



# イチから理解するサーバーレスアプリ開発 エラー制御の勘所とモニタリング



@\_kensh

# Who am I?

## Name

Kensuke Shimokawa

## Company

Amazon Web Services Japan K.K.

## Role

Serverless Specialist Solutions Architect



@\_kensh

# 本セッションについて

Serverlessな構成、AWS Lambdaの導入でサーバー管理はなくなるが、運用業務は発生します。

本セッションでは、サーバーレス のエラー制御・モニタリングを説明する

本セッションの対象

- AWS Lambda
  - Amazon API Gateway
- etc.

お話ししないこと

- Lambda@Edge のお話
- AWS Fargate/Amazon Aurora Serverlessのお話

# 本日のアジェンダ

- AWS Lambdaのおさらい
- AWS Lambdaでのプログラミング
- 仕組みを理解しよう
- エラー制御について
- AWS Lambdaをモニタリング



# 本日のアジェンダ

- AWS Lambdaのおさらい
- AWS Lambdaでのプログラミング
- 仕組みを理解しよう
- エラー制御について
- AWS Lambdaをモニタリング

# AWS Lambdaについて（おさらい）



インフラのプロビジョニング不要  
管理不要



自動でスケール

価値に対する支払い



高可用かつ安全



# サポートされているランタイム言語

- Python 2.7, 3.6, 3.7, 3.8
- Node.js 10.x, 12.x
- .NET Core 2.1, 3.1(C# / PowerShell)
- Go 1.x
- Java 8, Java11
- Ruby 2.5, 2.7

サポートされていない言語は  
Custom Runtimeを実装することで  
利用可能

# AWS Lambdaのハンドラー

- 使用する言語の関数もしくはメソッドを指定し、実行時に呼び出すエントリーポイントとする
- ハンドラーが呼び出されることで、Lambda関数のコードが実行開始される
- 呼び出しパラメータとして渡されるイベントデータを参照可能

```
import json
```

Python3.7

```
def lambda_handler(event, context):  
    # 任意のコード  
    return {  
        'statusCode': 200,  
        'body': json.dumps('Hello')  
    }
```

# Management Console上の表示

## 関数コード 情報

```
12  
13  
14 def lambda_handler(event, contex  
15  
16     print("helloworld 2020")  
17     print("Request ID:", context.  
18     print(json.dumps(event))  
19  
20     #time.sleep(30)  
21
```

Lambda > 関数 > helloworld > 基本設定を編集

## 基本設定を編集

### 基本設定 情報

説明 - オプション

ランタイム

Python 3.7

ハンドラ 情報

lambda\_function.lambda\_handler

メモリ (MB)

作成する関数には、設定したメモリに比例する CPU が割り当てられます。

128 MB

タイムアウト

15

分

0

秒

ランタイム言語

ファイル名.関数名

# 他サービスとの連携

設定 | アクセス権限 | モニタリング

▼ デザイナー

Lambdaを呼び出すサービス

helloworld  
Layers (1)

EventBridge (CloudWatch Events)  
API Gateway (15)  
DynamoDB

+ トリガーを追加

Amazon SQS

+ 送信先を追加

Lambdaから呼び出されるサービス

# 他サービスとの連携

設定

アクセス権限

モニタリング

## Resource summary

[View role document](#)



Amazon DynamoDB  
1 actions, 1 resources



AWS Key Management Service  
2 actions, 1 resources



AWS Lambda  
7 actions, 1 resources



AWS Resource Groups  
6 actions, 1 resources



Amazon CloudWatch  
8 actions, 2 resources



Amazon CloudWatch Logs  
1 actions, 1 resources



Amazon DynamoDB  
1 actions, 1 resources

Lambdaが呼び出し可能なサービス

# Amazon API Gateway から AWS Lambda を呼び出した場合

```
{  
  "resource": "/helloworld",  
  "path": "/helloworld",  
  "httpMethod": "GET",  
  "headers": {  
    "Content-Type": "text/html",  
    "Host": "eefsdgfr1.execute-api.ap-northeast-1.amazonaws.com",  
    "User-Agent": "hey/0.0.1",  
    "X-Forwarded-For": "11.23.222.132",  
    "X-Forwarded-Port": "443",  
    "X-Forwarded-Proto": "https"  
  },  
  "queryStringParameters": null,  
  "body": null  
}
```

Eventデータ（抜粋）



# 本日のアジェンダ

- AWS Lambdaのおさらい
- **AWS Lambdaでのプログラミング**
- 仕組みを理解しよう
- エラー制御について
- AWS Lambdaをモニタリング

基本的には言語のベストプラクティスに従う



python™

OpenJDK



# AWS Lambdaのプログラミングの基本ルール

## • ステートレス

- 関数で保持したデータは揮発するので、保持したいデータ（ステート）は別のところで管理（Amazon DynamoDB, Amazon S3等）

## • べき等

- At least onceの原則に従うと、関数は複数回起動することがある
    - 例) Amazon S3から非同期に呼び出される場合のリトライ対応
- ※詳細は後ほど



<https://youtu.be/QvPgjEwgiew>

# [AWS Black Belt Online Seminar] Let's Dive Deep into AWS Lambda

Solutions Architect 西谷 圭介  
2019/04/02

AWS 公式 Webinar  
<https://amzn.to/JPWebinar>



過去資料  
<https://amzn.to/JPArchive>



# AWS LambdaでのLog出力の方法

- 基本的には、言語の標準出力機能をそのまま使えます。



```
print ("hello, world")
```



```
Console.WriteLine("hello, world");
```



```
System.out.println("hello, world");
```



```
console.log("hello, world");
```

# AWS Lambdaでのログ用権限設定

設定

アクセス権限

モニタリング

## Resource summary

[View role document](#)



Amazon CloudWatch Logs  
3 actions, 2 resources

Lambdaが呼び出し可能なサービス

To view the resources and actions that your function has permission to access, choose a service.

アクション別

リソース別

アクション

リソース

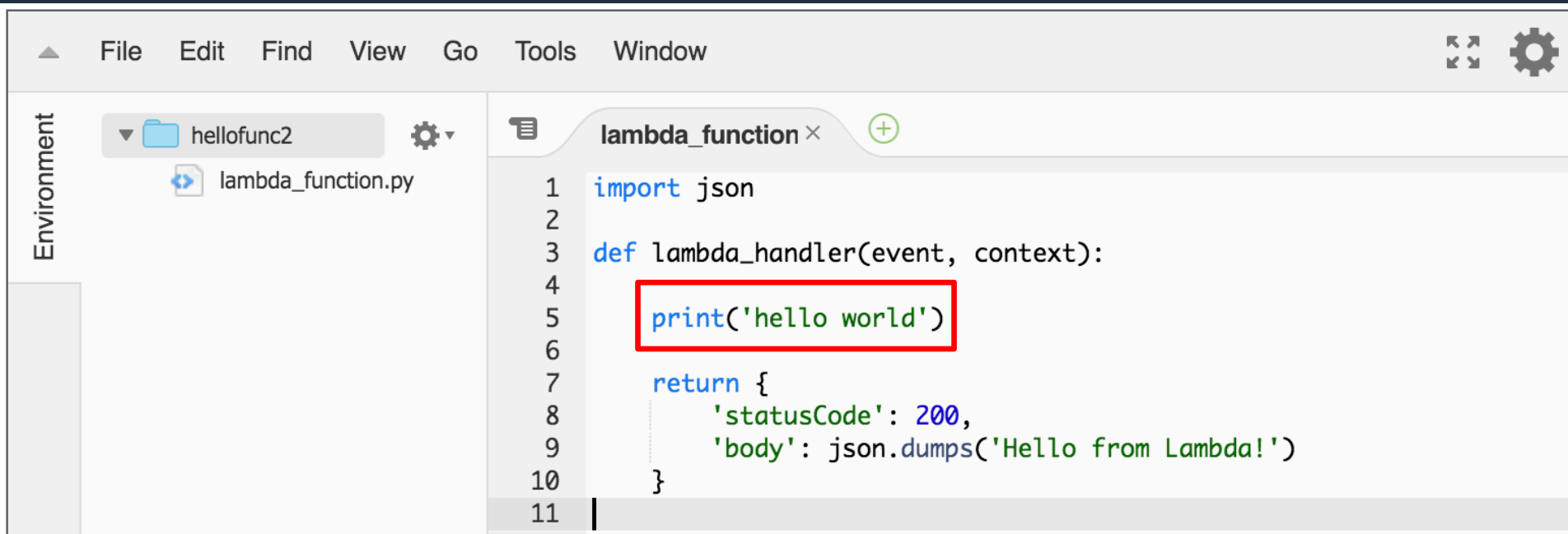
logs:CreateLogGroup

Allow: arn:aws:logs:ap-northeast-1:494319857181:\*

logs:CreateLogStream

Allow: arn:aws:logs:ap-northeast-1:494319857181:log-group:/aws/lambda/binary:\*

# AWS Lambdaで標準出力



The image shows a code editor window with a menu bar (File, Edit, Find, View, Go, Tools, Window) and a toolbar. The left sidebar shows the 'Environment' with a folder named 'hellofunc2' and a file named 'lambda\_function.py'. The main editor area shows the code for 'lambda\_function.py' with the following content:

```
1 import json
2
3 def lambda_handler(event, context):
4     print('hello world')
5
6     return {
7         'statusCode': 200,
8         'body': json.dumps('Hello from Lambda!')
9     }
10
11
```

The line `print('hello world')` on line 4 is highlighted with a red rectangular box.

# Amazon CloudWatch Logsでの表示



CloudWatch  
ダッシュボード  
アラーム  
アラーム  
不足  
OK  
請求  
イベント  
ルール  
イベントバス  
ログ  
インサイト  
メトリクス  
Alpine  
設定  
お気に入り  
+ ダッシュボードを追加

CloudWatch > ロググループ > /aws/lambda/hellofunc2 >  
2019/09/05/[\$LATEST]6de7b98db9664815b631899edac6885a

すべて展開  行  テキスト

イベントのフィルター  
メッセージ  
すべて 2019-09-04 (02:57:22)

2019-09-05 02:57:22

START RequestId: a0c9ec80-9b8b-409e-8b92-4a4c006c15c1 Version: \$LATEST

hello world

END RequestId: a0c9ec80-9b8b-409e-8b92-4a4c006c15c1

REPORT RequestId: a0c9ec80-9b8b-409e-8b92-4a4c006c15c1 Duration: 6.17 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 114.87 ms  
XRAY TraceId: 1-5d707991-3ca4492eb1c2caf48d140ef0 SegmentId: 594344027702a828 Sampled: false

いまのところ古いイベントはありません。再試行。

いまのところ新しいイベントはありません。再試行。

標準出力

レポート



# AWS LambdaでのLog出力の方法

- ただし、ログ運用のためにはLoggerなどログ出力ライブラリを用いる



Logの検索性を高めることで  
後のLog解析に役に立つ

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
def lambda_handler(event, context):
    logger.debug('debug')
    logger.info('info')
    logger.warning('warn')
    logger.error('error')
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

# Loggerの出力例

- レベルタイプ
- タイムスタンプ
- リクエストID
- 標準出力

イベントのフィルター

メッセージ

2019-09-05 03:09:26

いまのところ古いイベントはありません。 [再試行](#)

START RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b Version: \$LATEST

[INFO] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b info

[WARNING] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b warn

[ERROR] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b error

END RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b

REPORT RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b Duration: 5.65 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 106.60 ms

XRAY TraceId: 1-5d707c66-1b02255204a5aa3e993b8c44 SegmentId: 30ba849b7bb8f1c5 Sampled: false

# 本日のアジェンダ

- AWS Lambdaのおさらい
- AWS Lambdaでのプログラミング
- 仕組みを理解しよう
- エラー制御について
- AWS Lambdaをモニタリング

# AWS Lambdaの仕組み

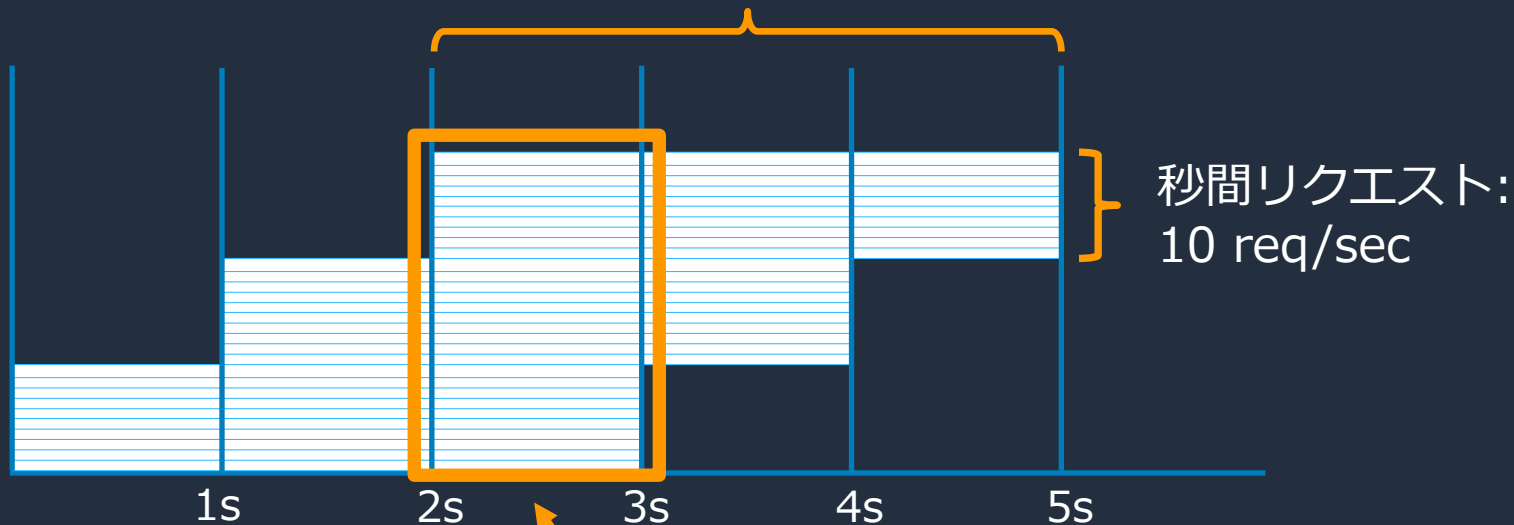
## □ RequestResponseベース



リクエスト数が伸びれば、Lambdaの同時実行数も伸びる

# 同時実行数とは

関数の平均実行時間: 3s / exec



同時実行数 = “同時”に実行されているタイミング  
= 3 × 10 = 30

# 本日のアジェンダ

- AWS Lambdaのおさらい
- AWS Lambdaでのプログラミング
- 仕組みを理解しよう
- エラー制御について
- AWS Lambdaをモニタリング

