# 全部教えます！
# サーバレスアプリのアンチパターンとチューニング

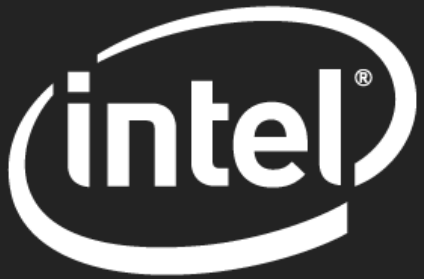
Keisuke Nishitani (@Keisuke69), Specialist SA, Serverless

Jun 2, 2017



amazon web services

# THANKS TO OUR FRIENDS AT:






amazon web services

# 本セッションのFeedbackをお願いします

受付でお配りしたアンケートに本セッションの満足度やご感想などをご記入ください
アンケートをご提出いただきました方には、もれなく素敵なAWSオリジナルグッズをプレゼントさせていただきます

アンケートは各会場出口、パミール3FのEXPO展示会場内にて回収させて頂きます



amazon web services

# Who am I

**Keisuke Nishitani**

Specialist Solutions Architect, Serverless
Amazon Web Service Japan K.K

@Keisuke69

Keisuke69

Keisuke69x

Keisuke69

Keisuke69



amazon web services

# 本セッションの対象

AWS Lambdaおよびサーバレス自体は知っている

実際に自分で実装したことがある

これから自分で実装しようとしている

# なぜ私のLambdaファンクションは遅いのか？



amazon web services

# 遅くなる理由

プログラムの問題

コンピューティングリソースの不足

コールドスタート

アーキテクチャの問題

同時実行数

# 遅くなる理由

プログラムの問題

コンピューティングリソースの不足

コールドスタート

アーキテクチャの問題

同時実行数

# プログラムの問題

Lambdaファンクションとして実装されているプログラムのロジックそのものの問題

- 実際のところ、プラットフォーム側でできることはない

各言語のベストプラクティス、最適化手法はそのまま当てはまる

# 遅くなる理由

プログラムの問題

コンピューティングリソースの不足

コールドスタート

アーキテクチャの問題

同時実行数

# コンピューティングリソース不足

メモリ設定

- 設定値としてはメモリとなっているが実際はコンピューティングリソース全体の設定
- メモリサイズと比例してCPU能力も割り当てられる
- メモリ設定はパフォーマンス設定と同義
- コストを気にしがちだが、メモリを増やすことで処理時間がガクンと減り、結果的にコストはそれほど変わらずとも性能があがることもある

少しずつ調整し、変更しても性能が変わらない値が最適値

# 遅くなる理由

プログラムの問題

コンピューティングリソースの不足

コールドスタート

アーキテクチャの問題

同時実行数

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

- VPCを利用する場合だけ
- 10秒～30秒かかる
- Durationには含まれない

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

- 指定されたランタイム
- S3からのダウンロードとZIPファイルの展開
- Durationには含まれない

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

- 各ランタイムの初期化処理
- グローバルスコープの処理もこのタイミングで実行される
- Durationには含まれない

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

- ハンドラーで指定した関数/メソッドの実行
- いわゆるDurationの値はここの実行時間

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

すべてを実行するのがコールドスタート

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

基本的に毎回同じ内容が実行される

# Lambdaファンクション実行時に起きていること

1. （ENIの作成）
2. コンテナの作成
3. デプロイパッケージのロード
4. デプロイパッケージの展開
5. ランタイム起動・初期化
6. 関数/メソッドの実行

基本的に毎回同じ内容が実行される

再利用することで省略して効率化
（ウォームスタート）

# コールドスタートが起こる条件

簡単に言うと、利用可能なコンテナがない場合に発生

- そもそも1つもコンテナがない状態
- 利用可能な数以上に同時に処理すべきリクエストが来た
- コード、設定を変更した
  - コード、設定を変更するとそれまでのコンテナは利用できない

# コールドスタートが起こる条件

簡単に言うと、利用可能なコンテナがない場合に発生

- そもそも1つもコンテナがない状態
- 利用可能な数以上に同時に処理すべきリクエストが来た
- コード、設定を変更した
  - コード、設定を変更するとそれまでのコンテナは利用できない

