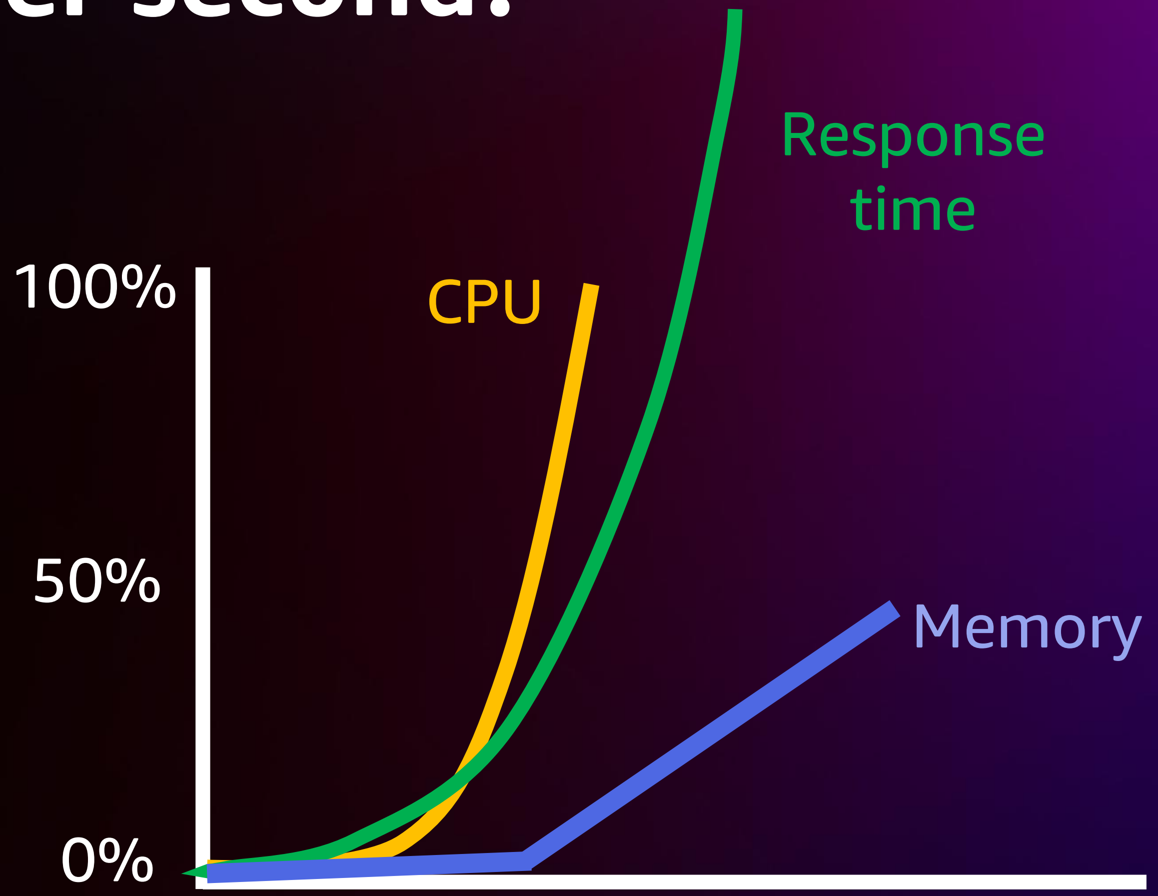
CPU time: 10 ms

1,000 ms / 10 ms = 100 function invokes/sec per single threaded app process

Downside of concurrency: What happens if you have too many requests per second?

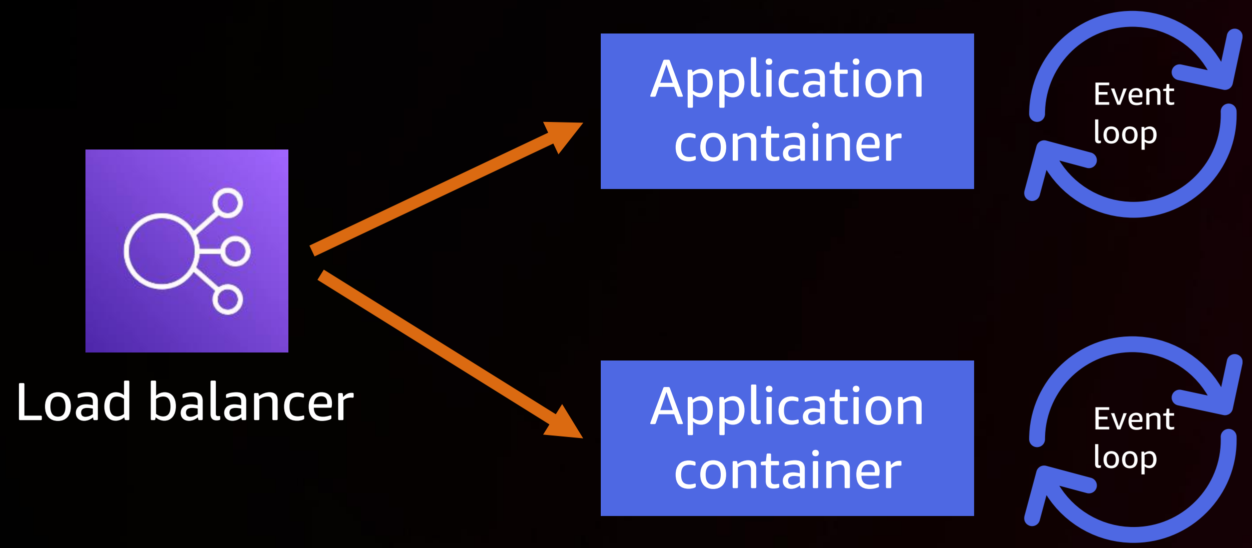
Code to run
Code to run
Code to run
Code to run
Code to run
Code to run
Code to run
Code to run
Code to run
Code to run

Work starts queuing up



As CPU saturates, the response time skyrockets

Solution: Load balance concurrency across multiple copies of the application process, each with its own event loop



Processes could be both running on the same physical machine and sharing separate cores of a multicore machine