# エラー制御についてのパターン

モニタリング、トレース、アラート

サーバーレスのワークフローサービスを利用

AWS Lambdaの制御

プログラミング内の制御構造

# エラー制御についてのパターン

モニタリング、トレース、アラート

サーバーレスのワークフローサービスを利用

AWS Lambdaの制御

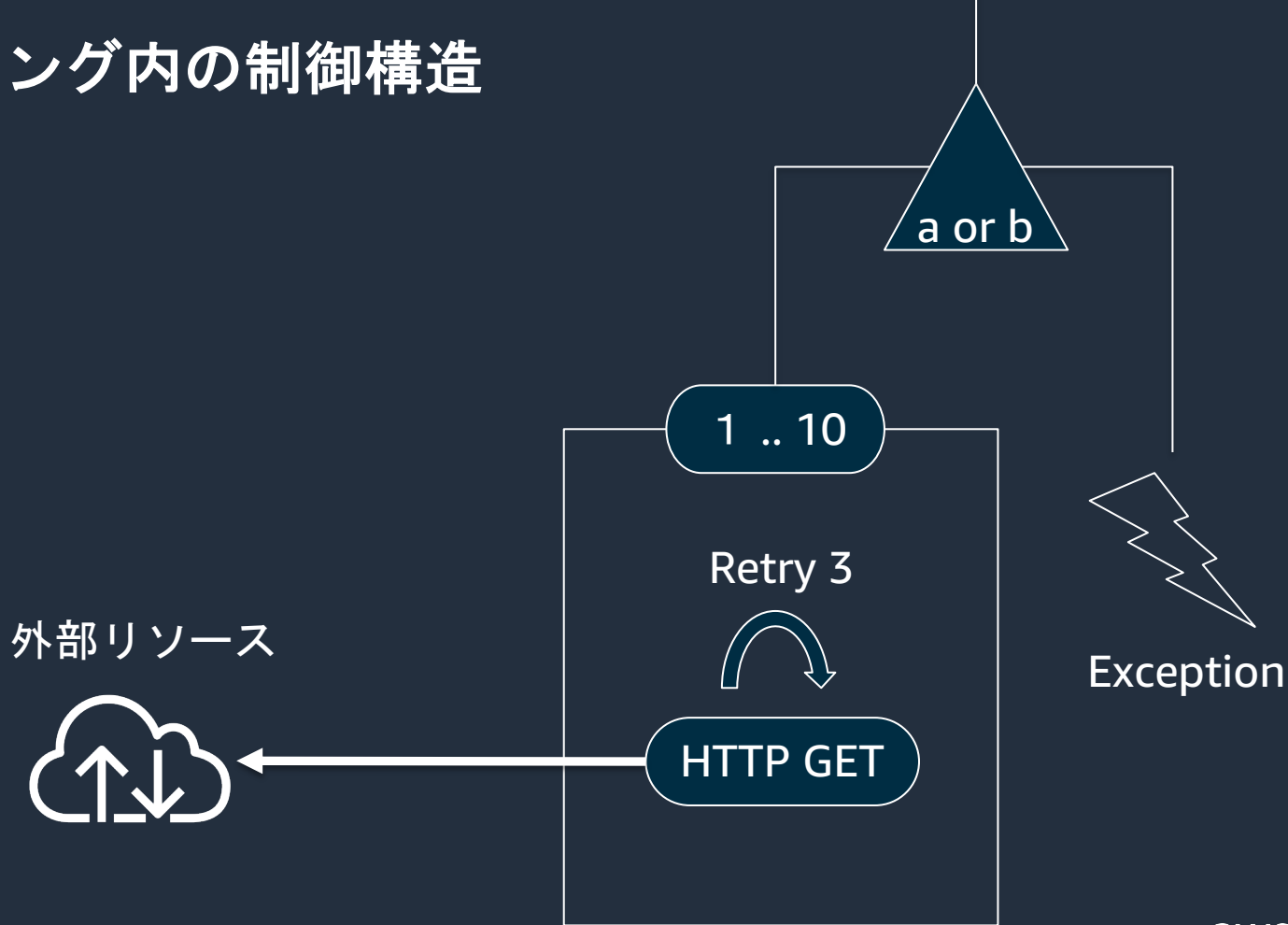
プログラミング内の制御構造





# プログラミング内の制御構造

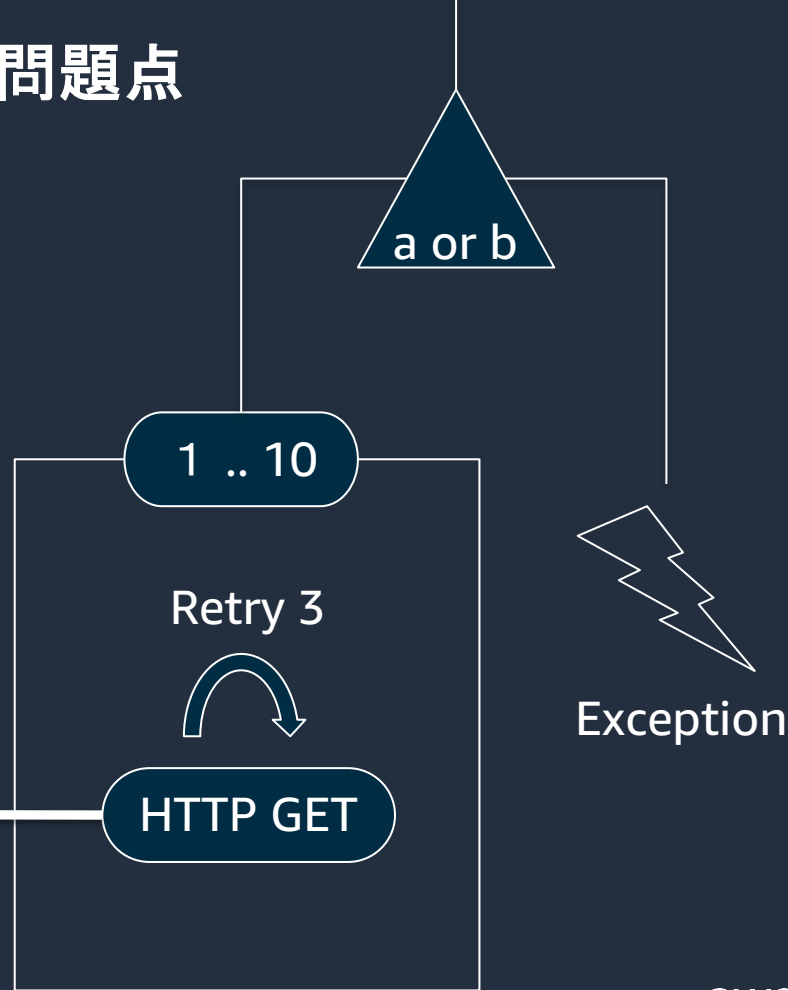
- 分岐
- 繰り返し
- 例外
- リトライ



# プログラミング内の制御構造の問題点

- コード量が多くなると、Lambdaの起動が遅くなる
- どのコードブロックでエラーが起きたか発見が困難

外部リソース



# エラー制御についてのパターン

モニタリング、トレース、アラート

サーバーレスのワークフローサービスを利用

AWS Lambdaの制御

プログラミング内の制御構造

# AWS Lambdaの非同期呼び出しについて

関数を間接的に呼び出す際、呼び出し元やリクエストが遭遇するすべてのサービスの再試行動作について把握する必要があります。これには、以下のシナリオが含まれます。

- **非同期呼び出し** – Lambda は、関数エラーを **2回再試行** します。関数にすべての受信リクエストを処理する十分なキャパシティがない場合、関数に送信されるまで、イベントはキューの中に数時間または数日間保持される可能性があります。正常に処理できなかったイベントを把握するために、デッドレターキューを設定できます。詳細については、「**非同期呼び出し**」を参照してください。

[https://docs.aws.amazon.com/ja\\_jp/lambda/latest/dg/retries-on-errors.html](https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/retries-on-errors.html)

# Amazon Simple Storage Service (S3)での非同期呼び出し例

- S3バケットにファイルをアップロード
- アップロードでイベント発生
- Lambda関数を呼び出すが、合計3回失敗(リトライ2回)
- DLQ (Dead Letter Queue)にイベントが送信される
- DLQは後ほど、バッチで処理するなど対応



# AWS Lambdaの制御

- 非同期実行のAWS Lambdaは関数失敗時に、最大2回リトライがされる。（初回含め3回）
- DLQ (Dead Letter Queue)は、そのリトライを尽くしてもエラーとなった場合に、未処理イベントが送信される先