安定的にリクエストが来ているならば、
コールドスタートはほとんど発生しない

# コールドスタートとの付き合い方

コールドスタートを0にすることは難しい

- それでも一年前とくらべて半分くらいの時間になっている
- サービスチームも日々改善に臨んでいる
- そもそも、基本的にこの領域はお客様でできることは少ない

コールドスタートをまったく許容できないのであれば、そのシステムにとってAWS Lambdaは正解ではない

これらを踏まえて、コールドスタートを頑張って速くするしかない

# コールドスタートを速くする



amazon web services

# コールドスタートを速くする

## コンピューティングリソースを増やす

- コンピューティングリソースの割当を増やすことで初期化処理自体も速くなる

## ランタイムを変える

- 例えばAWS Lambdaに限らず、JVMの起動は遅い
- ただし、一度温まるとコンパイル言語のほうが速い傾向

# コールドスタートを速くする

パッケージサイズを小さくする

- サイズが大きくなるとコールドスタート時のコードのロードおよびZipの展開に時間がかかる
- 不要なコードは減らす
- 依存関係を減らす
  - 不要なモジュールは含めない
  - 特にJavaは肥大しがち
- JavaだとProGuardなどのコード最適化ツールを使って減らすという手もある
  - 他の言語でも同様のものはある

# コールドスタートを速くする

VPCは必要でない限り使用しない

- 使うのはVPC内のリソースにどうしてもアクセスする必要があるときだけ
- VPCアクセスを有効にしているとコールドスタート時に10秒から30秒程度余計に必要になる

同期実行が必要な箇所やコールドスタートを許容できない箇所ではなるだけ使わない

- VPC内のリソースとの通信が必要なのであれば非同期にする
  - RDBMSのデータ同期が必要なのであればDynamoDB StreamsとAWS Lambdaを使って非同期に

# コールドスタートを速くする

Javaの場合は以下も検討

- POJOではなくバイトストリームを使う
  - 内部で利用するJSONシリアライゼーションライブラリは多少時間がかかるので、バイトストリームにしてより軽量なJSONライブラリを使ったり最適化することも可能
    - https://github.com/FasterXML/jackson-jr
    - http://docs.aws.amazon.com/lambda/latest/dg/java-handler-io-type-stream.html
- 匿名クラスをリプレースするようなJava8の機能を利用しない（lambda、メソッド参照、コンストラクタ参照など）

# コールドスタートを速くする

初期化処理をハンドラの外に書くとコールドスタートが遅くなるので遅延ロードを行う

```
import boto3

client = None

def my_handler(event, context):
    global client
    if not client:
        client = boto3.client("s3")

    # process
```