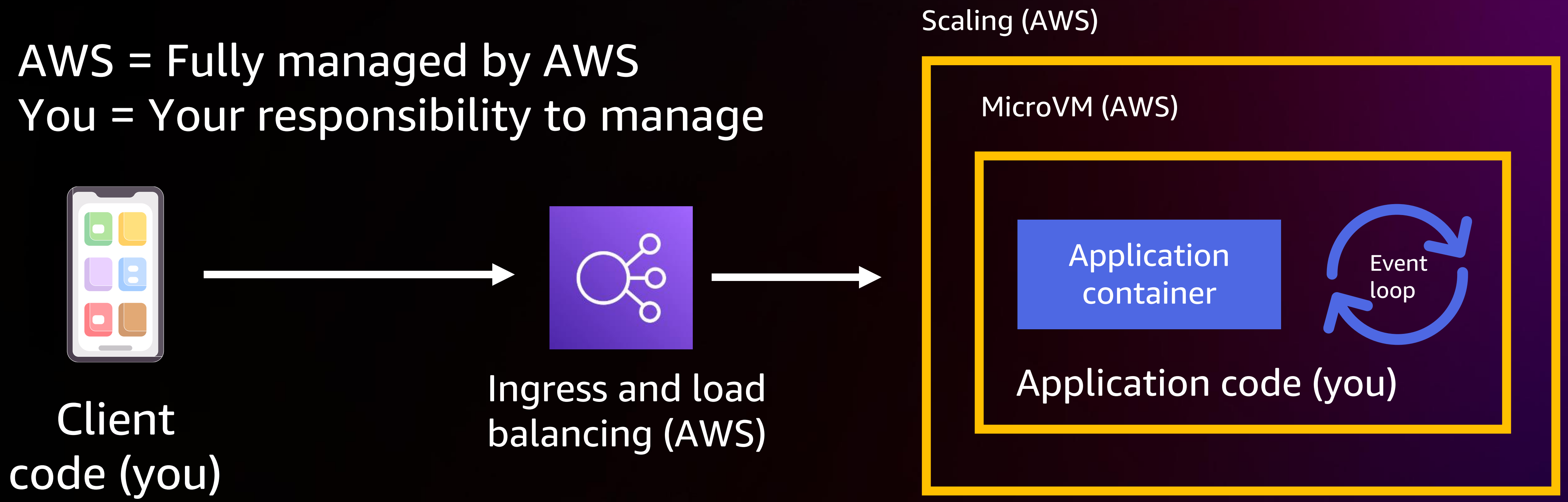
Or they could be on their own machines

AWS App Runner

Each copy of your application
handles many concurrent requests

AWS App Runner manages horizontal scaling for you

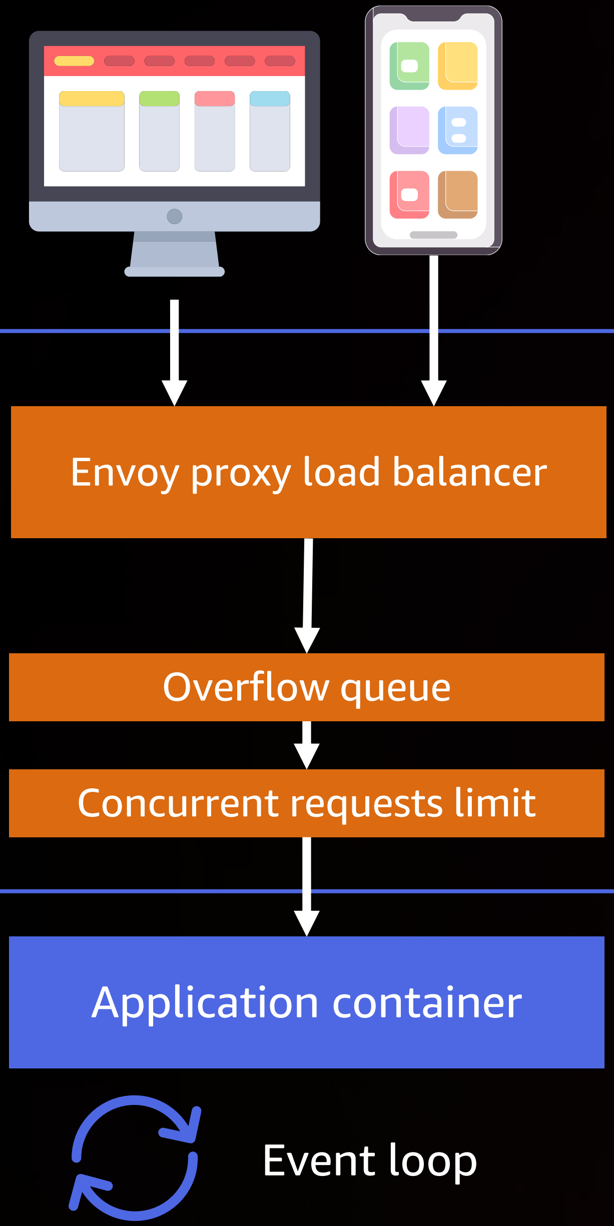
AWS = Fully managed by AWS
You = Your responsibility to manage