Amazon SNSまたは  
Amazon SQSが指定可能

## デバッグとエラー処理

### DLQ リソース [情報](#)

最大再試行回数を超えた後にイベントペイロードを送信する AWS のサービスを選択します。

Amazon SQS ▼

### SQS Queue

DLQ として機能する、既存のキューまたはトピックの名前です。

LambdaDLQ ▼

### AWS X-Ray [情報](#)

アクティブトレースを有効にして、呼び出しのサブセットタイミングとエラー情報を記録します。

アクティブトレース

# エラー制御についてのパターン

モニタリング、トレース、アラート

サーバーレスのワークフローサービスを利用

AWS Lambdaの制御

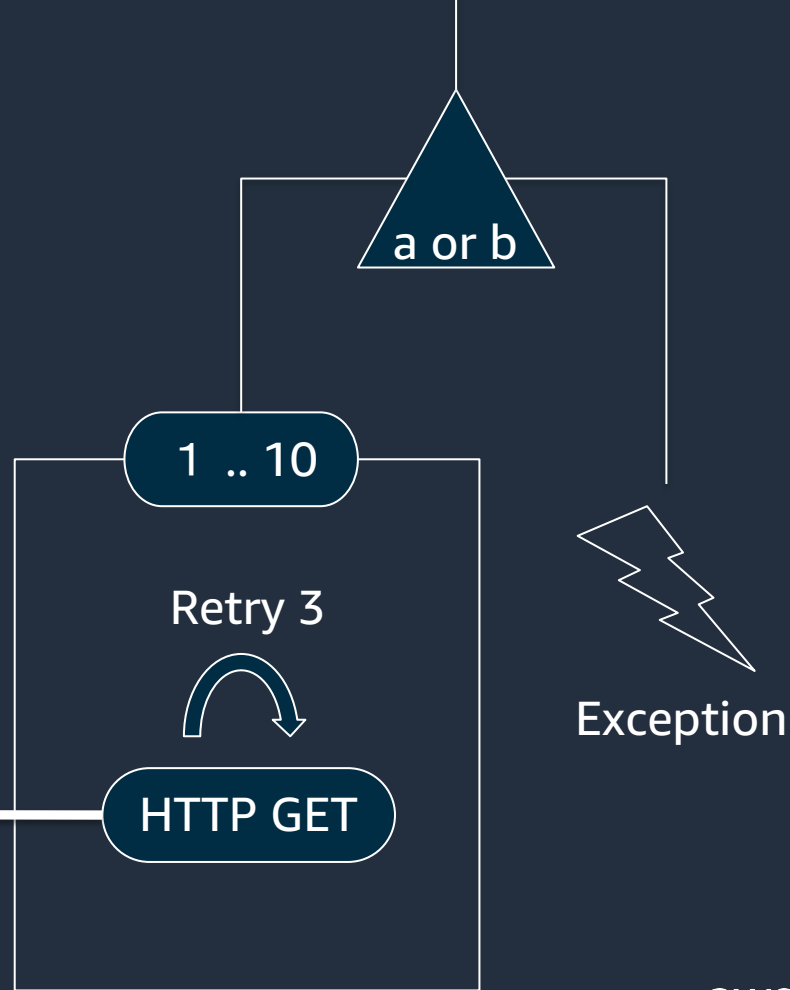
プログラミング内の制御構造

# Lambda関数内でロジック実装

- 分岐
- 繰り返し
- 例外
- リトライ



外部リソース



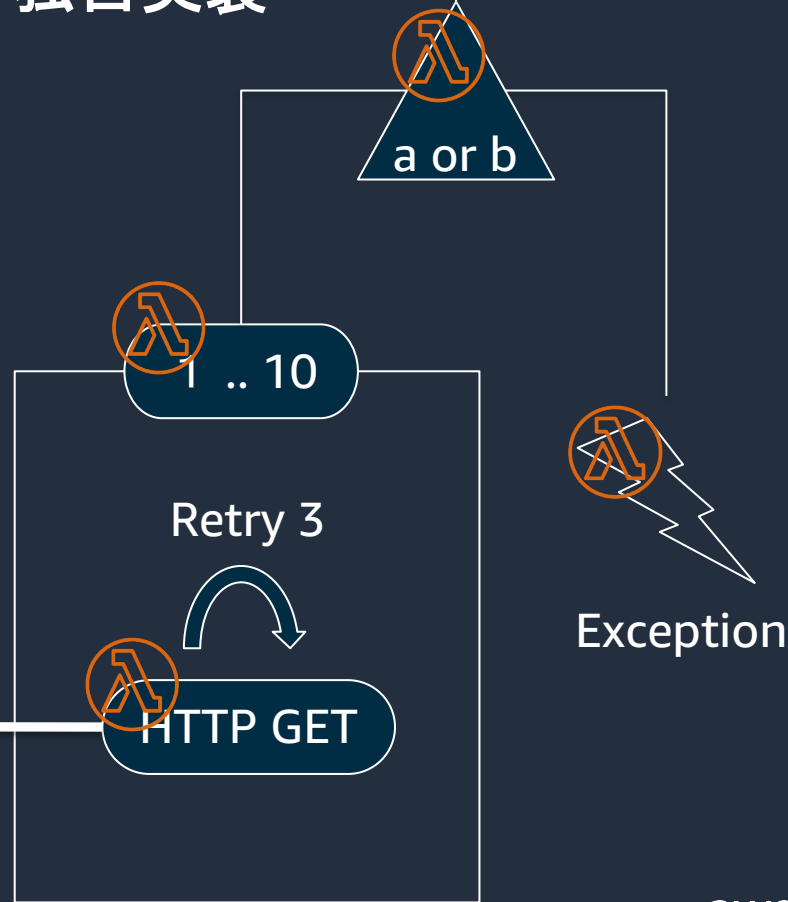


# Lambda関数内でワークフローを独自実装

- 分岐
- 繰り返し
- 例外
- リトライ

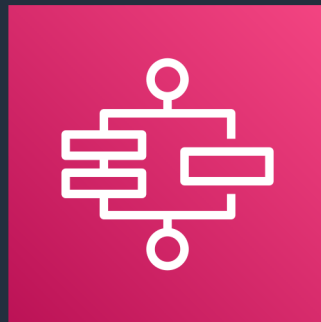


外部リソース

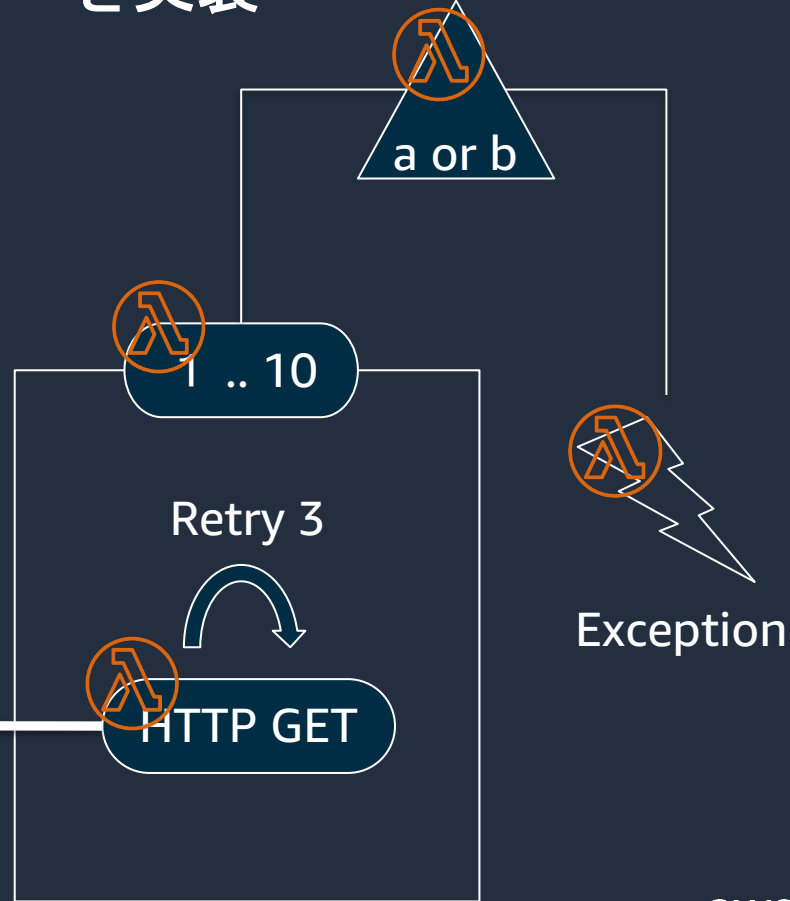


# AWS Step Functionsでワークフローを実装

- 分岐
- 繰り返し
- 例外
- リトライ



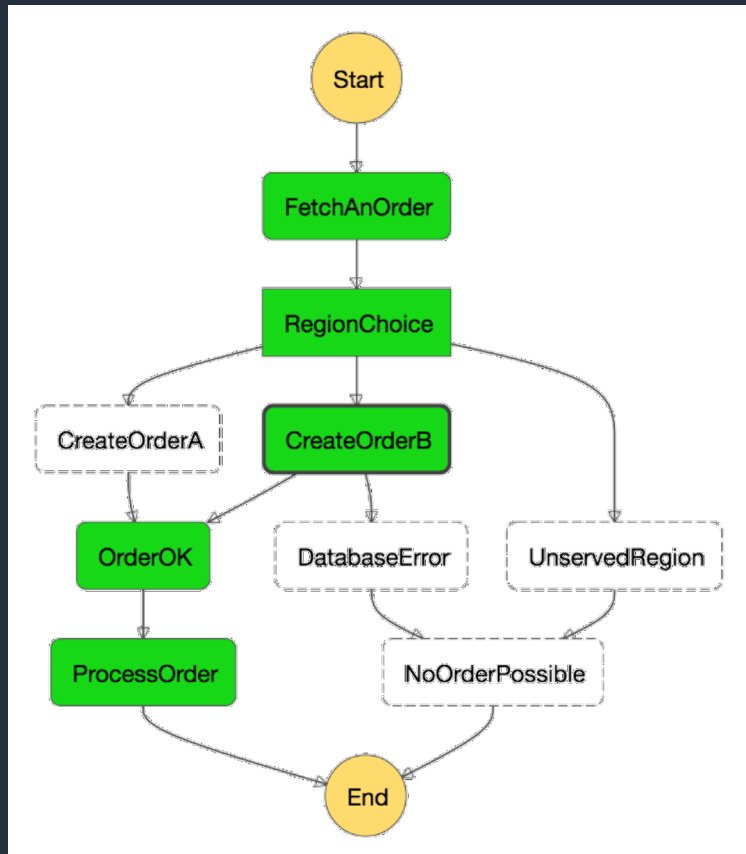
外部リソース



# AWS Step Functions



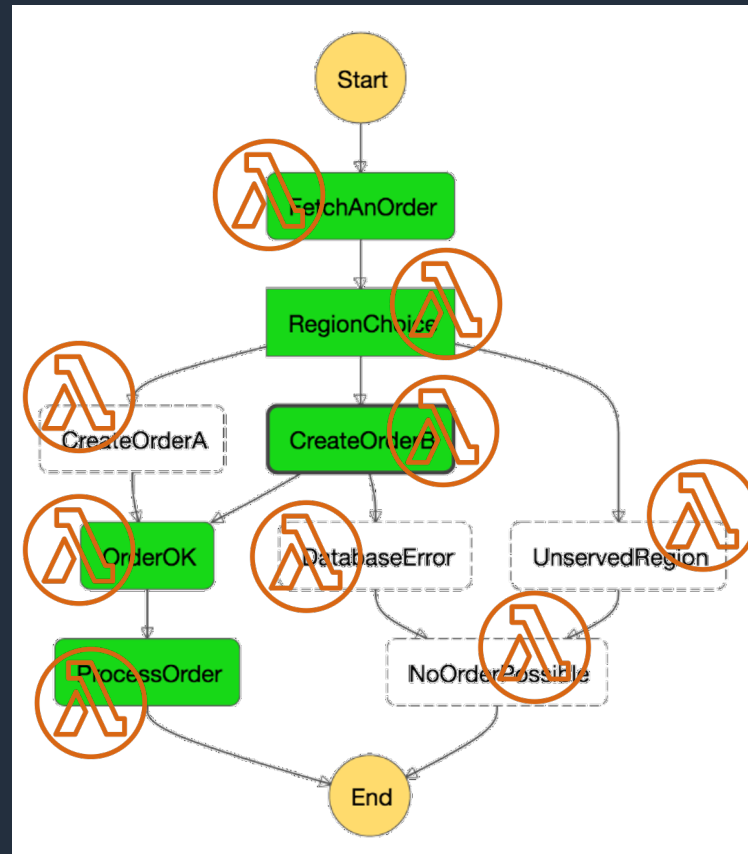
- 分散アプリケーション全体をオーケストレーションできる
- Management Console から「ワークフロー」(右図) という形式で見やすく可視化できる
- 各ステップの実行履歴をログから追跡できる



# AWS Step Functions



- 分散アプリケーション全体をオーケストレーションできる
- Management Console から「ワークフロー」(右図) という形式で見やすく可視化できる
- 各ステップの実行履歴をログから追跡できる



# ステートマシンから呼び出し可能な AWS のサービス

- AWS Lambda : Lambda 関数の実行
- Amazon Dynamo DB : 既存のアイテムの取得、新規アイテムの登録
- AWS Batch : ジョブの起動、ジョブ完了の待機
- Amazon ECS : ECS/Fargate タスクの実行
- Amazon SNS : SNS トピックへのメッセージ送信
- Amazon SQS : SQS キューへのメッセージ送信
- AWS Glue : Glue ジョブの実行
- Amazon SageMaker : トレーニングジョブ、トランスフォームジョブの起動

# エラー制御についてのパターン

モニタリング、トレース、アラート

サーバーレスのワークフローサービスを利用

AWS Lambdaの制御

プログラミング内の制御構造

# 本日のアジェンダ

- AWS Lambdaのおさらい
- AWS Lambdaでのプログラミング
- 仕組みを理解しよう
- エラー制御について
- **AWS Lambdaをモニタリング**