amazon web services

# 遅くなる理由

プログラムの問題

コンピューティングリソースの不足

コールドスタート

アーキテクチャの問題

同時実行数

# アーキテクチャの問題

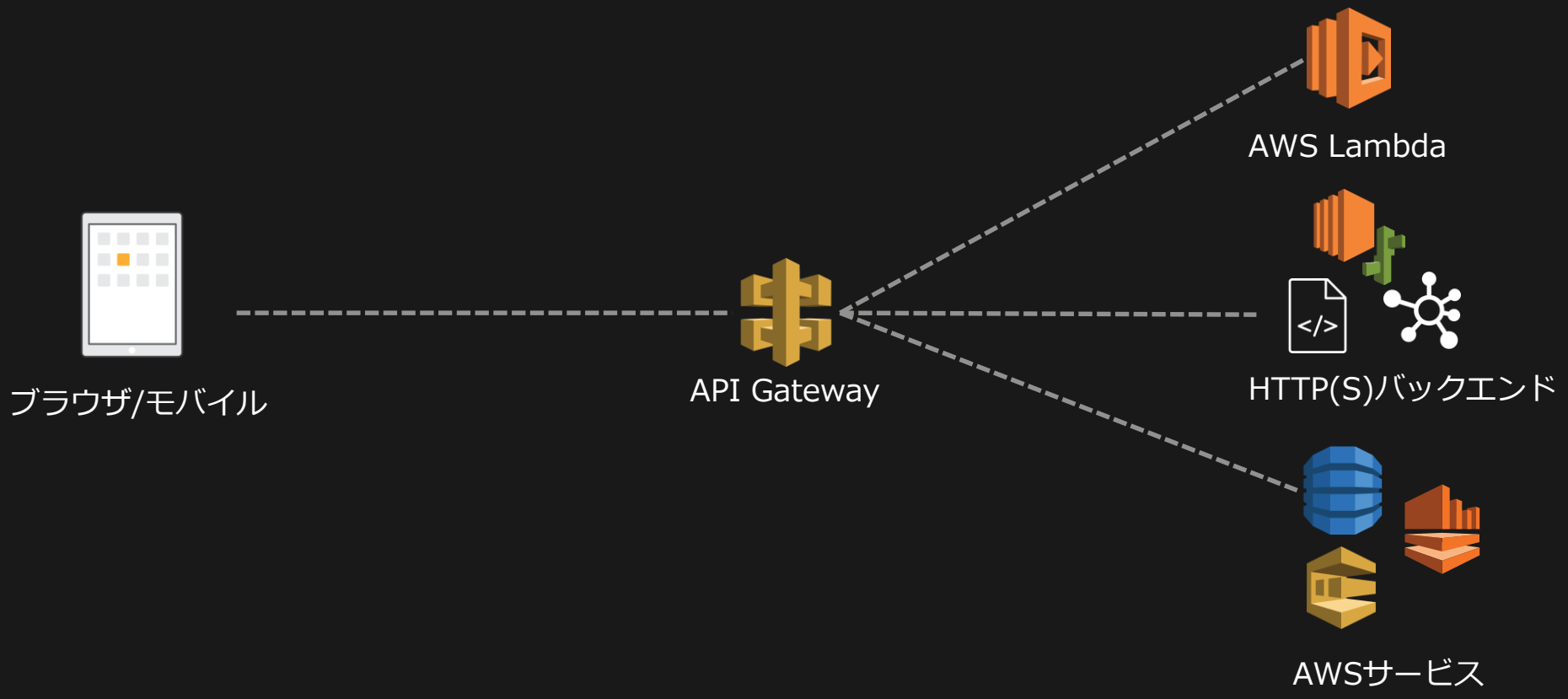
同期でInvokeすると同時実行数の制限に引っかかってつまりがち

- 非同期呼び出しの場合、許可された同時実行数内で順次処理をし、バーストも許容されている
- 同期呼び出しの場合、許可された同時実行数を超えた時点でエラーが返されてしまう
  - 非同期の場合はリトライされる