You

AWS

You

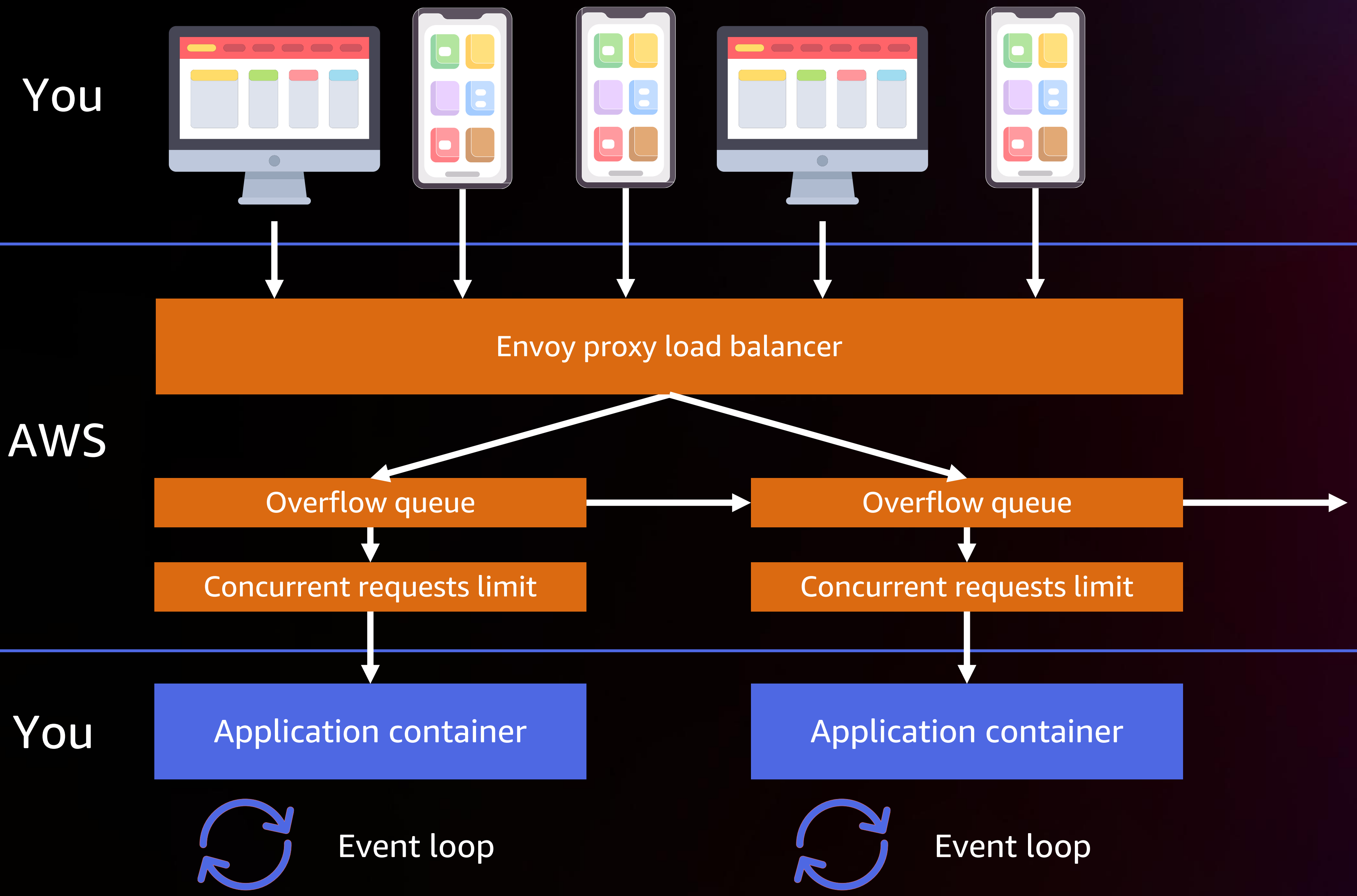


Your client-side code sends requests to the endpoint for an AWS App Runner service

The endpoint goes to a fully AWS managed Envoy proxy load balancer

Each instance of your application container has its own concurrency limit and overflow queue

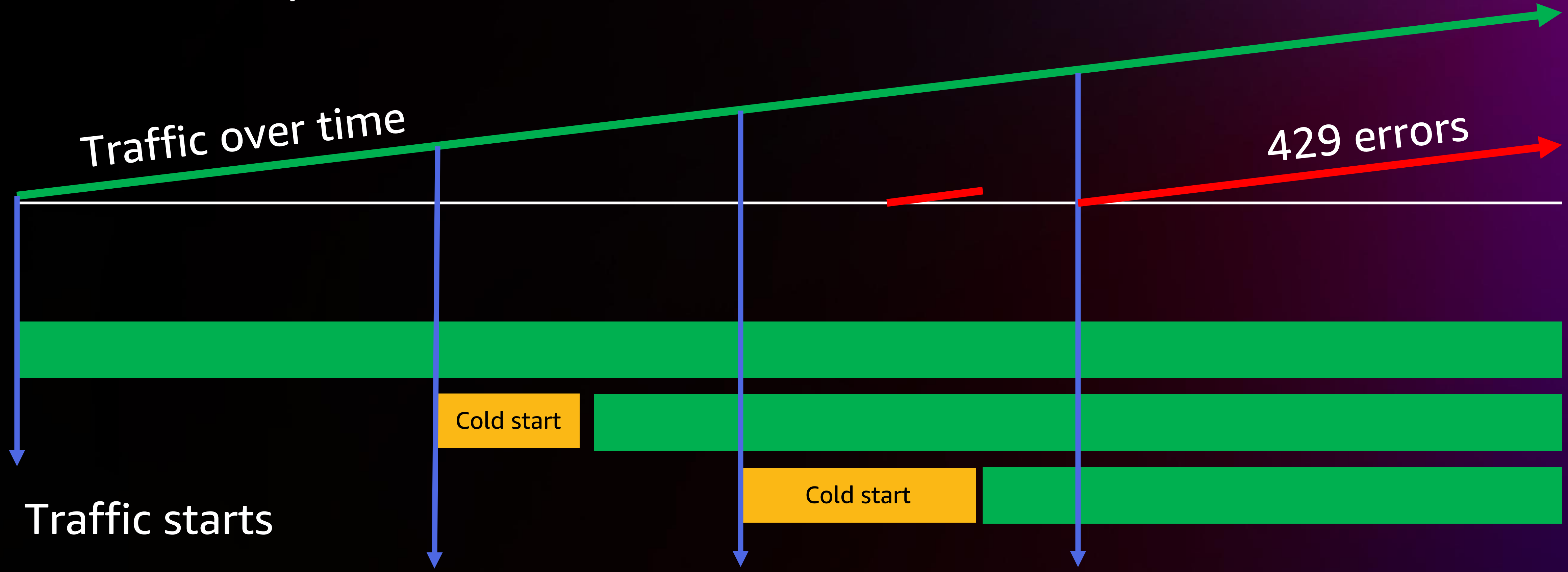
Your application receives concurrent requests, up to the limit you defined, and processes them concurrently