# モニタリングについて





## Inventory and classification



AWS Systems Manager



AWS Config



AWS CloudTrail

## Service request



AWS Service Catalog



AWS OpsWorks



AWS CloudFormation

## Provisioning and orchestration

## Cost management and resource optimization



AWS Trusted Advisor

## Cloud migration, backup, and DR



AWS Snowball



AWS SMS

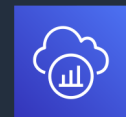


AWS DMS

## Packaging and delivery

## security and compliance

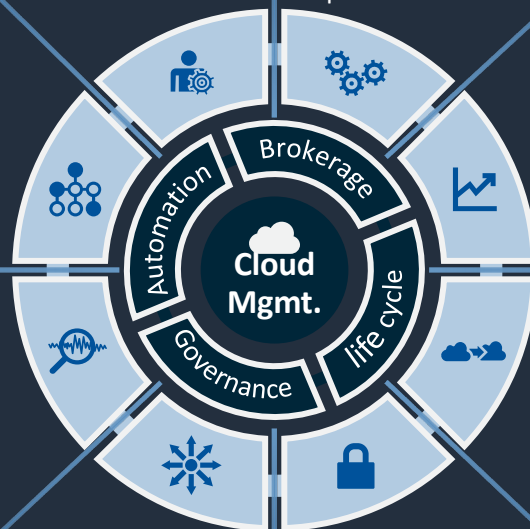
## Monitoring and analytics



AWS X-Ray



Amazon CloudWatch





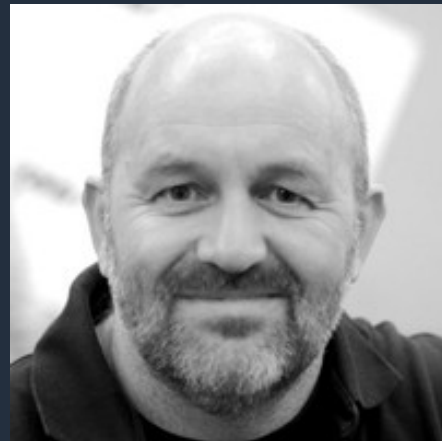
誰がモニタリングするのか

伝統的なシステムでは開発と運用の間に壁がある。  
そんな壁はぶん投げてしまえ。  
忘れてしまってもいい。Amazonでは不要だ。

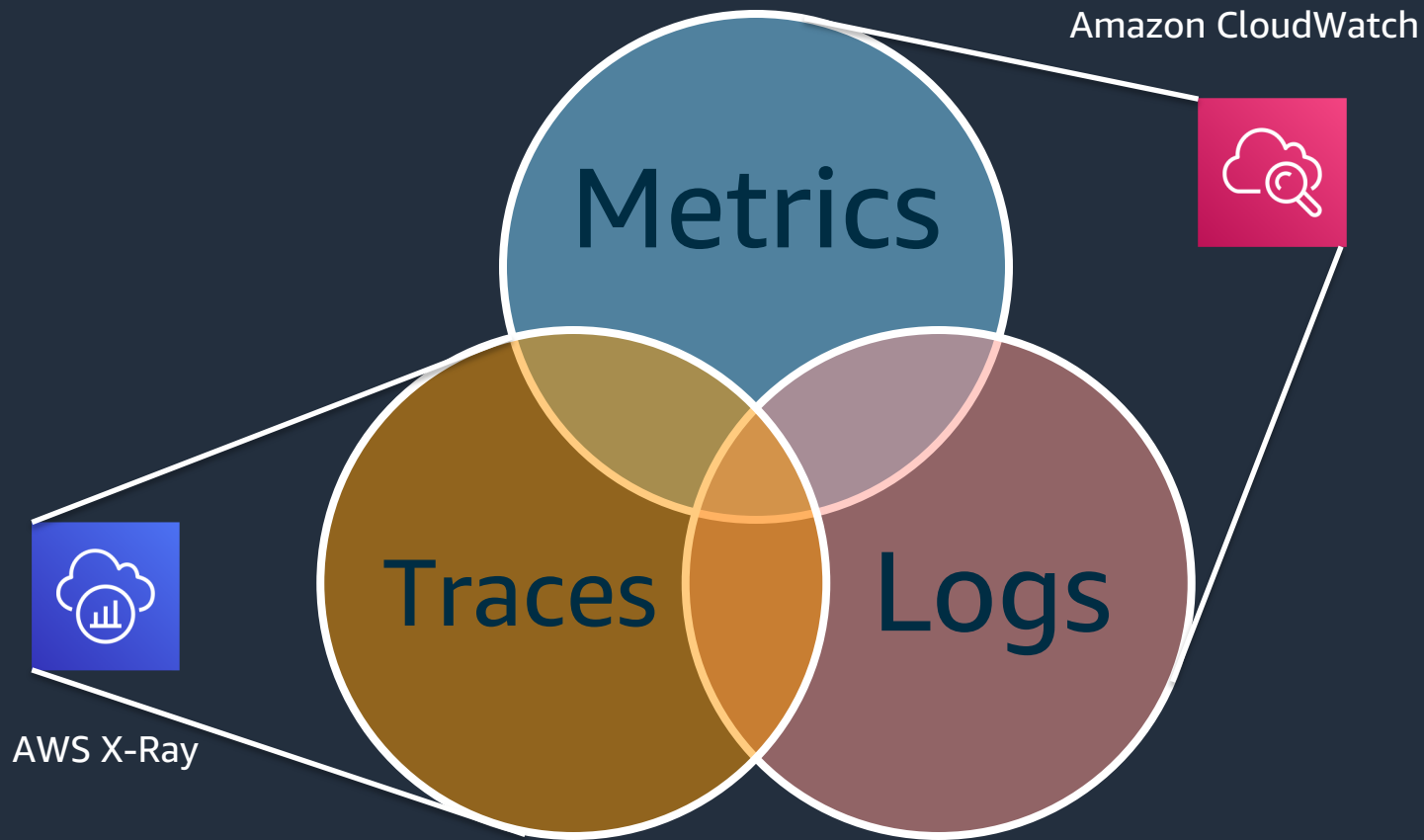
## you build it, you run it

開発者も日々の運用に参加しなさい。  
そうすればカスタマーのフィードバックが得られる。  
これが、サービスの質を高めるのだ。

-- Werner Vogels in May 2006 / Amazon.com CTO

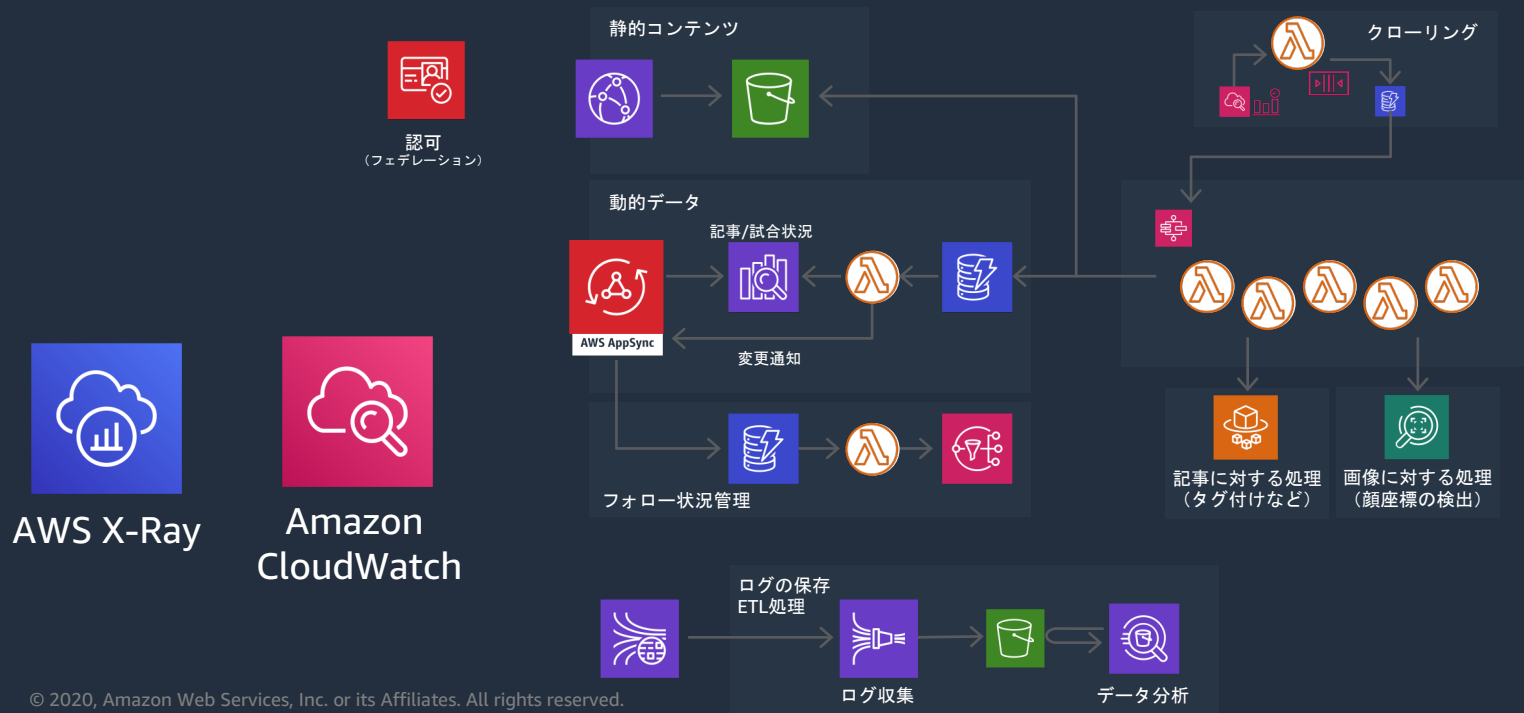


# モニタリングサービス

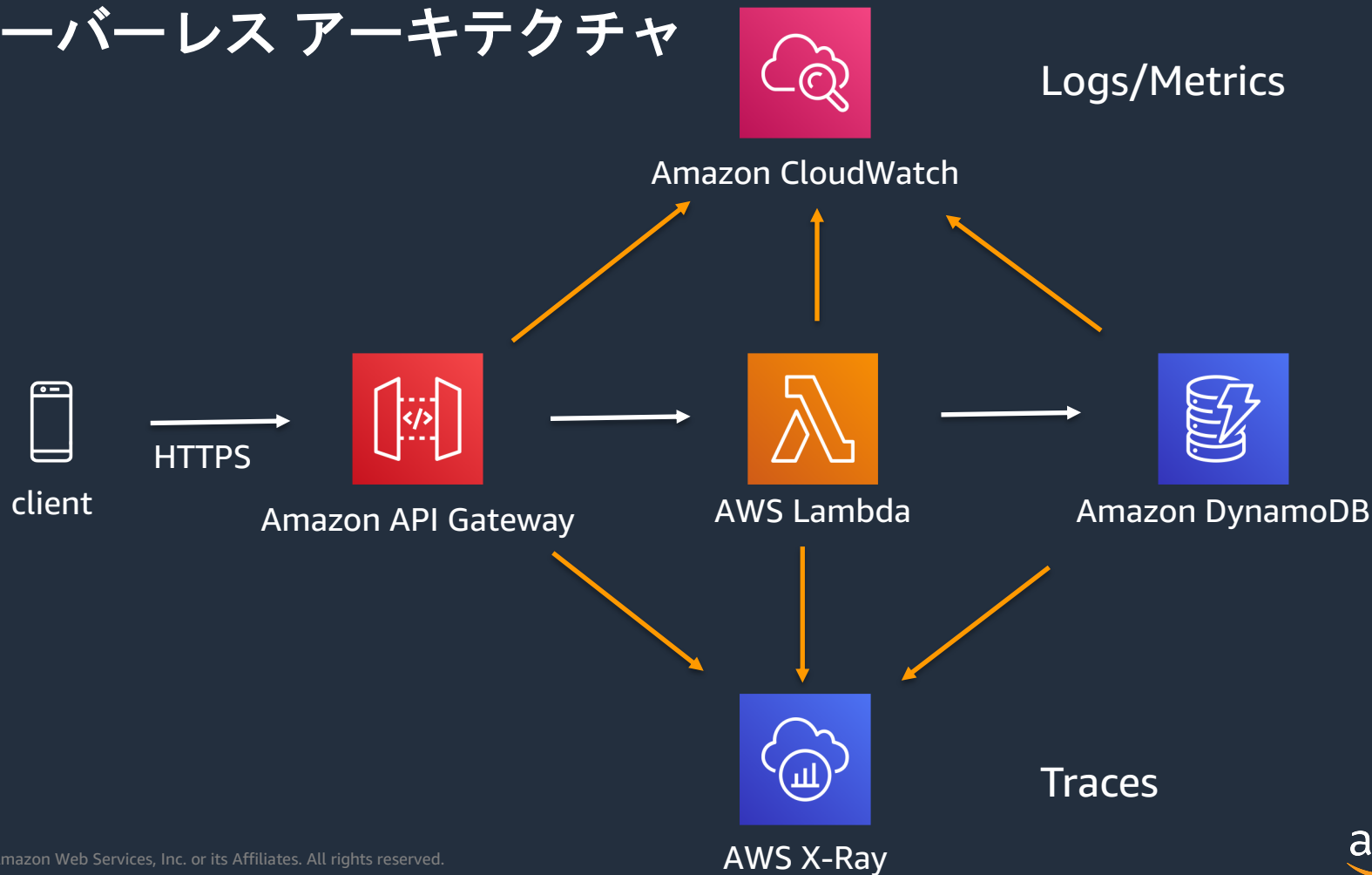


# サーバーレスのモニタリング

サーバーレス アーキテクチャは、多くの場合分散しており、個々の分散パーツがモニタリングを必要とします。



# サーバーレス アーキテクチャ



# Metricsの理解



# 開発者にとってのモニタリングとは？

いま下記の質問にすぐに回答できますか？

- Lambdaの平均処理時間
- エラー回数
- deploy前後での実行時間差分

自分が作ったものがどう使われているのか/動いているのか？ 改善点が見えるようにしておくのが重要。

また、運用を人に託すのであれば何を見るのかを提示する責任を持つ。





# AWS LambdaのMetrics

# AWS Lambda の Metrics

項目	単位	意味
Invocations	count	Lambdaの実行回数を計測したもの。=課金カウント ≠同時実行数
Duration	ms	Lambdaの実行時間を計測。（課金は100msで切り上げる） ※コールドスタートのLambda起動までの時間は含まない
Errors, Availability	count	Lambdaが正常終了しなかった回数を計測したもの Availabilityは%で表示される
Throttles	count	Throttleの発生回数 <u>アカウントにおける</u> Lambdaの同時起動数超過が発生している
IteratorAge	ms	ストリーム(Kinesis/DynamoDB)でのみ利用。バッチサイズ分取得したレコード終端の時刻とLambdaがイベントとして受信した時刻差で表示される
DeadLetterErrors	count	DLQを設定し、そのDLQへの書き込みが失敗すると増加する。

# AWS LambdaのManagement Consoleの再確認

b...

スロットリング

限定条件 ▼

アクション ▼

test ▼

テスト

保存

✓ 実行結果: 成功 (ログ)

▶ 詳細

実行結果の表示

テスト実行

設定

アクセス権限

モニタリング

モニタリング表示に  
切り替え

▼ Designer

 binary

 Layers (0)

 API Gateway

(2)

+ 送信先を追加

実行結果: 成功 (ログ)



▼ 詳細

下のセクションに、関数の実行から返された結果が表示されます。

```
{
  "statusCode": 200,
  "body": "\"Hello from Lambda!\""
}
```

## 関数の実行結果

### 概要

コード SHA-256

CzZO0a9QX4MGEg46ZPAEHwsJDDpQI3ISw3/gYe4E7G0=

リクエスト ID

d1527f2e-75d1-404c-856f-b42bbda5d90b

所要時間

5.65 ms

課金期間

100 ms

設定済みリソース

128 MB

使用中の最大メモリ

56 MB Init Duration: 106.60 ms

## 実行情報の概要

### ログ出力

下のセクションに、コード内のログ記録呼び出しが表示されます。これらはそれぞれ、CloudWatch ロググループ内でこの Lambda 関数に対応する単一行です。CloudWatch ロググループを表示するには、[ここをクリックし](#)、をクリックしてください。

START RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b Version: \$LATEST

```
[INFO] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b info
[WARNING] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b warn
[ERROR] 2019-09-05T03:09:26.463Z d1527f2e-75d1-404c-856f-b42bbda5d90b error
```

## 標準出力

END RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b

REPORT RequestId: d1527f2e-75d1-404c-856f-b42bbda5d90b Duration: 5.65 ms

Billed Duration: 100 ms Memory Size: 128 MB

Max Memory Used: 56 MB Init Duration: 106.60 ms

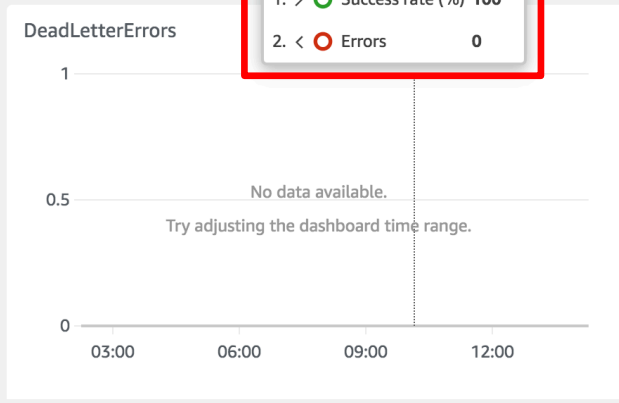
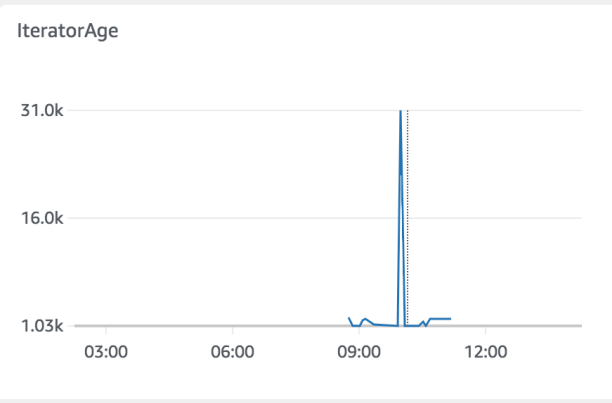
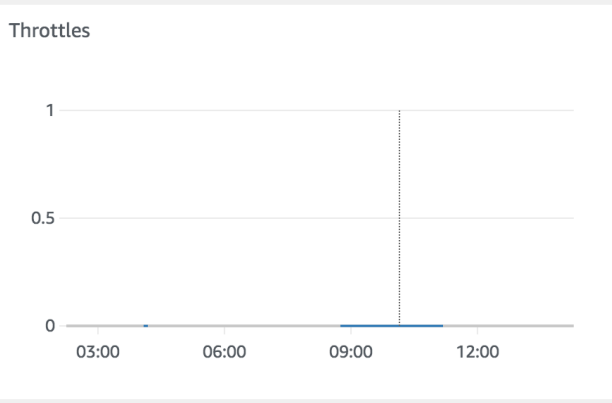
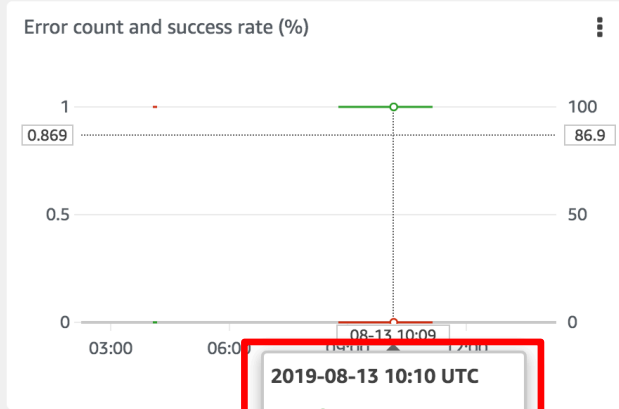
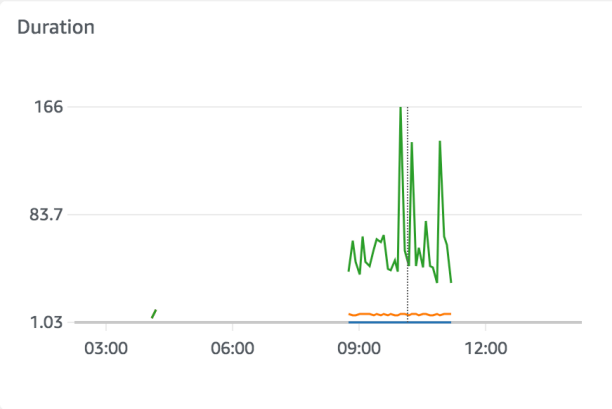
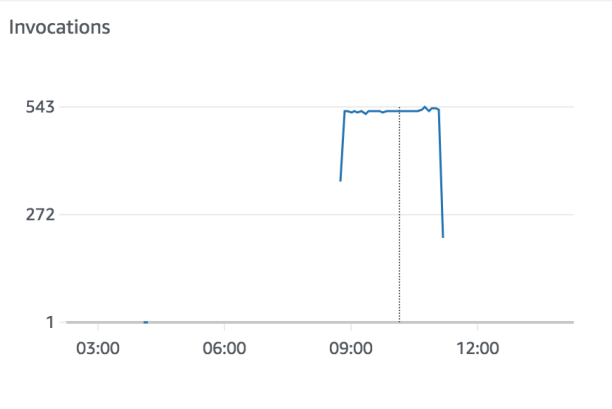
# AWS Lambda の Metrics