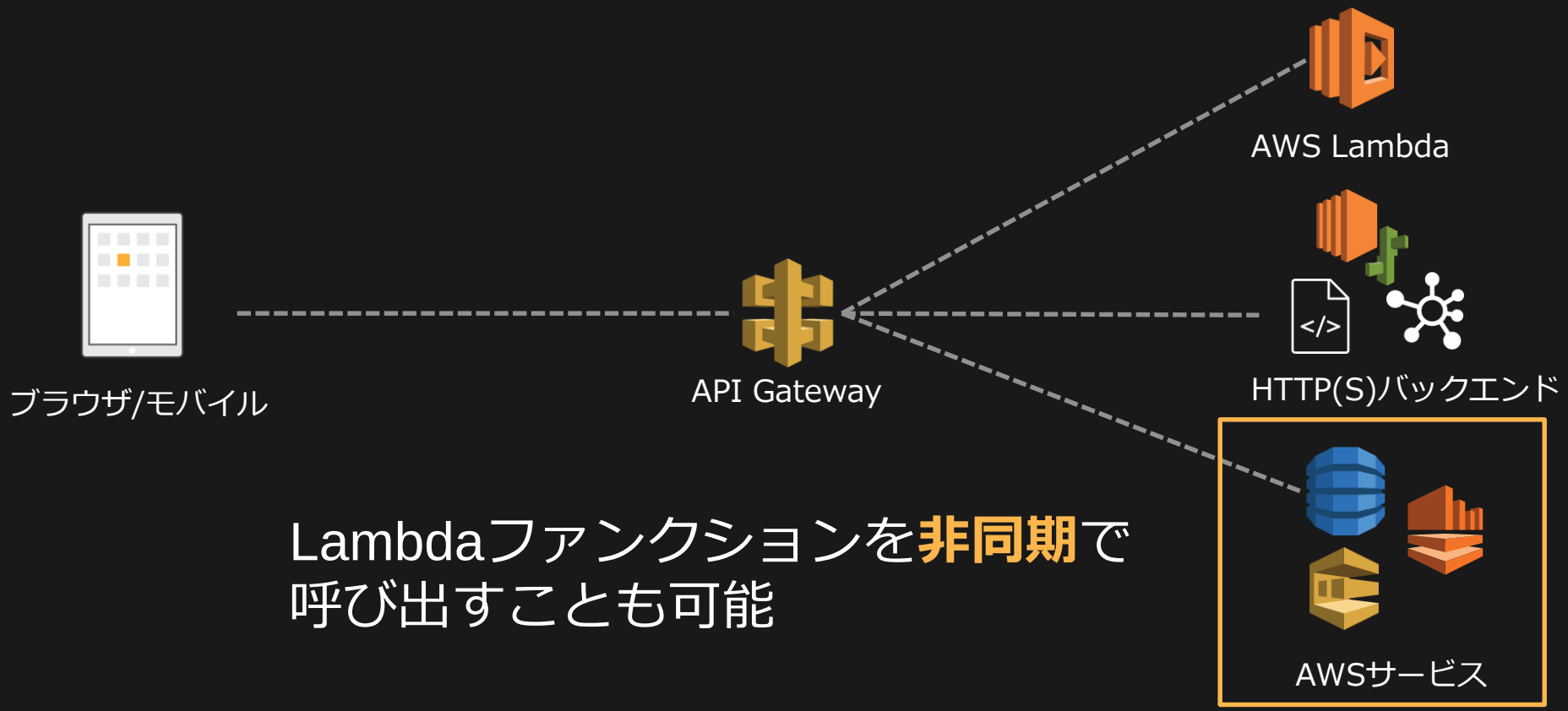
できるだけ非同期でInvokeするのがスケーラビリティの観点ではオススメ

- その処理、本当にレスポンス必要ですか？
- 特にAmazon API Gatewayとの組み合わせの場合、PUT系の処理をAWS Lambdaで直接処理するのではなく、サービスプロキシとして構成してAmazon SQS、Amazon Kinesisに流すなどする

# API Gatewayのサービスプロキシ



# API Gatewayのサービスプロキシ




Lambdaファンクションを**非同期**で
呼び出すことも可能

# アーキテクチャの問題

Think Parallel

- AWS Lambdaの最大限活かすにはいかに並列処理を行うか

1つあたりのイベントを小さくして同時に並列で動かせるようなアーキテクチャにする

- 1回のInvokeでループさせるのではなく、ループ回数分Lambdaファンクションを非同期Invokeする
- 長時間実行のLambdaファンクションが減るので、同時実行数の制限にも引っかかりにくくなる