429 Too Many Requests error

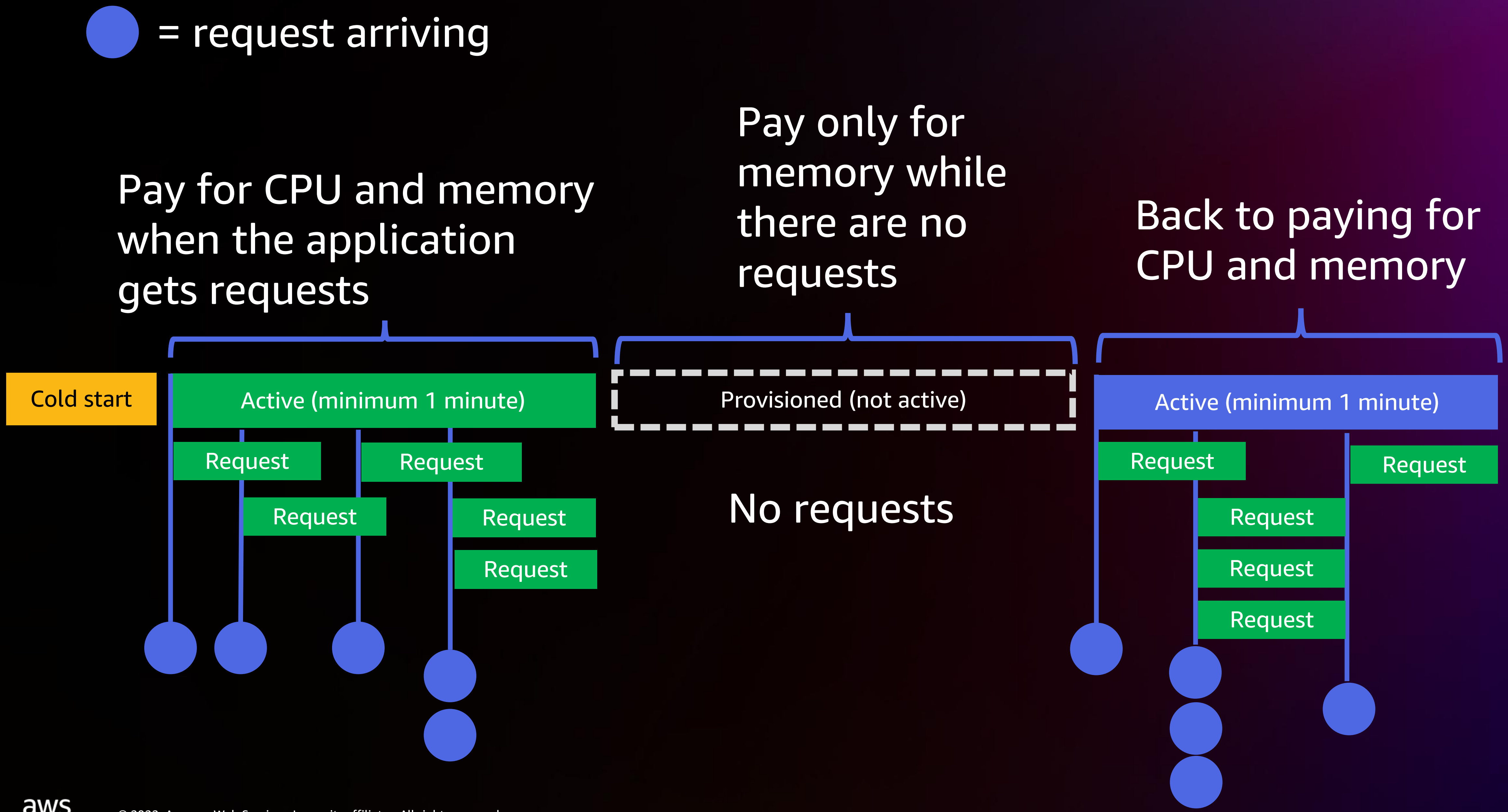
Your application is protected from traffic bursts

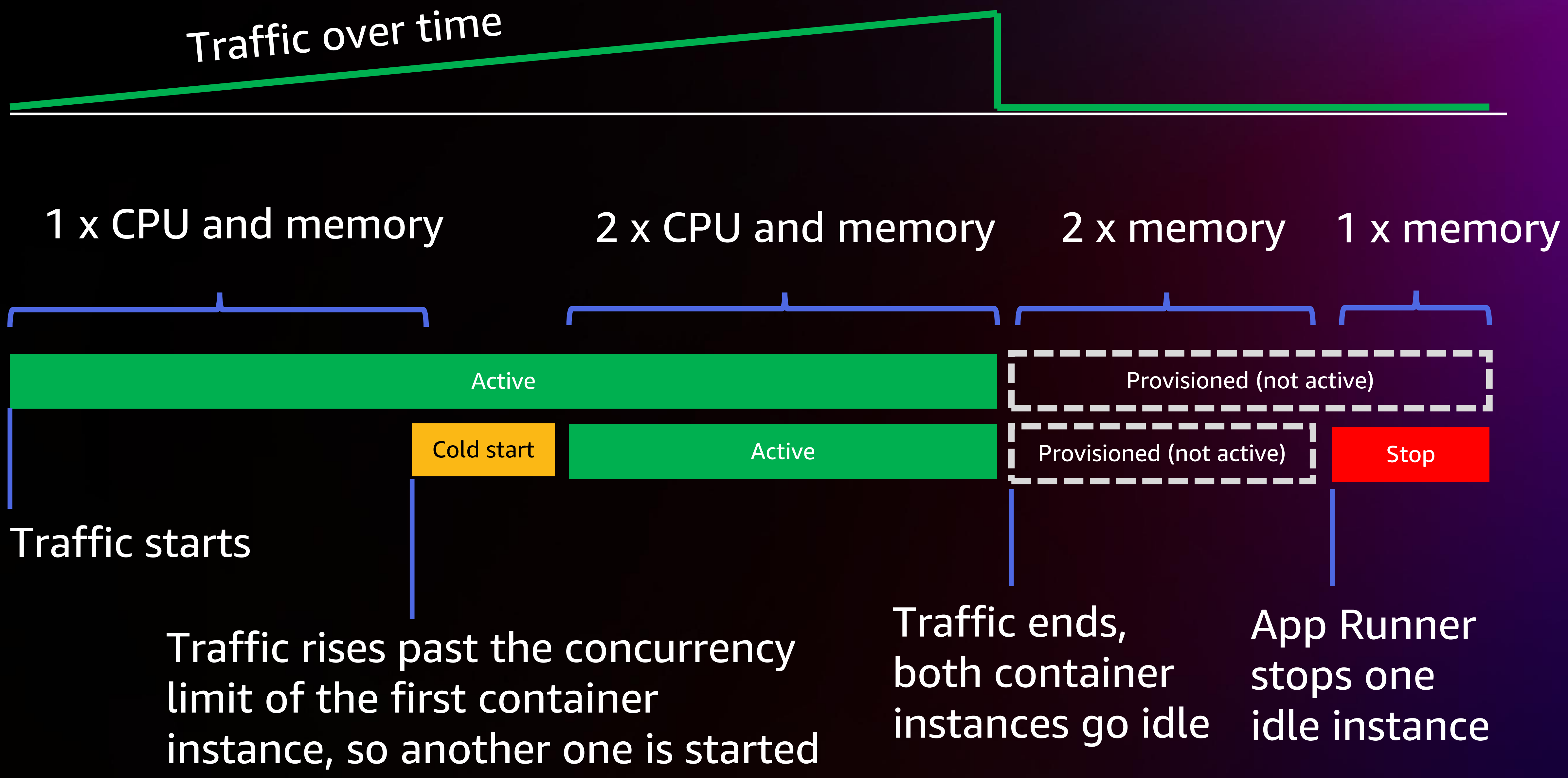
In this example, max container instances = 3