Amazon CloudWatch  
Dashboards  
に追加可能

表示時間範囲は変更可能

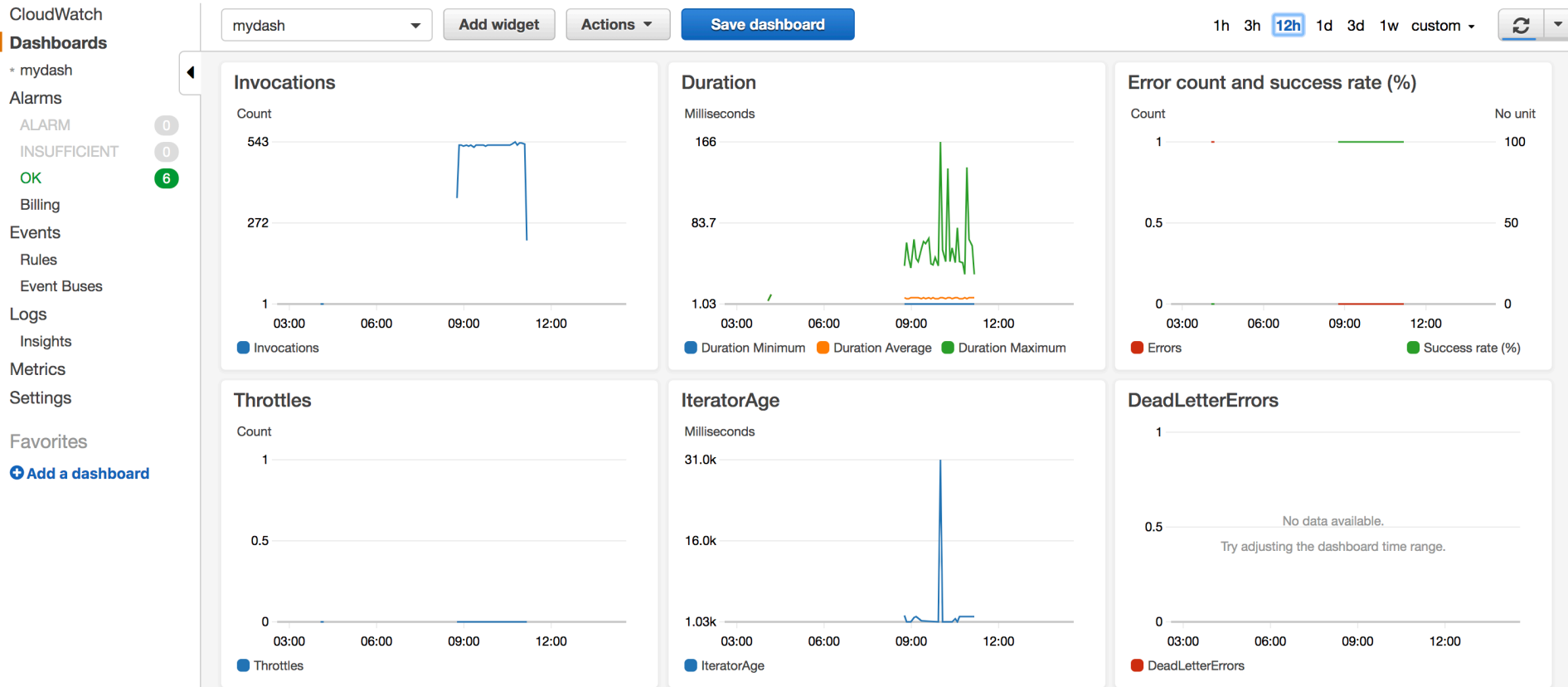
Add to dashboard | 1h 3h **12h** 1d 3d 1w | Refresh



**2019-08-13 10:10 UTC**

- 1. > ● Success rate (%) 100
- 2. < ● Errors 0

# Amazon CloudWatch Dashboardsに追加された Metrics



# AWS Lambda の Metrics

項目	単位	意味
Invocations	count	Lambdaの実行回数を計測したもの。=課金カウント ≠同時実行数
Duration	ms	Lambdaの実行時間を計測。（課金は100msで切り上げる） ※コールドスタートのLambda起動までの時間は含まない
Errors, Availability	count	Lambdaが正常終了しなかった回数を計測したもの Availabilityは%で表示される
Throttles	count	Throttleの発生回数 <u>アカウントにおける</u> Lambdaの同時起動数超過が発生している
IteratorAge	ms	ストリーム(Kinesis/DynamoDB)でのみ利用。バッチサイズ分取得したレコード終端の時刻とLambdaがイベントとして受信した時刻差で表示される
DeadLetterErrors	count	DLQを設定し、そのDLQへの書き込みが失敗すると増加する。

# AWS Lambda の Metrics

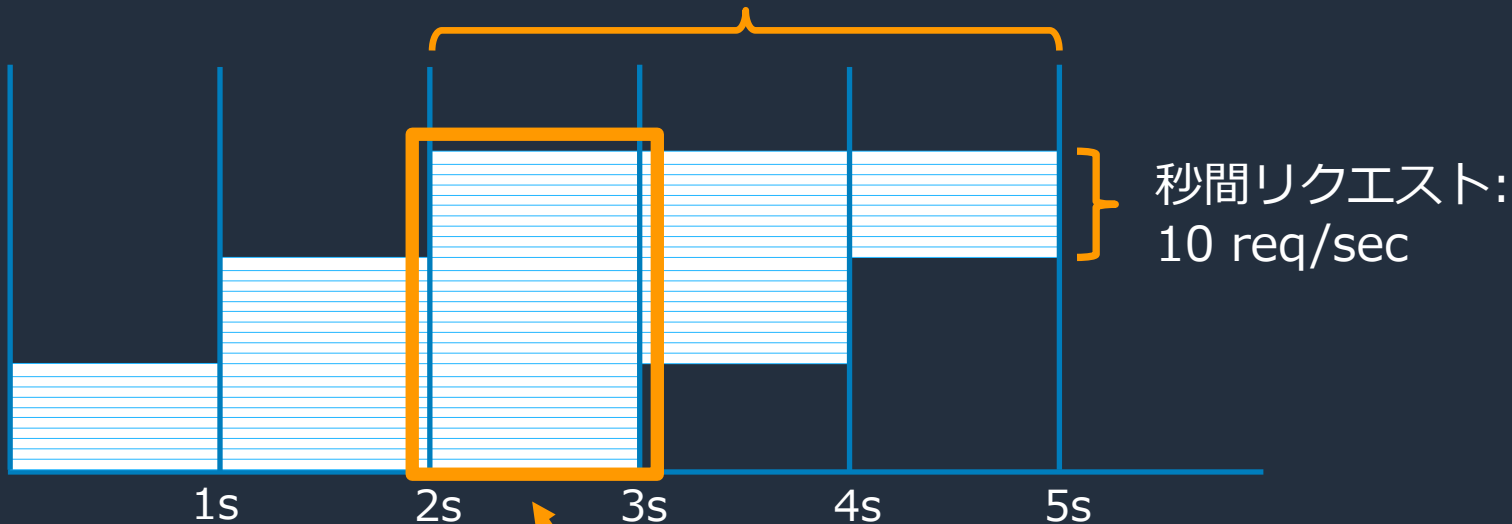
項目	単位	意味
Invocations	count	Lambdaの実行回数を計測したもの。=課金カウント ≠同時実行数
Duration	ms	Lambdaの実行時間を計測。（課金は100msで切り上げる） ※ <b>コールドスタートのLambda起動までの時間は含まない</b>
Errors, Availability	count	Lambdaが正常終了しなかった回数を計測したもの Availabilityは%で表示される
Throttles	count	Throttleの発生回数 <b>アカウントにおけるLambdaの同時起動数超過が発生している</b>
IteratorAge	ms	ストリーム(Kinesis/DynamoDB)でのみ利用。バッチサイズ分取得した <b>レコード終端の時刻とLambdaがイベントとして受信した時刻差</b> で 表示される
DeadLetterErrors	count	DLQを設定し、そのDLQへの書き込みが失敗すると増加する。



# 同時実行数とは

Throttles

関数の平均実行時間: 3s / exec



同時実行数 = “同時”に実行されているタイミング  
= 3 × 10 = 30

# AWS Lambdaの制限

[AWS ドキュメント](#) » [AWS Lambda](#) » [開発者ガイド](#) » [AWS Lambda の使用開始](#) » [AWS Lambda の制限](#)

## AWS Lambda の制限

AWS Lambda では、関数の実行と保存に使用できるコンピューティングおよびストレージリソースの量が制限されます。以下の制限は、リージョンごとに適用され、引き上げることができます。引き上げをリクエストするには、[サポートセンターコンソール](#)を使用してください。

リソース	デフォルトの制限
同時実行数	1,000
関数とレイヤーストレージ	75 GB

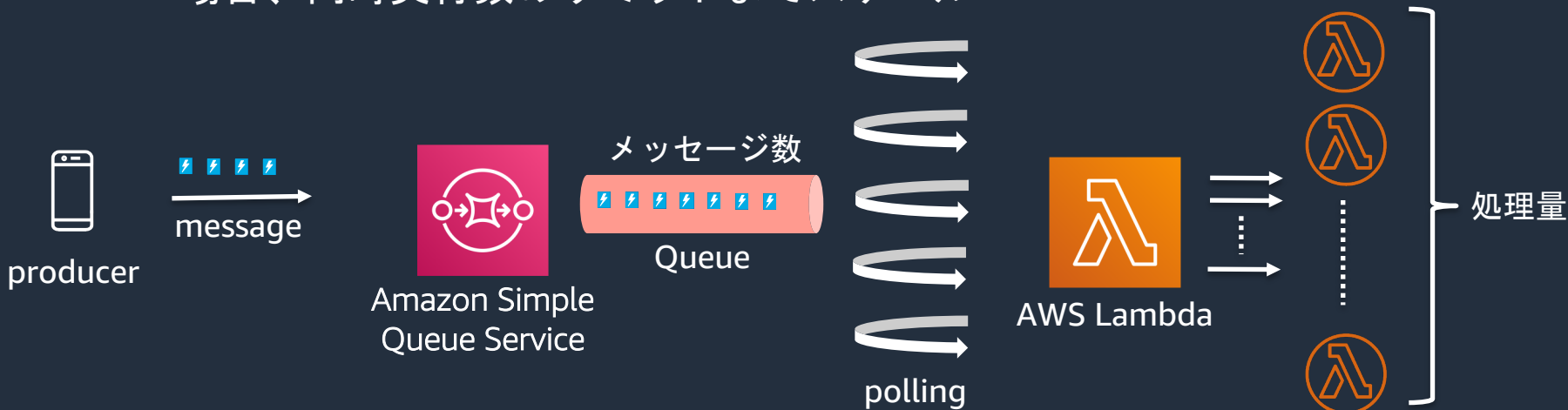
[https://docs.aws.amazon.com/ja\\_jp/lambda/latest/dg/limits.html](https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/limits.html)

Throttlingの対策としては

- Lambda関数をチューニングして平均実行時間を下げる
- SQSなどを用いて、非同期にリクエストを処理するようにアーキテクチャを変更する
- アカウントの同時実行数の制限緩和を申請する

### メッセージ到達時の挙動

- 5つのパラレルロングポーリング接続を使用してSQSのQueueをポーリング
  - メッセージ数 > 処理量 の傾向が続き、最終的に処理が追いつかない場合、同時実行数のリミットまでスケール



# SQSにおける注意事項

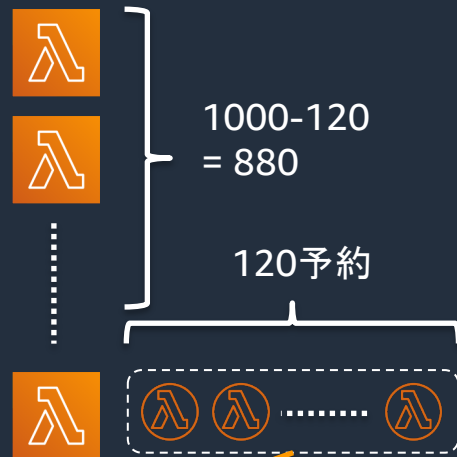
## 自動スケール

- Lambda関数単位に同時実行数を制限しない場合
  - アカウントの上限までスケールする可能性
  - 他のLambda関数の起動を妨げる恐れがある

## 対処方法

- 大規模に使用する際には同時実行数の設定も検討

AWS Account : 1000 in Tokyo



## 同時実行数

予約されていないアカウントの同時実行 **880**

- 予約されていないアカウントの同時実行の使用
- 同時実行の予約

**120**

Throttles

# AWS Lambda の Metrics

項目	単位	意味
Invocations	count	Lambdaの実行回数を計測したもの。=課金カウント ≠同時実行数
Duration	ms	Lambdaの実行時間を計測。（課金は100msで切り上げる） ※ <b>コールドスタートのLambda起動までの時間は含まない</b>
Errors, Availability	count	Lambdaが正常終了しなかった回数を計測したもの Availabilityは%で表示される
Throttles	count	Throttleの発生回数 <b>アカウントにおける</b> Lambdaの同時起動数超過が発生している
IteratorAge	ms	ストリーム(Kinesis/DynamoDB)でのみ利用。バッチサイズ分取得した <b>レコード終端の時刻とLambdaがイベントとして受信した時刻差</b> で表示される
DeadLetterErrors	count	DLQを設定し、そのDLQへの書き込みが失敗すると増加する。

# Lambda Function Lifecycle

Duration



- コンテナ生成
- S3からのZIPダウンロード
- ZIPファイルの展開
- Durationには含まれない

# Lambda Function Lifecycle

Duration

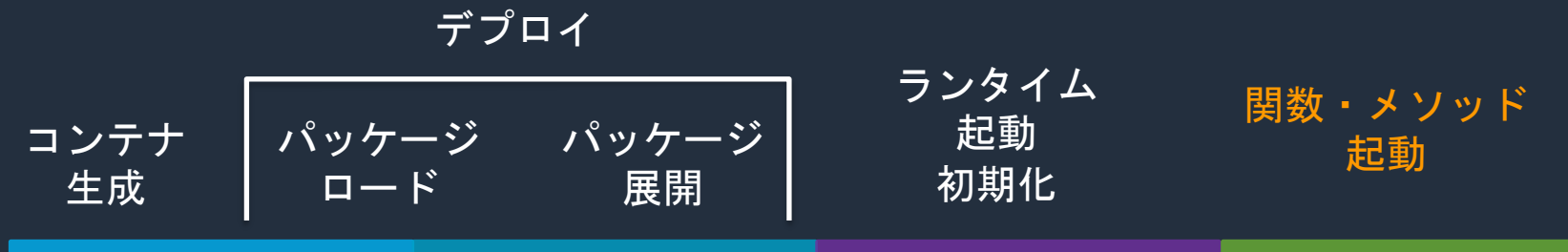


- 各ランタイムの初期化処理
- グローバルスコープ処理
- Durationには含まれない



# Lambda Function Lifecycle

Duration



- ハンドラーで指定した関数/メソッドの実行
- **Duration値は実行時間**

# Lambda Function Lifecycle

Duration



# AWS LambdaのMetrics監視のまとめ

## Throttlesを観測した場合

- どのAWS Lambdaが大量に使っているかの特定
  - CloudWatchのアカウント単位と個別Lambda単位の使用状況を確認
    - アカウント単位で飽和している => Limit Increaseの申請
    - Lambda単位で飽和している => Reserve Concurrency設定変更

## Errorsが増加している場合

- IAMでの権限設定漏れ
  - IAM Roleを他の関数と共有している場合、設定変更が行われたことを疑ってみる
- Queue/Streamは、batch sizeと全体処理時間とtimeout(min/sec)を確認
  - 単位処理時間 x batch size = 全体処理時間 > timeout
  - これも処理が進まないパターンになる

# AWS LambdaのMetrics監視のまとめ

## Dead Letter Queueが増え続ける

- 後段処理、連携システムのダウンなど、システム系としての障害の確認などを実施

## リトライ処理の考慮

- 非同期型であれば1回の実行、2回のリトライが行われ、すべて失敗した場合に、（設定していれば）Dead Letter Queueへ送られる



# Amazon API Gateway Metrics

# Amazon API Gateway の Log・Traceの有効化

dev ステージエディター ステージの削除 タグの設定

URL の呼び出し: https://[redacted]execute-api.ap-northeast-1.amazonaws.com/dev

設定 **ログ/トレース** ステージ変数 SDK の生成 エクスポート デプロイ履歴 ドキュメント履歴 Canary

ステージのログギングおよびトレース設定を指定します。

### CloudWatch 設定

**CloudWatch ログを有効化**  ⓘ

ログレベル

**リクエスト/レスポンスをすべてログ**

**詳細 CloudWatch メトリクスを有効化**  ⓘ

### カスタムアクセスのログ記録

**アクセスログの有効化**

**CloudWatch グループ**  ⓘ

**ログの形式**

入力の例:

[ログ変数のリスト](#)

X-Ray トレース [詳細はこちら](#)

**X-Ray トレースの有効化**  ⓘ [X-Ray サンプルングルールの設定](#)

# Amazon API Gateway の Log・Traceの有効化

dev ステージエディター ステージの削除 タグの設定

URL の呼び出し: https://[redacted]execute-api.ap-northeast-1.amazonaws.com/dev

設定 **ログ/トレース** ステージ変数 SDK の生成 エクスポート デプロイ履歴 ドキュメント履歴 Canary

ステージのログギングおよびトレース設定を指定します。

### CloudWatch 設定

CloudWatch ログを有効化  ⓘ

ログレベル

リクエスト/レスポンスをすべてログ

**詳細 CloudWatch メトリクスを有効化  ⓘ**

### カスタムアクセスのログ記録

アクセスログの有効化

CloudWatch グループ  ⓘ

ログの形式 

```
$context.identity.sourceIp $context.identity.caller $context.identity.user
[$context.requestTime] "$context.httpMethod $context.resourcePath
$context.protocol" $context.status $context.responseLength $context.requestId
```

入力の例:

[ログ変数のリスト](#)

X-Ray トレース [詳細はこちら](#)

X-Ray トレースの有効化  ⓘ [X-Ray サンプルングルールの設定](#)

# Amazon API Gateway の Log・Traceの有効化

dev ステージエディター ステージの削除 タグの設定

URL の呼び出し: [https://\[redacted\].execute-api.ap-northeast-1.amazonaws.com/dev](https://[redacted].execute-api.ap-northeast-1.amazonaws.com/dev)

設定 **ログ/トレース** ステージ変数 SDK の生成 エクスポート デプロイ履歴 ドキュメント履歴 Canary

ステージのログギングおよびトレース設定を指定します。

### CloudWatch 設定

CloudWatch ログを有効化  ⓘ

ログレベル

リクエスト/レスポンスをすべてログ

詳細 CloudWatch メトリクスを有効化  ⓘ

### カスタムアクセスのログ記録

アクセスログの有効化

CloudWatch グループ  ⓘ

ログの形式

入力の例:

[ログ変数のリスト](#)

X-Ray トレース [詳細はこちら](#)

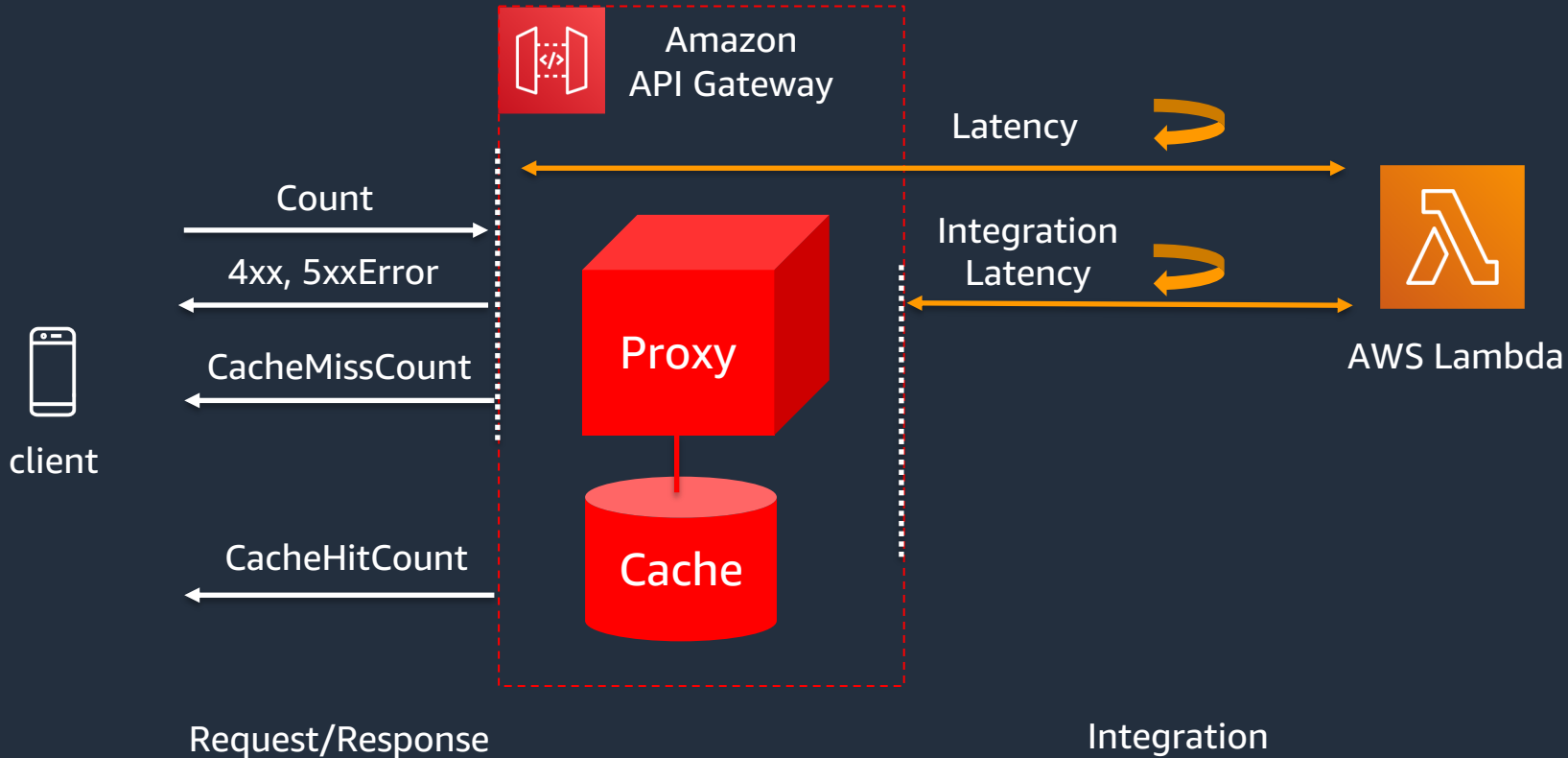
X-Ray トレースの有効化  ⓘ [X-Ray サンプルングルールの設定](#)



# Amazon API Gateway の Metrics

項目	単位	意味
4xxError	count	Sumは4xxエラーの合計数、Averageは4xxエラー率
5xxError	count	Sumは5xxエラーの合計数、Averageは5xxエラー率
CacheHitCount	count	APIキャッシュからレスポンスした、 Sumはキャッシュヒットした数、Averageはキャッシュヒット率
CacheMissCount	count	キャッシュを有効にしているが、バックエンドから応答した数 Sumはキャッシュミスヒット数、Averageはキャッシュミス率
Count	count	APIのリクエスト数
IntegrationLatency	ms	API Gatewayが、バックエンドへリクエストして、バックエンドからレスポンスが返却されるまでの時間
Latency	ms	API Gatewayがクライアントからリクエストを受け取り、クライアントに返却するまでの時間。API Gatewayのオーバヘッドも含まれる

# Amazon API Gateway Metrics



# API GatewayのCache設定

設定 ログ/トレース ステージ変数 SDK の生成 エクスポート デプロイ履歴 ドキュメント履歴 Canary

## キャッシュ設定

キャッシュのステータス AVAILABLE

キャッシュ全体をフラッシュ

API キャッシュを有効化

API キャッシュを有効にすると追加料金が発生します。これは無料利用枠の対象ではありません。詳細については、[料金表を参照してください](#)

キャッシュキャパシティー 0.5GB

キャッシュデータを暗号化する

キャッシュ有効期限 (TTL)

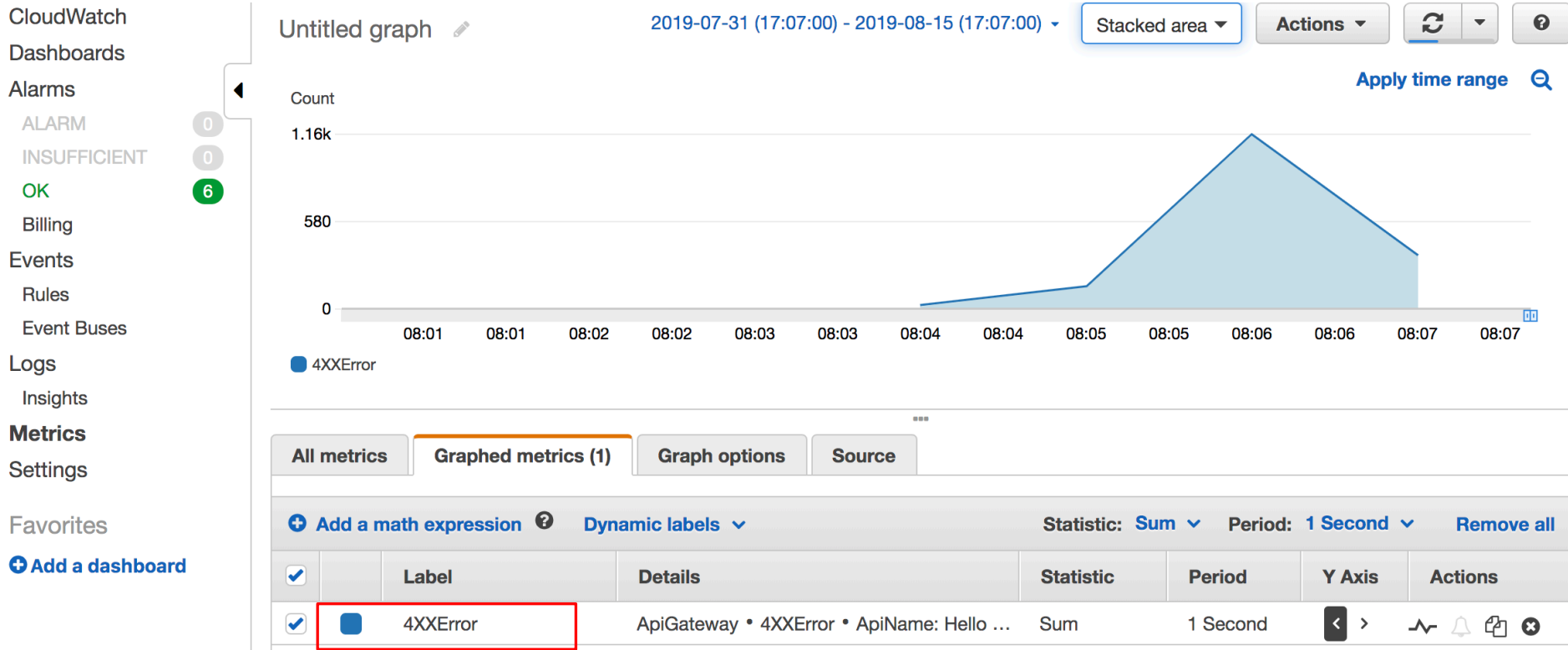
## キーごとのキャッシュの無効化

認可が必要

未認可リクエストの処理 キャッシュコントロールヘッダーを無視、レスポンスヘッダーで警告を追加

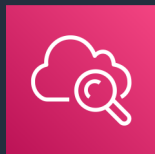
# API Gatewayのスロットリング監視

MetricsとしてThrottlesが定義されていないためHTTP 429 のレスポンスを確認する (HTTP 429 = too many requests)





# Log運用と可視化



# Amazon CloudWatch でのログ管理



# Amazon CloudWatch Logsの使い方を知る

CloudWatchは簡単にLog出力でき、とても便利です。

CloudWatch

Dashboards

Alarms

ALARM

INSUFFICIENT

OK

Billing

Events

Rules

Event Buses

Logs

Insights

Metrics

Settings

CloudWatch > Log Groups > St...

Search Log Group

Create Log Stream

Delete Log Stream

Filter: Log Stream Name Prefix

Log Streams

Last Event

<input type="checkbox"/>	2019/08/12/[\$LATEST]a9ffa3f08823042129b1f0e97543684af	2019-08-	ist-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]6...d1a2044fca7613b03b4129499	2019-08-	ist-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]4...350c849b6a802e9cd74cc7af1	2019-08-12 10:...	arn:aws:logs:ap-northeast-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]2...adc154cc1af65e565942c5f33	2019-08-12 10:16 UTC+9	arn:aws:logs:ap-northeast-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]4...3b4ef4ea3ac251414580e2799	2019-08-12 18:13 UTC+9	arn:aws:logs:ap-northeast-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]3...3a4c847389a782ba3ea08667a	2019-08-12 18:11 UTC+9	arn:aws:logs:ap-northeast-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]6...23f34c80a77bee1df2a48721	2019-08-12 18:10 UTC+9	arn:aws:logs:ap-northeast-1:4
<input type="checkbox"/>	2019/08/12/[\$LATEST]222d1cb2f6604f229b1a14d3e0f168d9	2019-08-12 17:59 UTC+9	arn:aws:logs:ap-northeast-1:4

ログの内容を横断的に確認する機能が欲しい

Lambdaのコンテナ起動単位でログストリームが分かれている。

# フィルターパターンの利用

- ログが正規化されていなくても一致する文字列の検索も可能。
- アプリケーションが出力するログフォーマットと利用ルールを明確化することでメトリックスの割り当てもできる。
- printで出力するのではなく、logger関数などを利用してログ出力することがおすすめ

## ログメトリックスフィルターの定義

ロググループのフィルター: /aws/lambda/helloworld

メトリックスフィルタを使用し、ロググループ内のイベントが CloudWatch Logs に送信されるときに、それらのイベントを自動的にモニタリングできます。特定の用語のモニタリングやカウントを行ったり、ログイベントから値を抽出したりでき、その結果をメトリックスに関連付けることができます。 [パターン構文の詳細はこちら](#)。

フィルターパターン

START

例の表示

テストするログデータの選択

2019/08/10/[\$LATEST]66995e98112d34c50b536982105b0d3c1

クリア

パターンのテスト

START RequestId: 58d92802-5ae7-4d08-aec6-cc1bd7ec13ce Version: \$LATEST  
END RequestId: 58d92802-5ae7-4d08-aec6-cc1bd7ec13ce  
REPORT RequestId: 58d92802-5ae7-4d08-aec6-cc1bd7ec13ce Duration: 1.83 ms Billed Duration: 100 ms Me  
START RequestId: da780748-f9a4-4a81-91e9-058ae93f2019 Version: \$LATEST  
END RequestId: da780748-f9a4-4a81-91e9-058ae93f2019  
REPORT RequestId: da780748-f9a4-4a81-91e9-058ae93f2019 Duration: 1.32 ms Billed Duration: 100 ms Me  
START RequestId: 36d91912-2767-456d-8c3d-05598f017472 Version: \$LATEST