# 遅くなる理由

プログラムの問題

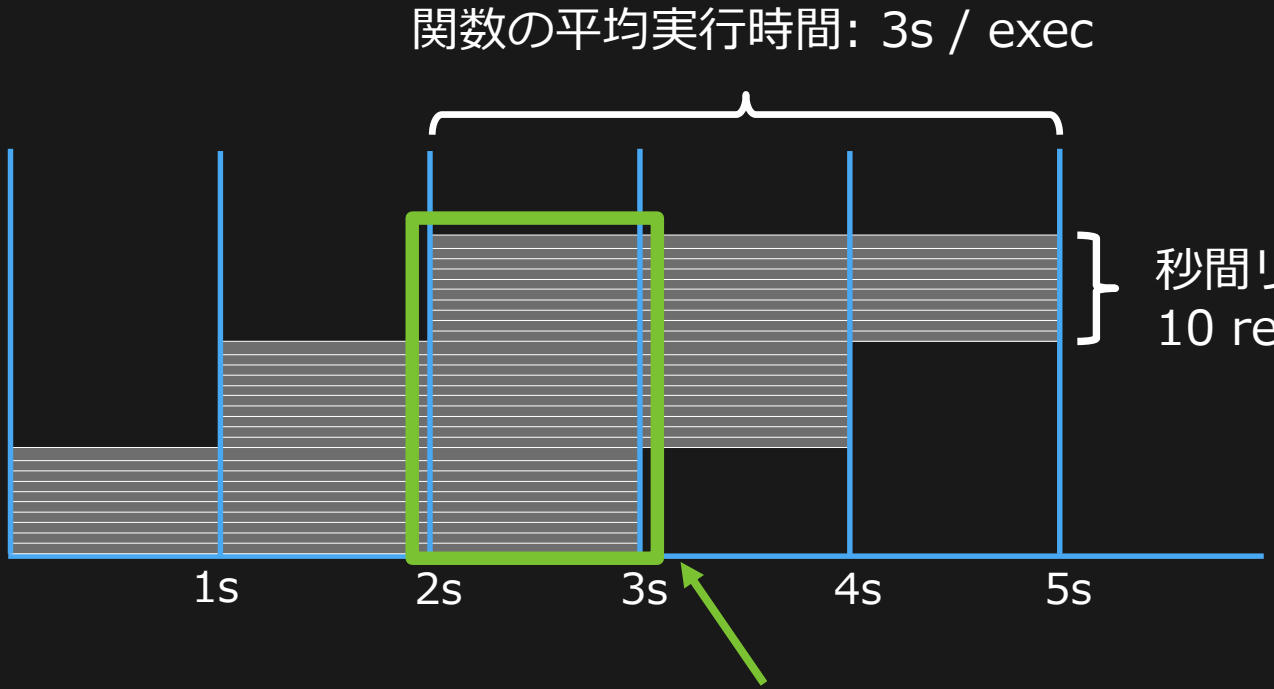
コンピューティングリソースの不足

コールドスタート

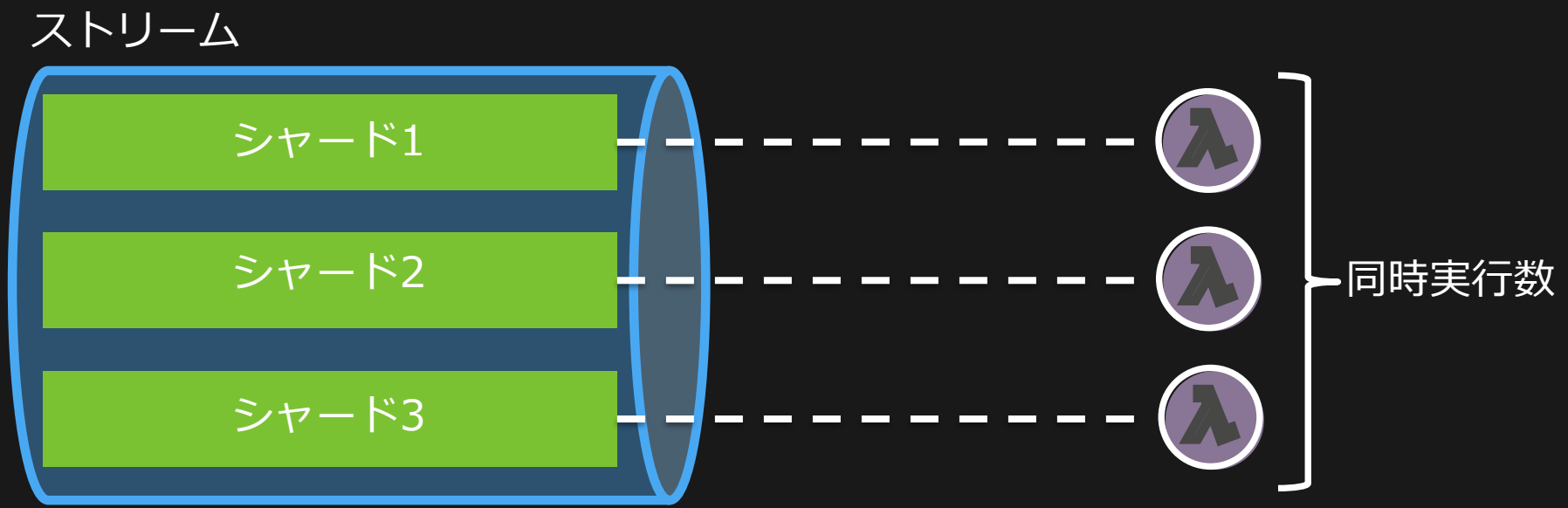
アーキテクチャの問題

同時実行数

# 同時実行数



# 同時実行数 - ストリームベースの場合



# Limit Increaseについて

標準では1000、実績がない状態でいきなり数千とか数万を申請しても通らない

- 実際にThrottleされているかどうかの確認を
- Throttleされているかなどはメトリクスで確認可能
- サービスローンチなどの場合は、余裕をもって担当セールス、SAにご相談を

Limit Increaseしても性能上のボトルネックが解消しないこともある

- ストリーム系の場合、シャード数を増やしたりバッチサイズを調整したりしないと変わらない
- そもそもファンクションの実行に時間がかかっている場合、レイテンシは改善しない

# 他にもまだあるアンチパターン



amazon web services

# AWS LambdaでRDBMS使いがち問題

AWS Lambda + RDBMSがアンチパターンな理由

- コネクション数の問題
  - AWS Lambdaはステートレスなプラットフォームであるため、コネクションプールの実装は難しい
  - AWS Lambdaがスケールする、つまりファンクションのコンテナが大量に生成された場合に、各コンテナからDBへコネクションが張られることになり、耐えられないケースがある
- VPCコールドスタートの問題
  - VPCのコールドスタートが発生する場合、通常のコールドスタートに比べて10秒程度の時間を必要とする