App Runner scales out preemptively, but make sure your app starts up fast or traffic might exceed available concurrency capacity

Extra traffic that can't be served by 3 container instances starts to get a 429 status code


= request arriving
Pay for CPU and memory when the application gets requests
Pay only for memory while there are no requests
Back to paying for CPU and memory
Cold start
Active (minimum 1 minute)
Request
Request
Request
Request
Request
Provisioned (not active)
No requests
Active (minimum 1 minute)
Request
Request
Request
Request
Request



Traffic over time
1 x CPU and memory
2 x CPU and memory
2 x memory
1 x memory
Active
Provisioned (not active)
Cold start
Active
Provisioned (not active)
Stop
Traffic starts
Traffic rises past the concurrency limit of the first container instance, so another one is started
Traffic ends, both container instances go idle
App Runner stops one idle instance

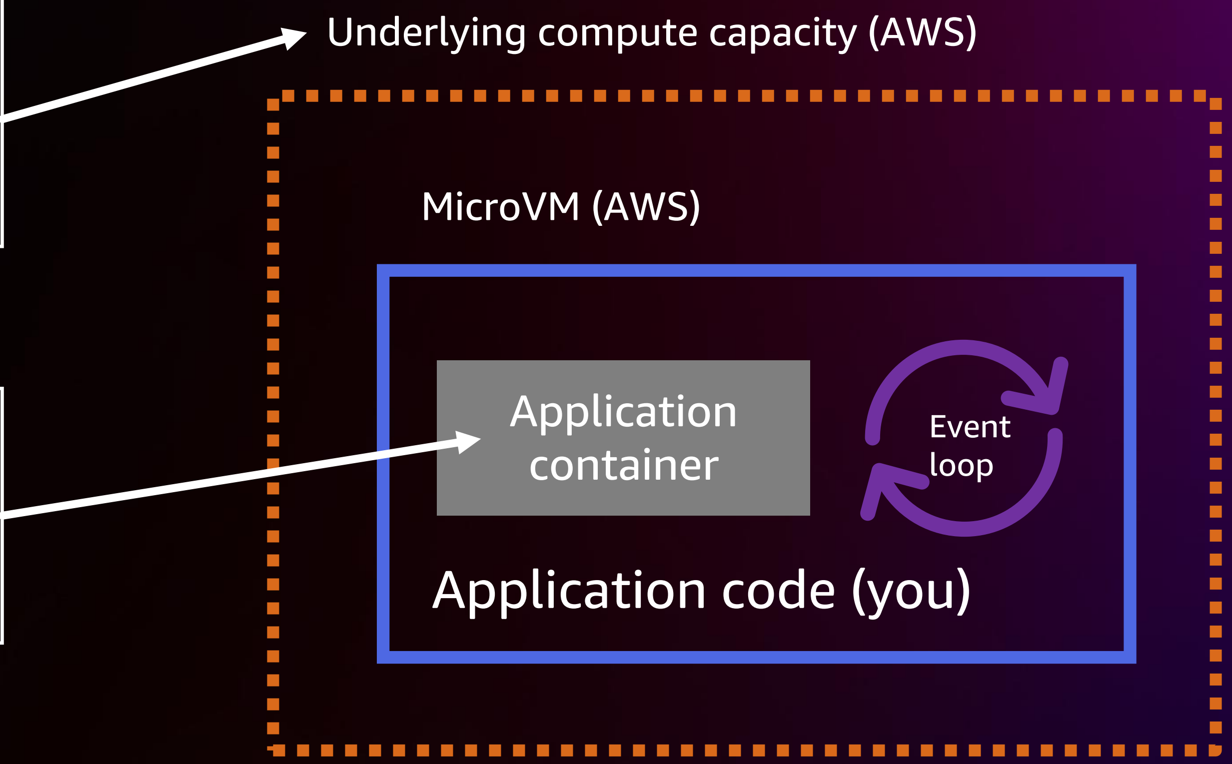
AWS Fargate

Serverless containers, but you manage scaling and concurrency

AWS Fargate provides microVMs on demand and patches and maintains them

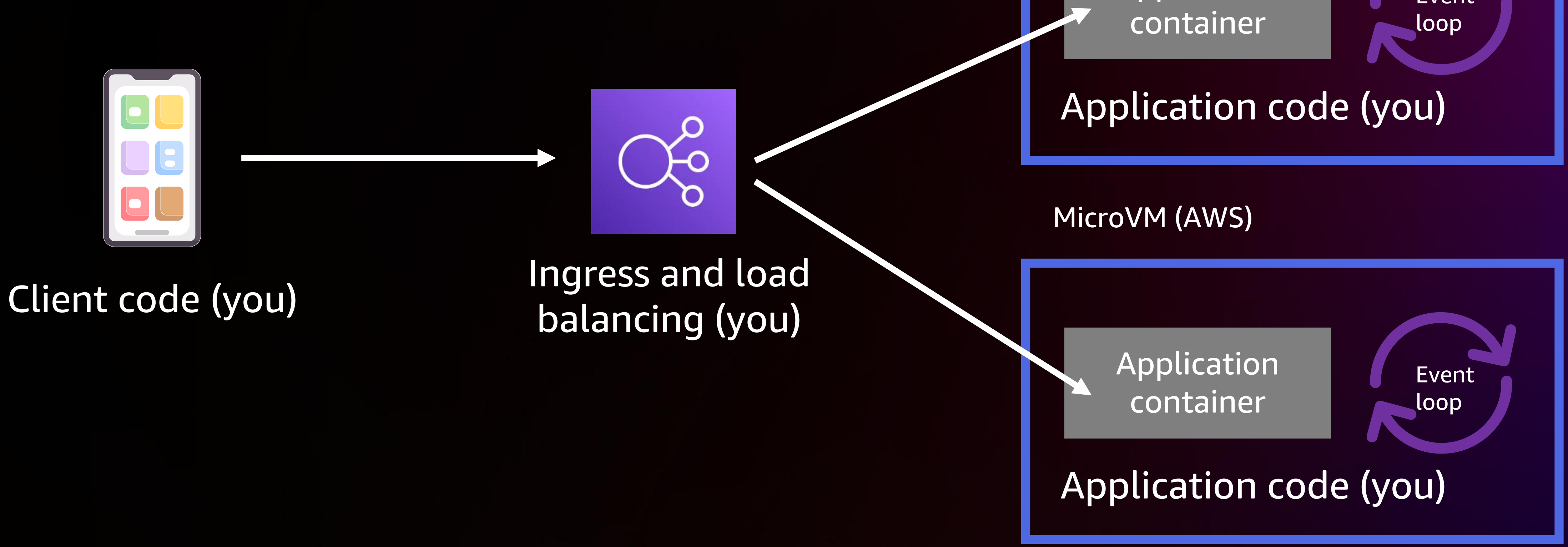
Patches to underlying hosts and infrastructure are managed by AWS