結果

サンプルログの9個のイベントから3の一致が見つかりました。

テスト結果の表示

キャンセル

メトリックスの割り当て



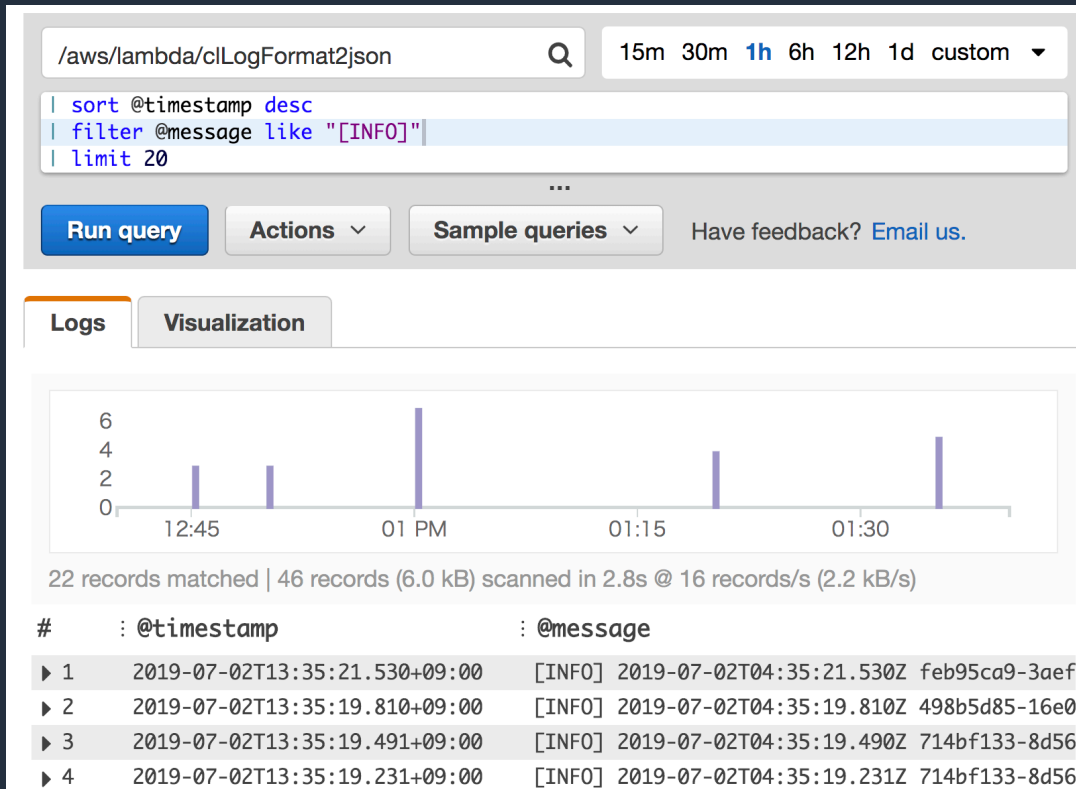
# CloudWatch Logs filter pattern

## filter patternの仕様

- 大文字、小文字は区別される
- 正規表現は利用できない
- 簡単なパターンマッチのみ利用可能
- JSON形式、スペース区切りは利用可能

# CloudWatch Logs Insightsの活用

**fields** @timestamp, @message  
| **sort** @timestamp desc  
| **filter** @message like "[INFO]"  
| **limit** 20



The screenshot shows the AWS CloudWatch Logs Insights interface. At the top, the log group path is `/aws/lambda/clLogFormat2json`. The query editor contains the following query:

```
| sort @timestamp desc  
| filter @message like "[INFO]"  
| limit 20
```

Below the query editor, there are buttons for "Run query", "Actions", and "Sample queries", along with a link to "Have feedback? Email us." The "Visualization" tab is selected, showing a bar chart of log counts over time. The chart has a y-axis from 0 to 6 and an x-axis with time markers at 12:45, 01 PM, 01:15, and 01:30. Below the chart, the following statistics are displayed: "22 records matched | 46 records (6.0 kB) scanned in 2.8s @ 16 records/s (2.2 kB/s)".

Below the visualization, a table of log records is shown with the following columns: #, @timestamp, @message, and @message. The table contains 4 records:

#	@timestamp	@message	@message
▶ 1	2019-07-02T13:35:21.530+09:00	[INFO]	2019-07-02T04:35:21.530Z feb95ca9-3aef
▶ 2	2019-07-02T13:35:19.810+09:00	[INFO]	2019-07-02T04:35:19.810Z 498b5d85-16e0
▶ 3	2019-07-02T13:35:19.491+09:00	[INFO]	2019-07-02T04:35:19.490Z 714bf133-8d56
▶ 4	2019-07-02T13:35:19.231+09:00	[INFO]	2019-07-02T04:35:19.231Z 714bf133-8d56

# ログ保管期間

- Lambdaのログ保管期間はデフォルトが無期限
- 必要な期間に圧縮することを忘れがち

CloudWatch > ロググループ

メトリクスフィルターの作成    アクション ▾

フィルター: ロググループ名のプレフィックス ×

ロググループ	インサイト	次の期間経過後にイベントを失効
<input type="radio"/> /aws/apigateway/welcome	を試す	失効しない
<input type="radio"/> /aws/lambda/ConditionFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/CreateCognitoUser	を試す	失効しない
<input type="radio"/> /aws/lambda/CreateCognitoUserjs	を試す	失効しない
<input type="radio"/> /aws/lambda/DLQfunc	を試す	失効しない
<input type="radio"/> /aws/lambda/DynamoDBreadfunc	を試す	失効しない
<input type="radio"/> /aws/lambda/DynamoDBtoLambda	を試す	失効しない
<input type="radio"/> /aws/lambda/ErrorFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/JWTFuncion	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataEgestFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataIngestFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataIngestLoopFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/LogOperateFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/LogStreamTriggerFunc	を試す	失効しない

メトリクスフィルターの作成

アクション ▾

フィルター: ロググループ名のプレフィックス ×

ロググループ	インサイト	次の期間経過後にイベントを失効
<input type="radio"/> /aws/apigateway/welcome	を試す	失効しない
<input checked="" type="radio"/> /aws/lambda/ConditionFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/CreateCognitoUser	を試す	失効しない
<input type="radio"/> /aws/lambda/CreateCognitoUserjs	を試す	失効しない
<input type="radio"/> /aws/lambda/DLQfunc	を試す	失効しない
<input type="radio"/> /aws/lambda/DynamoDBreadfunc	を試す	失効しない
<input type="radio"/> /aws/lambda/DynamoDBtoLambda	を試す	失効しない
<input type="radio"/> /aws/lambda/ErrorFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/JWTFunction	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataEgestFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataIngestFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/KinesisDataIngestLoopFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/LogOperateFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/LogStreamTriggerFunc	を試す	失効しない
<input type="radio"/> /aws/lambda/LogsToElasticsearch_logsearch	を試す	失効しない
<input type="radio"/> /aws/lambda/PrivateLambda	を試す	失効しない
<input type="radio"/> /aws/lambda/SignUpForCognito	を試す	失効しない
<input type="radio"/> /aws/lambda/SignupCognitoUser	を試す	失効しない
<input type="radio"/> /aws/lambda/VPCLambda	を試す	失効しない
<input type="radio"/> /aws/lambda/VPCLambdaFunction	を試す	失効しない
<input type="radio"/> /aws/lambda/aries-dev-20190722145212-auth-UserPoolClient	を試す	失効しない

## 保持期間の編集 ×

保持期間:  失効しない

1日

3日間

5日間

1週間 (7日間)

2週間 (14日間)

1か月 (30日間)

2か月 (60日間)

3か月 (90日間)

4か月 (120日間)

5か月 (150日間)

6か月 (180日間)

1年 (365日間)

13か月 (400日間)

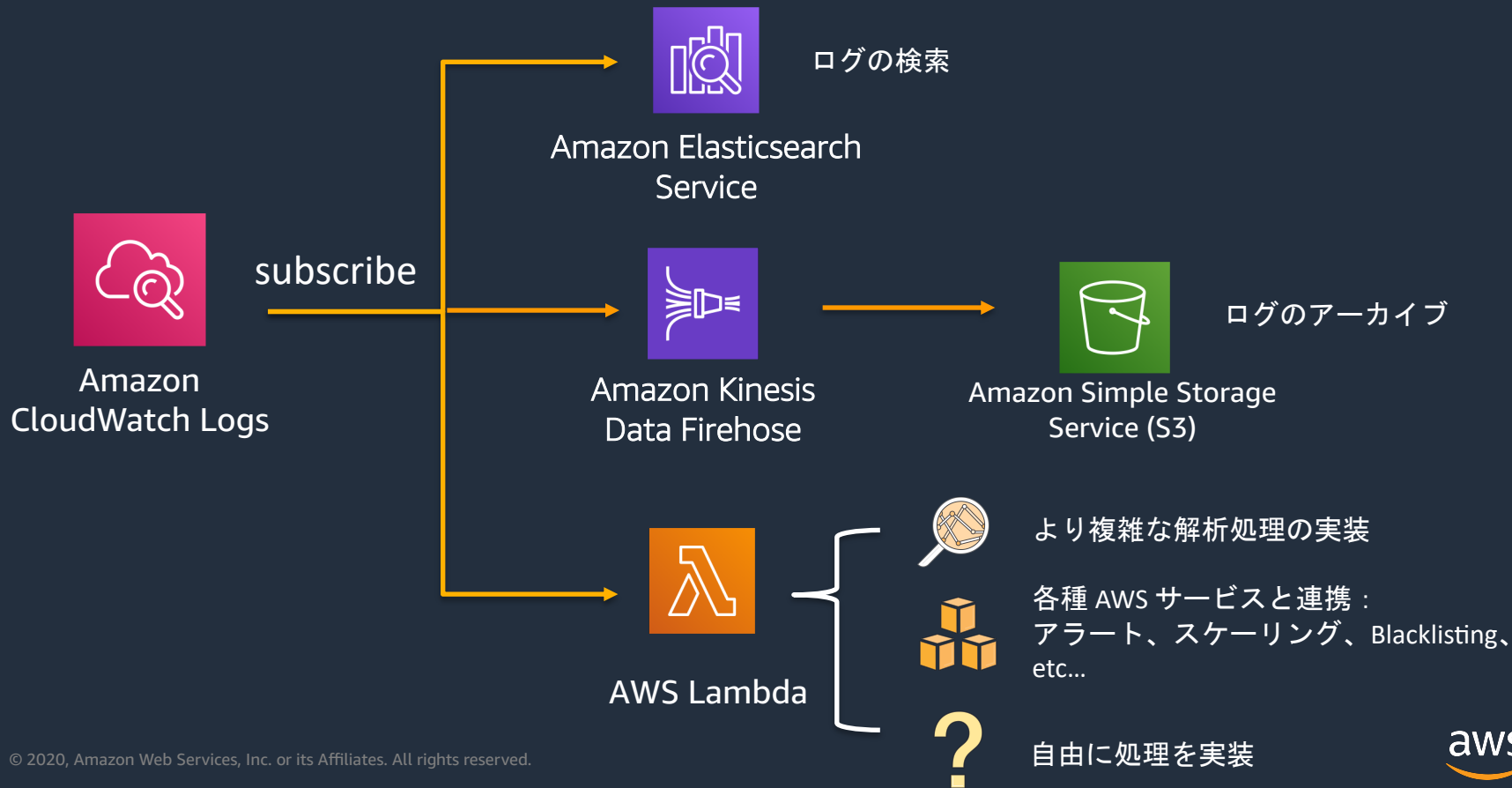
18か月 (545日間)

2年 (731日間)

5年 (1827日間)

10年 (3653日間)

# 他サービスに連携してログの活用





# Traceを使った可視化

# AWS X-Ray



## リクエスト実行状況の確認

アプリケーションを構成する個々のサービスやリソースの実行結果ステータスを集計し、アプリケーションの実行状況をエンドツーエンドで確認可能



## アプリケーションの問題の検出

アプリケーションの実行状況についての関連する情報を収集し、問題の根本原因を調査可能



## AWSとの連携

Amazon EC2, Amazon ECS, AWS Lambda, AWS Elastic Beanstalk と連携



## アプリケーションのパフォーマンス向上

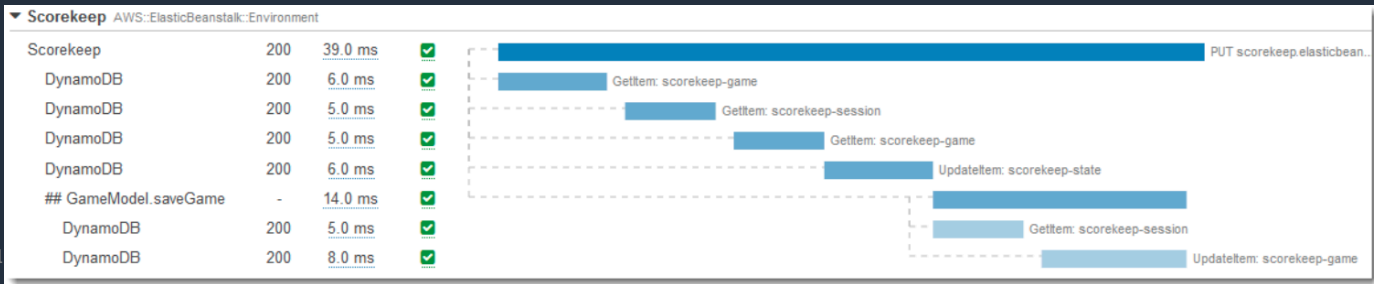
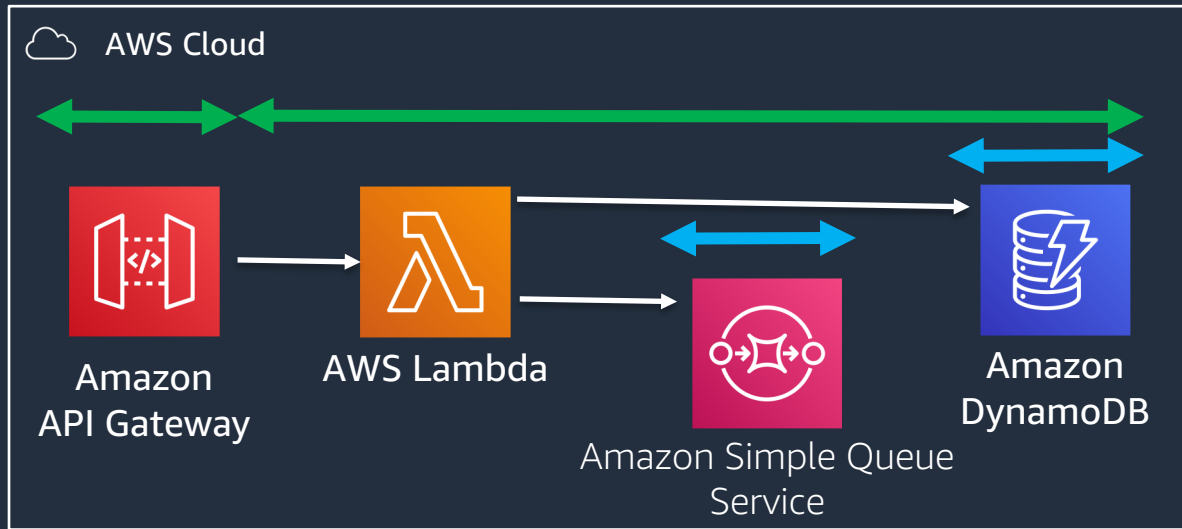
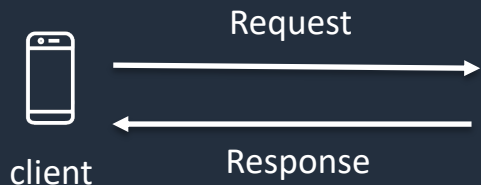
サービスやリソースの関係をリアルタイムで表示し、レイテンシ増加やパフォーマンス低下などのボトルネックを特定可能