ベストプラクティス

- Amazon DynamoDBを使う
- 何らかの理由でRDBMSとの連携が必要な場合はDynamoDB StreamsとAWS Lambdaを利用して非同期にする

# IP固定したがり問題

Amazon API GatewayやAWS Lambdaから別システムや外部APIにアクセスする際のソースIPアドレスを固定したい

署名や証明書などで担保すべき

- IPアドレスを固定するということはスケーラビリティを捨てることにもつながる

AWS LambdaではVPCを利用してNATインスタンスを使うという方法もなくはないが…

- VPCのコールドスタート問題
- 自前のNATインスタンスの場合、その可用性、信頼性、スケーラビリティを考慮する必要がある

# サーバレスに夢見がち問題

サーバレスであれば全く運用が必要ない、インフラ費用が10分の1になる

- サーバの管理は不要だが運用は必要
- コスト効率が高いため、サーバを並べて同様のことを実装するよりは安くなる可能性が高いが、リクエスト数が多い場合などはそれなりの費用になる
- インフラ費用だけでなく、トータルコストで考える必要がある
- 複雑なことをやろうとすると、設計・開発コストがあがる可能性も大きい

シンプルに使うべきものはシンプルに使いましょう

# 監視しなくていいと思ってる問題

Serverless != Monitorless

処理の異常を検知して対応するのはユーザの仕事

- 適切にログ出力を行い、適切にモニタする

ベストプラクティス

- CloudWatchのメトリクスを利用（Errors, Throttles）
- CloudWatchのカスタムメトリクス
- CloudWatch Logsへのログ出力とアラーム設定

# その他

通常の他のサービスと同様に障害発生を前提として実装をする

- リトライ
- Dead Letter Queueの活用（非同期の場合）

冪等性はお客様のコードで確保する必要がある

- AWS Lambdaで保証しているのは最低1回実行することであり1回しか実行しないことではない
- 同一イベントで同一Lambdaファンクションが2回起動されることがまれに発生する
- Amazon DynamoDBを利用するなどして冪等性を担保する実装を行うこと

# Wrap-up

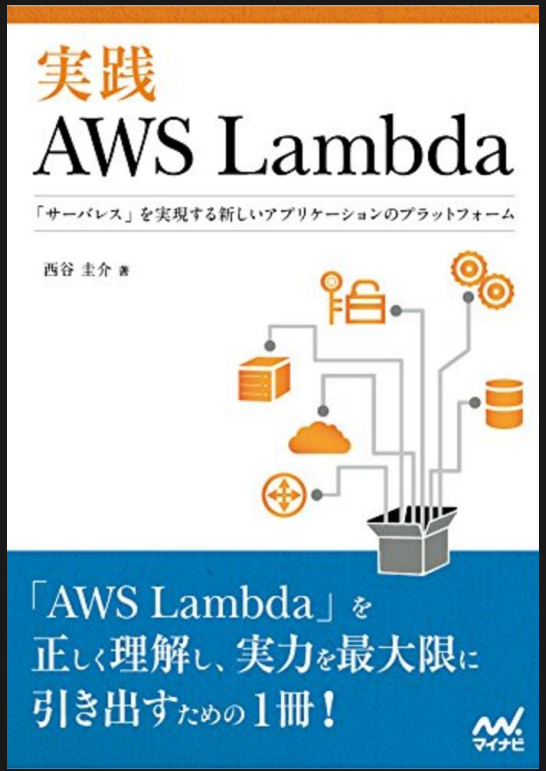
AWS Lambdaによるシステムの性能が出ない理由

- プログラムの問題
- コンピューティングリソースの不足
- コールドスタート
- アーキテクチャの問題
- 同時実行数

それぞれに対して適切なアプローチを行う

典型的なアンチパターンはできるだけ避けましょう！

# AWS Lambdaの本が出ます




2017年6月9日マイナビ出版より出版予定

3,240円（税込）

Amazonで予約受け付け中
http://amzn.asia/ew2WWPm

AWS Summit 本会場のExpoエリア奥、マイナビ出版様ブースで先行販売中

- 消費税分安くなってます
- 電子版ダウンロード権ついてきます



amazon web services

# Thank You!



amazon web services

# Don’t Forget Evaluations!



amazon web services