Patches to code, runtime, or libraries in your container are managed by you



You manage load balancing and ingress

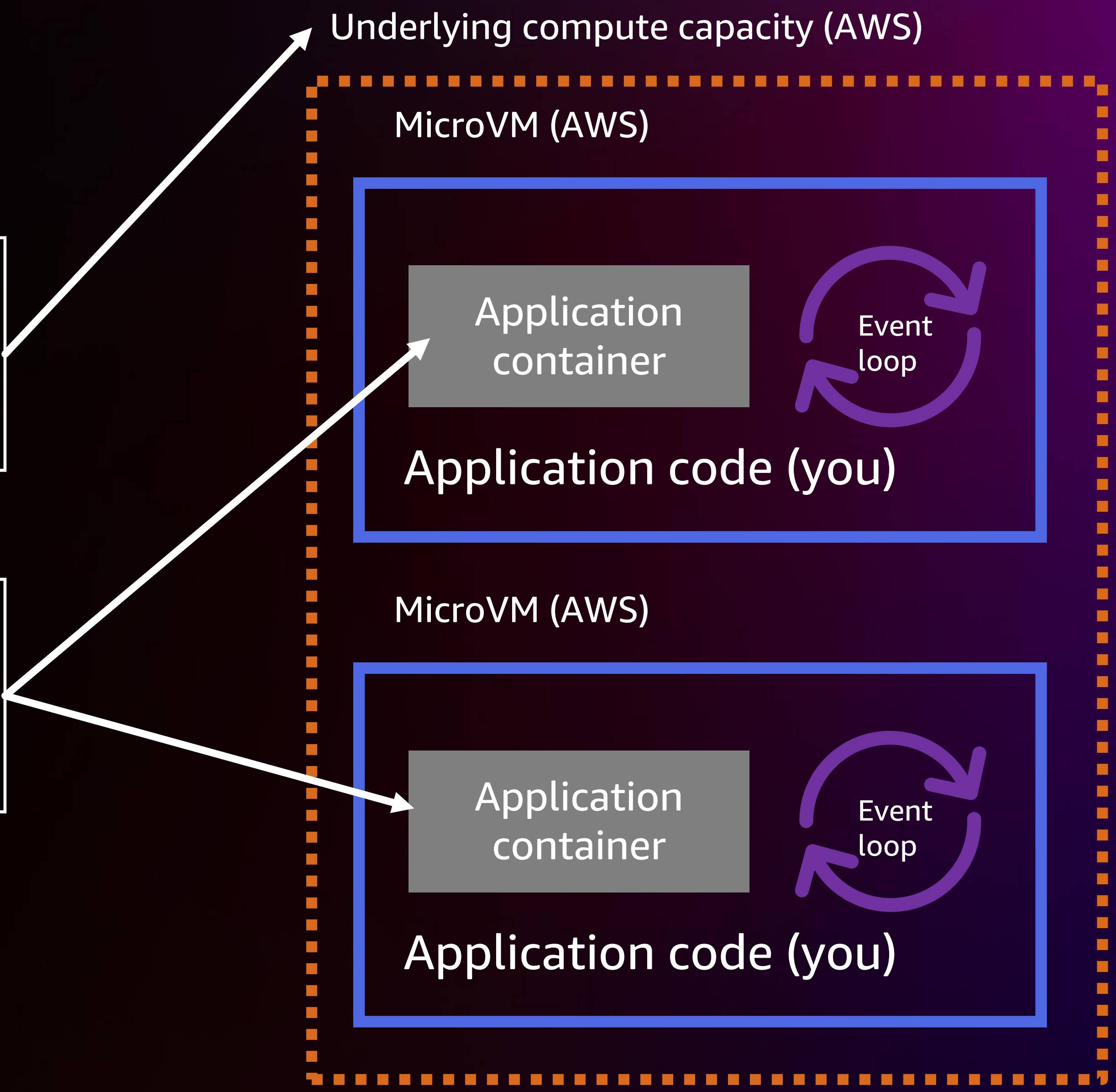
AWS = Fully managed by AWS
You = Your responsibility to manage