## さまざまなアプリケーション向けの設計

非同期のシンプルなイベント呼び出し、3層のウェブアプリケーション、数千のサービスから構成される複雑なマイクロサービスも分析可能

# AWS X-Ray コンポーネント

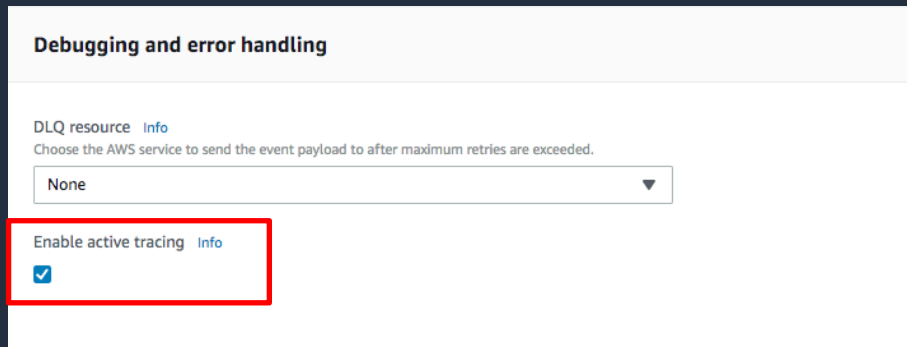




# AWS LambdaでX-Rayを利用するためには、

Lambda用パッケージにX-Ray SDKを追加し、アクティブトレースをONにする。  
IAM roleも必要、Managed policyが用意されている

- Pythonの場合
  - Python 2.7, Python3.6以降
- Node.jsの場合
  - Node.js 4.3以降
- Javaの場合
  - Java8以降
- Goの場合
  - Go1.7以降
- .NETの場合
  - .NET Core 2.0以降



**Debugging and error handling**

DLQ resource [Info](#)  
Choose the AWS service to send the event payload to after maximum retries are exceeded.

None ▼

Enable active tracing [Info](#)

CLIであれば、  
`--tracing-config` オプション

# 実装例(Python)

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
```

```
patch_all()
```

```
def main(event, context):
```

```
    xray_recorder.begin_segment('main segment')
```

```
    (main処理)
```

```
        xray_recorder.begin_subsegment('sub segment')
```

```
        (処理A)
```

```
        xray_recorder.end_subsegment('sub segment')
```

```
    xray_recorder.end_segment
```

```
    return
```

# 実装例(Python)

```
from aws_xray_sdk.core import xray_recorder  
from aws_xray_sdk.core import patch_all
```

## patch\_all()

対応しているライブラリにパッチ適用

pythonの場合

-botocore, boto3

-requests

-sqlite3

-mysql-connector-python

など

patch\_allではなく patch('boto3')などとすることも可能

# 実装例(Python)

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
```

```
patch_all()
```

```
def main(event, context):
```

```
    xray_recorder.begin_segment('main segment')
```

処理全体のセグメントを定義

```
    (main処理)
```

```
        xray_recorder.begin_subsegment('sub segment')
```

```
        (処理A)
```

```
        xray_recorder.end_subsegment('sub segment')
```

```
    xray_recorder.end_segment
```

```
    return
```

# 実装例(Python)

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all
```

```
patch_all()
```

```
def main(event, context):
```

```
    xray_recorder.begin_segment('main segment')
```

```
    (main処理)
```

```
        xray_recorder.begin_subsegment('sub segment')
```

```
        (処理A)
```

```
        xray_recorder.end_subsegment('sub segment')
```

```
    xray_recorder.end_segment
```

```
    return
```

処理の中でサブセグメントを作ることも可能

# サービスグラフ

各ノードの呼び出しの結果を  
色で分類、割合を円グラフに  
(サービスマップ)

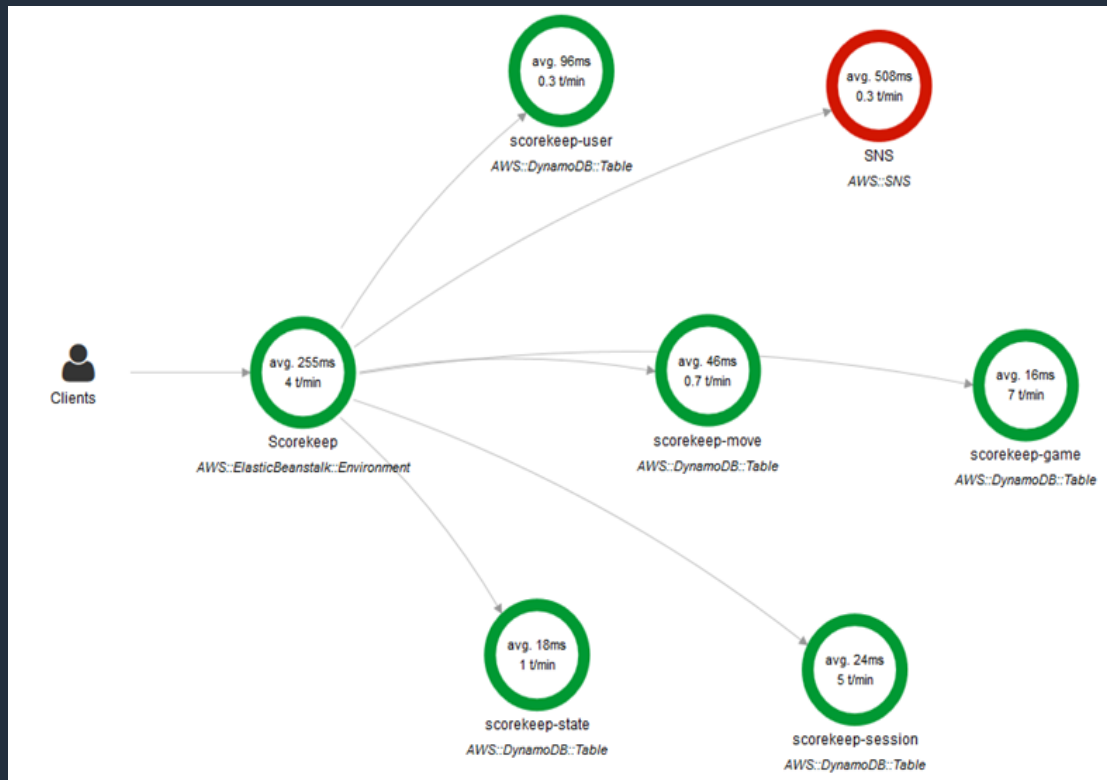
**グリーン** 成功した呼び出し

**レッド** 5xx errors

**イエロー** 4xx errors

**パープル** 429 Too Many Requests  
(スロットリングエラー)

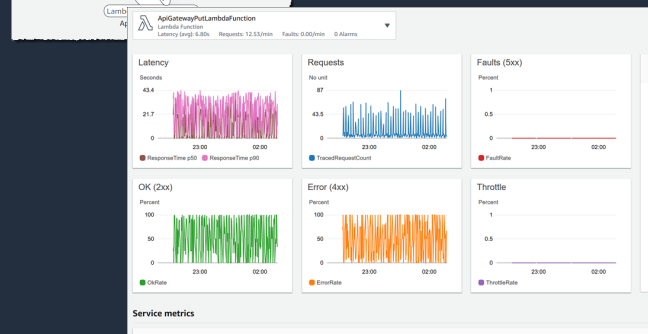
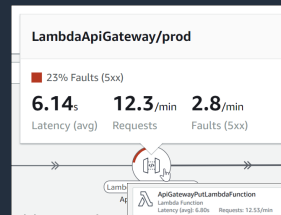
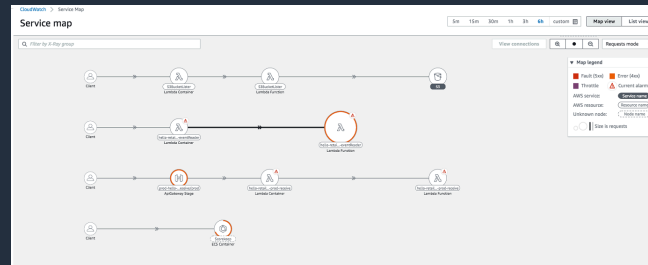
- ・ 平均レイテンシ (ms)
- ・ トレース数 (trace/min)
- ・ サービス名
- ・ サービスの分類



NEW

# Amazon CloudWatch ServiceLensを発表

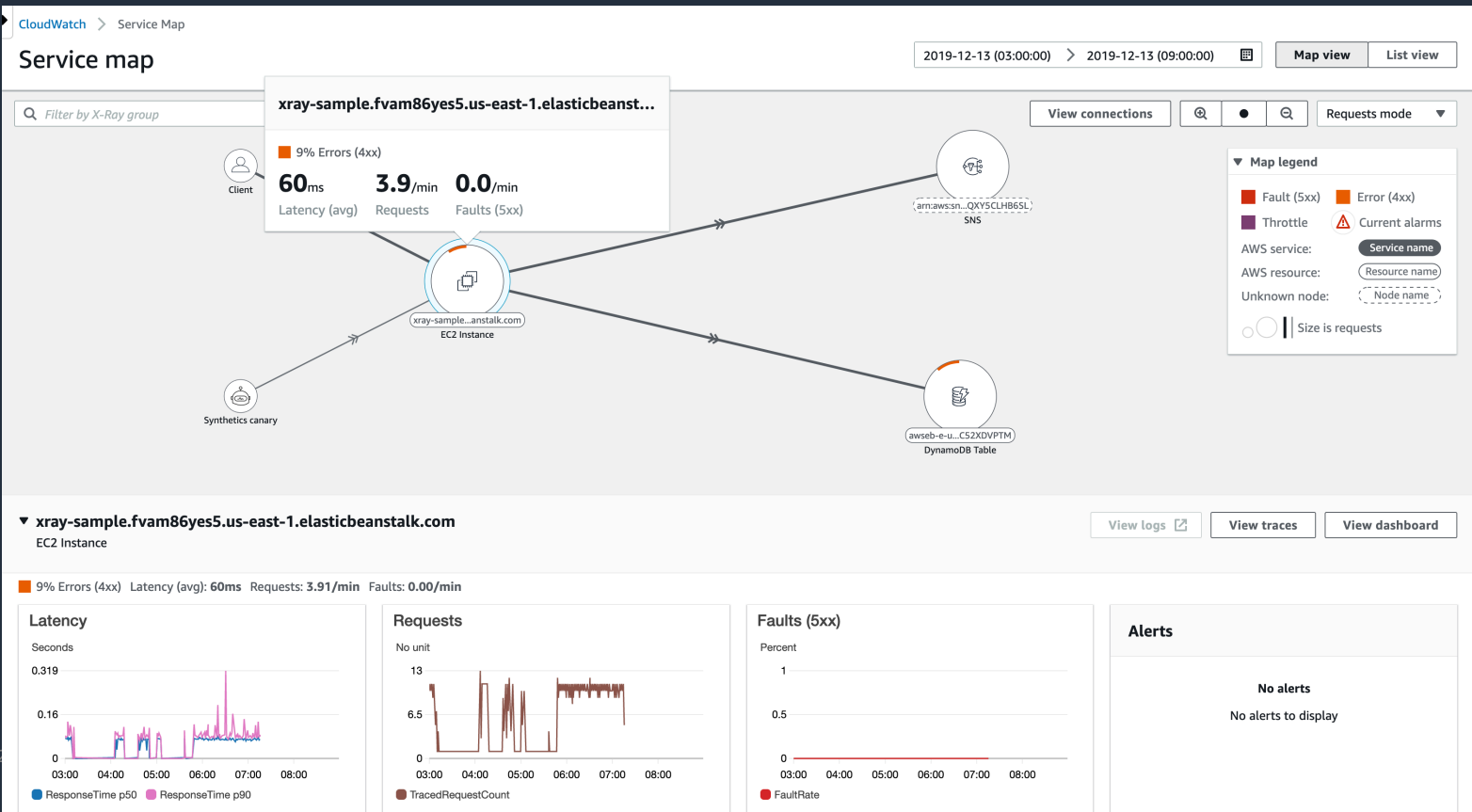
- アプリケーションの健全性、パフォーマンスなどを視覚化し分析を容易にするための機能
- CloudWatchメトリクスとログ、X-Rayからのトレース情報を結びつけてアプリとその依存関係をビジュアライズする
- メトリクスデータとトレースデータを可視化することでシステム全体を俯瞰し問題箇所を特定、原因を掘り下げることが容易に
- 東京リージョンを含むX-Rayが利用可能なリージョンで利用可能。ServiceLens自体は無料だがX-Rayのトレース費用が発生する



# Service Lens (1/2) Service map

XRayのサンプルを us-east-1 (N.Virginia) で実行することで ServiceLensの動作を確認できます

[https://docs.aws.amazon.com/ja\\_jp/xray/latest/devguide/xray-gettingstarted.html](https://docs.aws.amazon.com/ja_jp/xray/latest/devguide/xray-gettingstarted.html)





# Service Lens (2/2) Traces

Retrieved 685 traces Learn more

CloudWatch > Traces

## Traces [Info](#)

2019-12-13 (12:00:00) > 2019-12-13 (18:00:00)

You can use the filters to narrow down the table of traces. Adjust the filters to find traces that show performance issues or that relate to specific nodes or requests.

xray-sample.fvam86yes5.us-east-1.elasticbeanstalk.com  
EC2 Instance  
Latency (avg): 60ms Requests: 3.91/min Faults: 0.00/min 0 Alarms

**Filters** [Info](#) View in X-Ray Analytics

Choose a set of filters for your traces

**Filter type**

- Trace status
- Response code
- URL
- Resource ARN
- User
- User agent
- Client IP
- Instance ID
- Response time
- Custom query

**Traces by Response code** Add response code to filter

Select rows from the table and add them to your filter

Find Response code

<input type="checkbox"/>	Response code	Successful requests	% of retrieved traces	Mean response time
<input type="checkbox"/>	409	0.0%	10%	0.03s
<input type="checkbox"/>	200	100.0%	10%	0.00s
<input type="checkbox"/>	201	100.0%	81%	0.08s

A group of people are gathered around a table in a meeting. A woman in a blue shirt is pointing at a laptop screen. Another person is typing on the laptop. The scene is brightly lit, suggesting a modern office environment. The text 'まとめ' is overlaid on the image in a white font on a dark rectangular background.

まとめ

# まとめ

AWS サーバーレスサービスに関わる考え方やMetricsをご紹介してきました。

- アプリケーション開発方法やログ出力などは、これまでと変わらない知識と経験がサーバーレスでも活用できます
- ログフォーマットなども通常のアプリケーション設計と変わらない
  - CloudWatch Logsが利用できる
- サーバーレスサービスの制約や仕様を理解し、今までのアプリケーション開発経験と組み合わせることができる。

ありがとうございました



@\_kensh