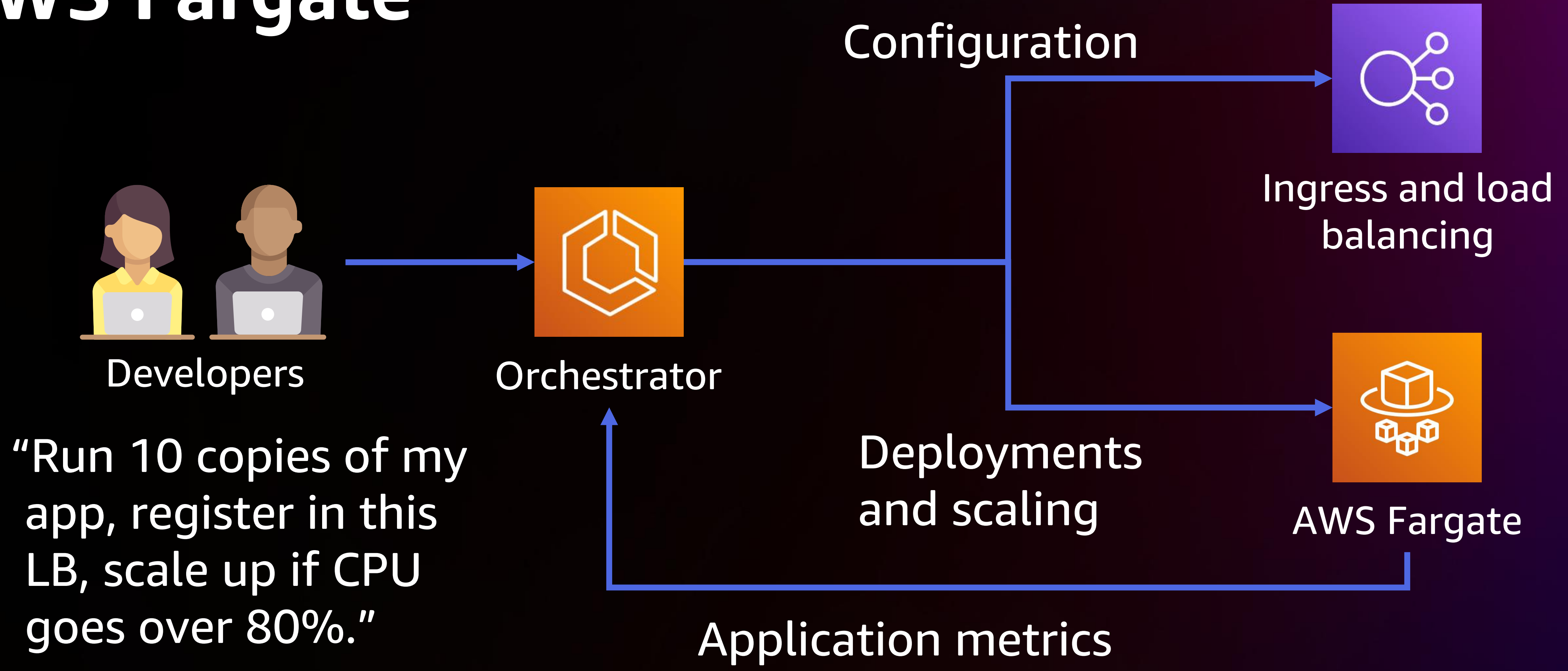
You configure scaling

Underlying compute capacity is provided by AWS to make sure you can launch containers on demand

You have to decide how many containers you need and what triggers scaling for those containers



You can use an orchestrator to configure and manage your own customized deployment on AWS Fargate



AWS Fargate gives you more choices

API

	Amazon ECS	Fully AWS managed serverless API; only pay for the Fargate tasks
	Amazon EKS	AWS managed open-source deployment; pay for your personal deployment of the control plane, plus AWS Fargate tasks

AWS Fargate gives you more choices

Ingress and load balancing

Application Load Balancer
Level 7, HTTP(S)

Network Load Balancer
Level 4

Amazon API Gateway

AWS Fargate gives you more choices

Scaling

AWS Auto Scaling

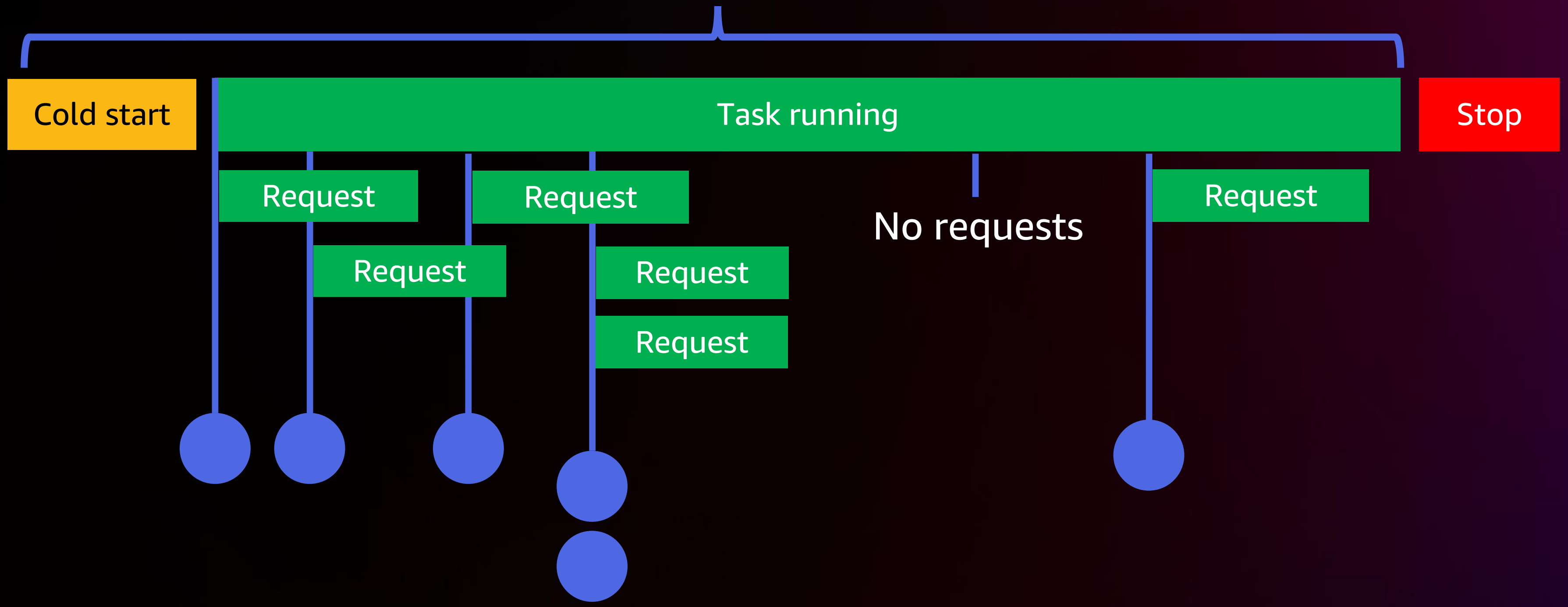
Step scaling
Target tracking
Scheduled scaling

Amazon CloudWatch

Scale based on concurrency, CPU utilization, available messages in an Amazon SQS queue, or any other custom metric

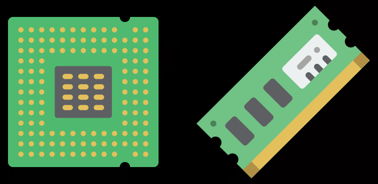
AWS Fargate charges based on time, not activity

Pay a constant rate per second for the task CPU and memory



Lowest per-request cost when concurrent traffic is high and code is efficient

AWS Fargate price saving strategies

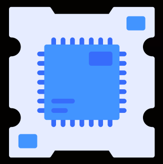


Fine-tune CPU and memory size to match application needs; Fargate offers tiny task sizes for small applications

Savings Plans

Fargate is part of AWS Compute Savings Plans

Spot tasks



Graviton2

Fargate has Spot tasks for a discounted price as well as AWS Graviton2–powered tasks with Arm architecture

So which serverless option is right for my application?

	AWS Lambda	AWS App Runner	AWS Fargate
Pricing	Per millisecond, per invoke, based on memory size of function Each concurrent invoke is charged separately	Per container instance per second, based on CPU and memory Container instances only charge for memory when not serving traffic	Per container instance per second, based on CPU and memory Constant price whether concurrent traffic is zero or hundreds of requests per second

	AWS Lambda	AWS App Runner	AWS Fargate
Pricing resolution	Each request is priced per millisecond, rounded up to the nearest millisecond Scale to zero between active requests	When a container instance activates, there is a minimum active time of 1 minute Active duration rounded up to the nearest second	Pricing is calculated per second with a 1-minute minimum Task duration rounded up to the nearest second

	AWS Lambda	AWS App Runner	AWS Fargate
Scaling	Each function instance only serves a single invoke at a time Lambda increases the number of function instances as needed	Each container instance serves many concurrent requests up to your limit App Runner manages the number of container instances automatically	High concurrency, but no built-in scaling You define your own custom scaling policy using an orchestrator, your metrics, and AWS Auto Scaling

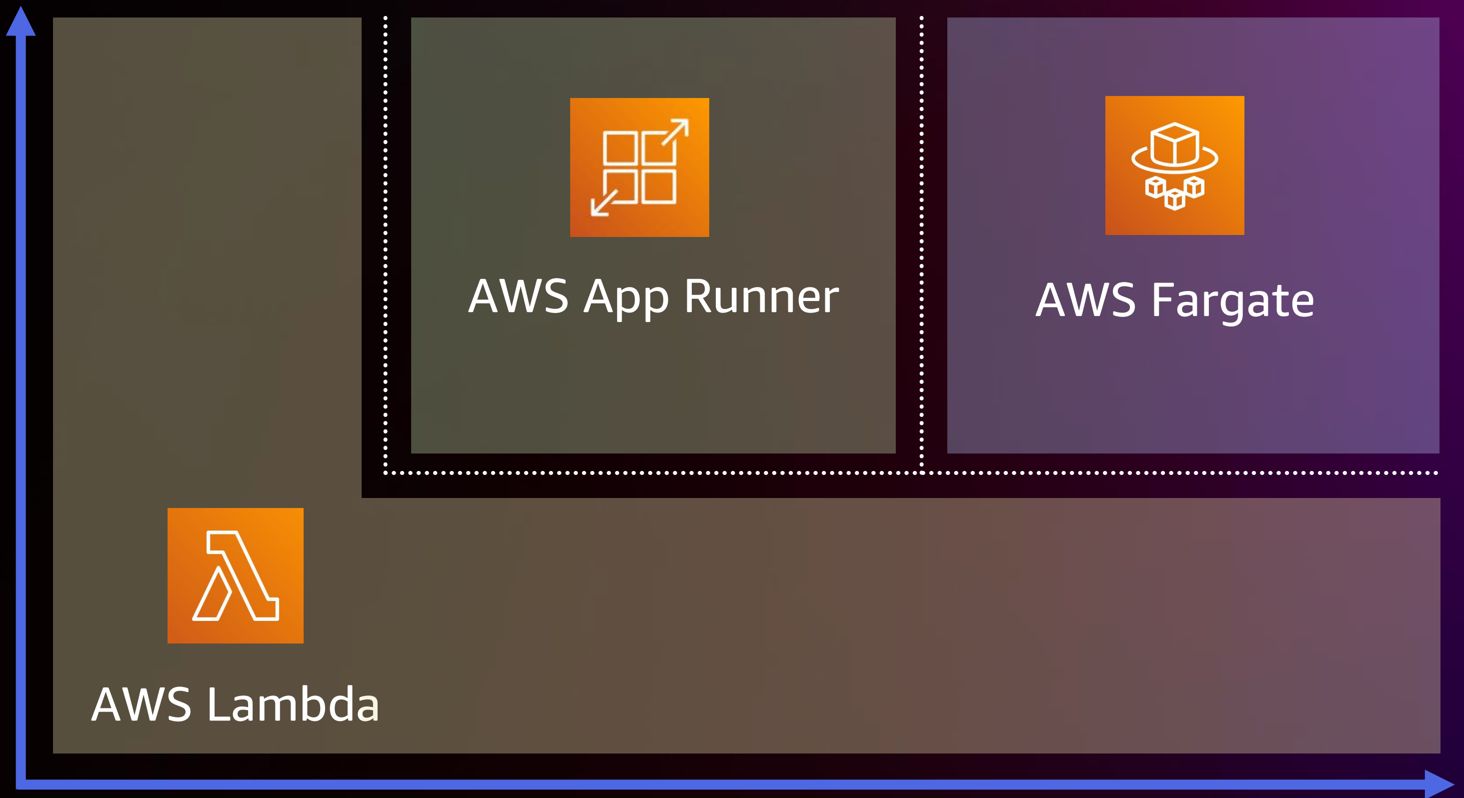
Serverless compute sweet spots for your container

Hundreds or thousands of requests at any time

Concurrency intensity

How much concurrency?

A few requests at a time



Very spiky, or it has "off hours"

Concurrency stability

How predictable is your concurrency?

Very stable or predictable

	AWS Lambda	AWS App Runner	AWS Fargate
Web traffic and ingress	Your choice Manage your own API Gateway or Application Load Balancer or use a built-in Lambda function URL	Built in Just call the endpoint for your App Runner service, no need to manage anything	Not included Manage your own API Gateway or Application Load Balancer

	AWS Lambda	AWS App Runner	AWS Fargate
Per-process isolation	✓	✓	✓
Per-request isolation	✓	✗	✗

A range of serverless compute for different needs

AWS Fargate

AWS App Runner

AWS Lambda

Configurable Control More complex	Opinionated Fully managed Simple

Thank you!

Nathan Peck
@nathankpeck

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