AWS-46

AWS Lambda Performance Tuning Deep Dive
～本当に知りたいのは”ここ”だった ～

下川 賢介
技術統括本部 ソリューションアーキテクト シニア サーバーレス スペシャリスト
アマゾン ウェブ サービス ジャパン合同会社

下川 賢介

Kensuke Shimokawa

シニア サーバーレス スペシャリスト
ソリューションアーキテクト
技術統括本部
アマゾン ウェブ サービス ジャパン合同会社

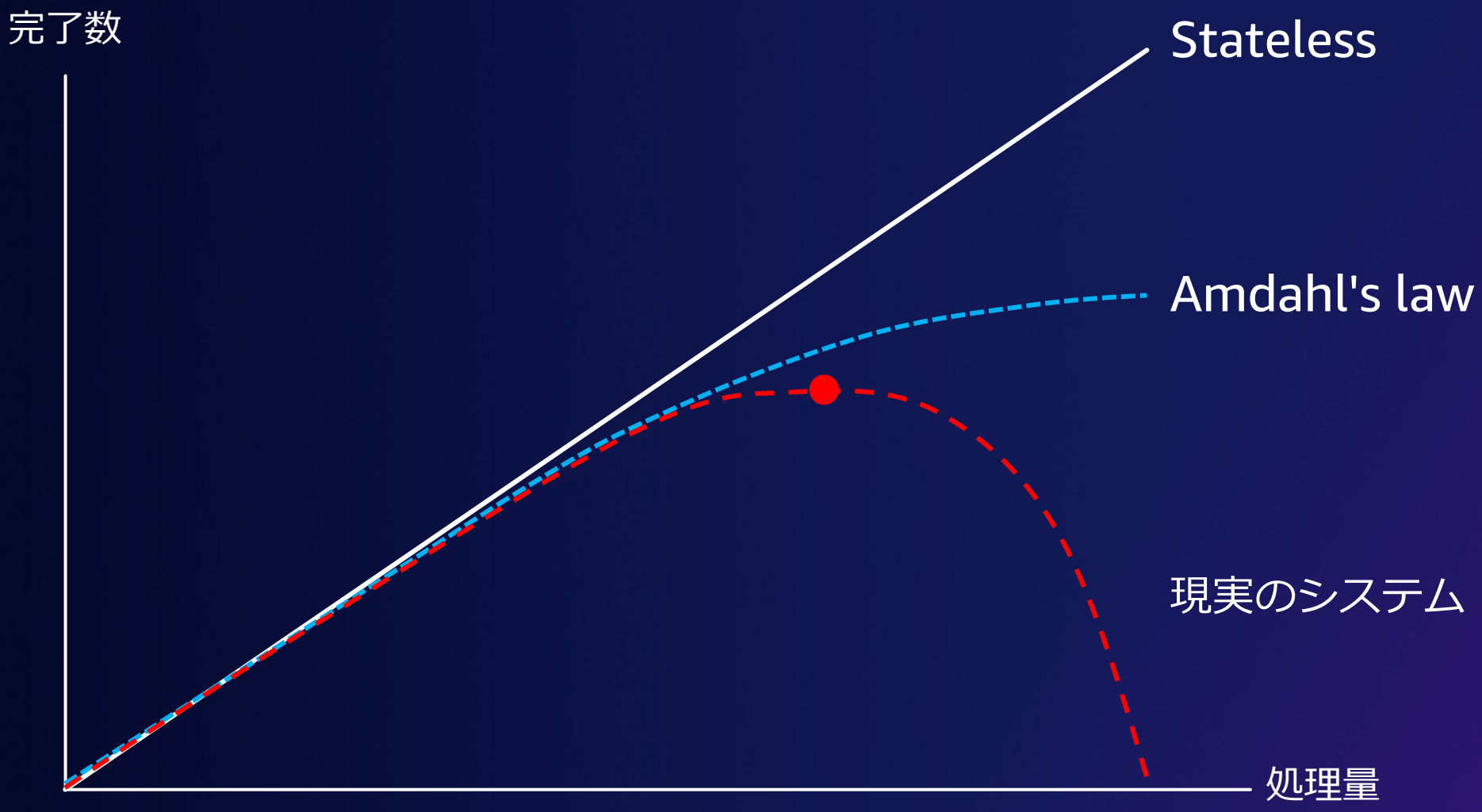
_kensh

Agenda

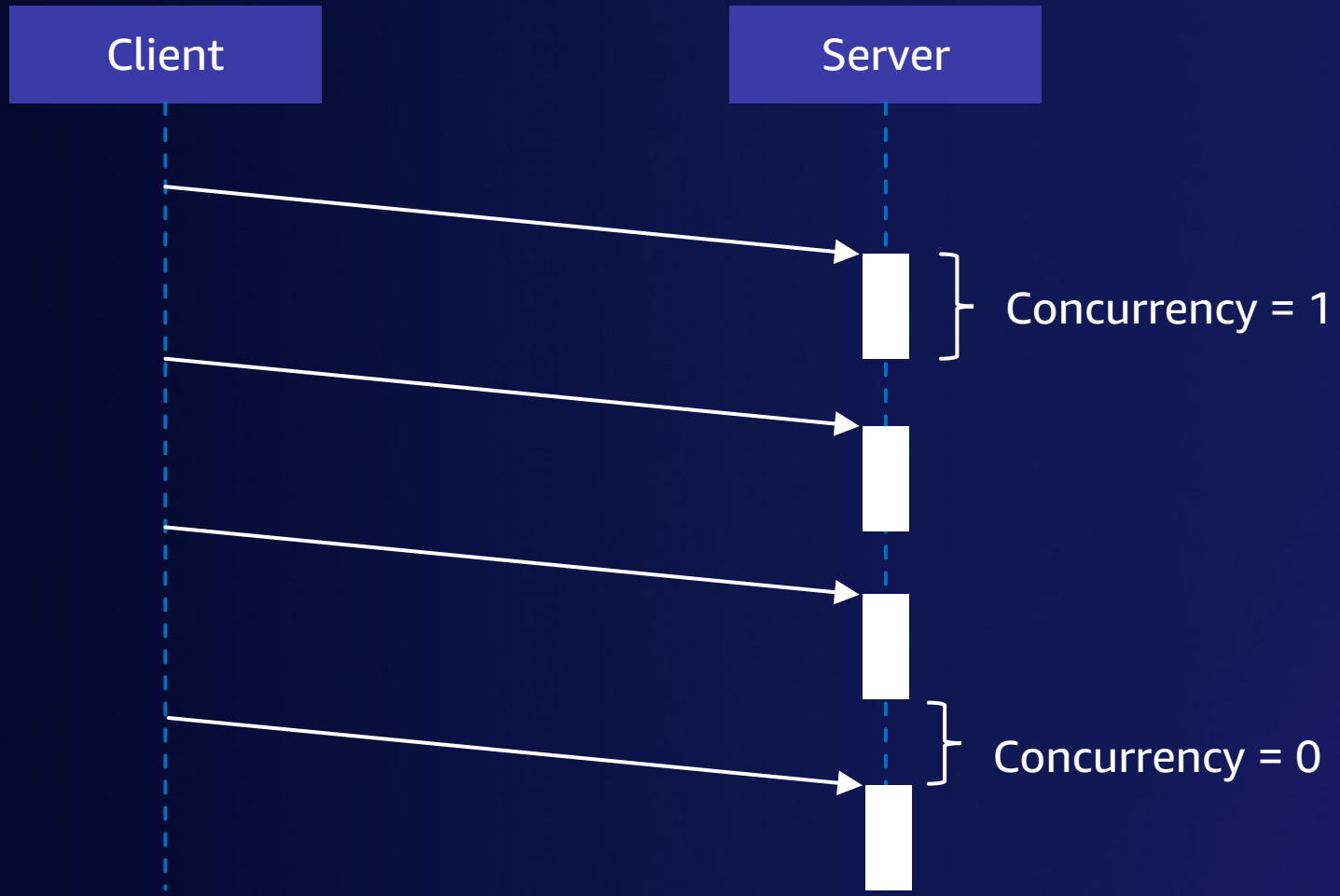
- AWS Lambda の スケーリング
- Tuning の観測
- Lambda 実装プラクティス
- CPU の上手な利用方法

AWS Lambda のスケーリング

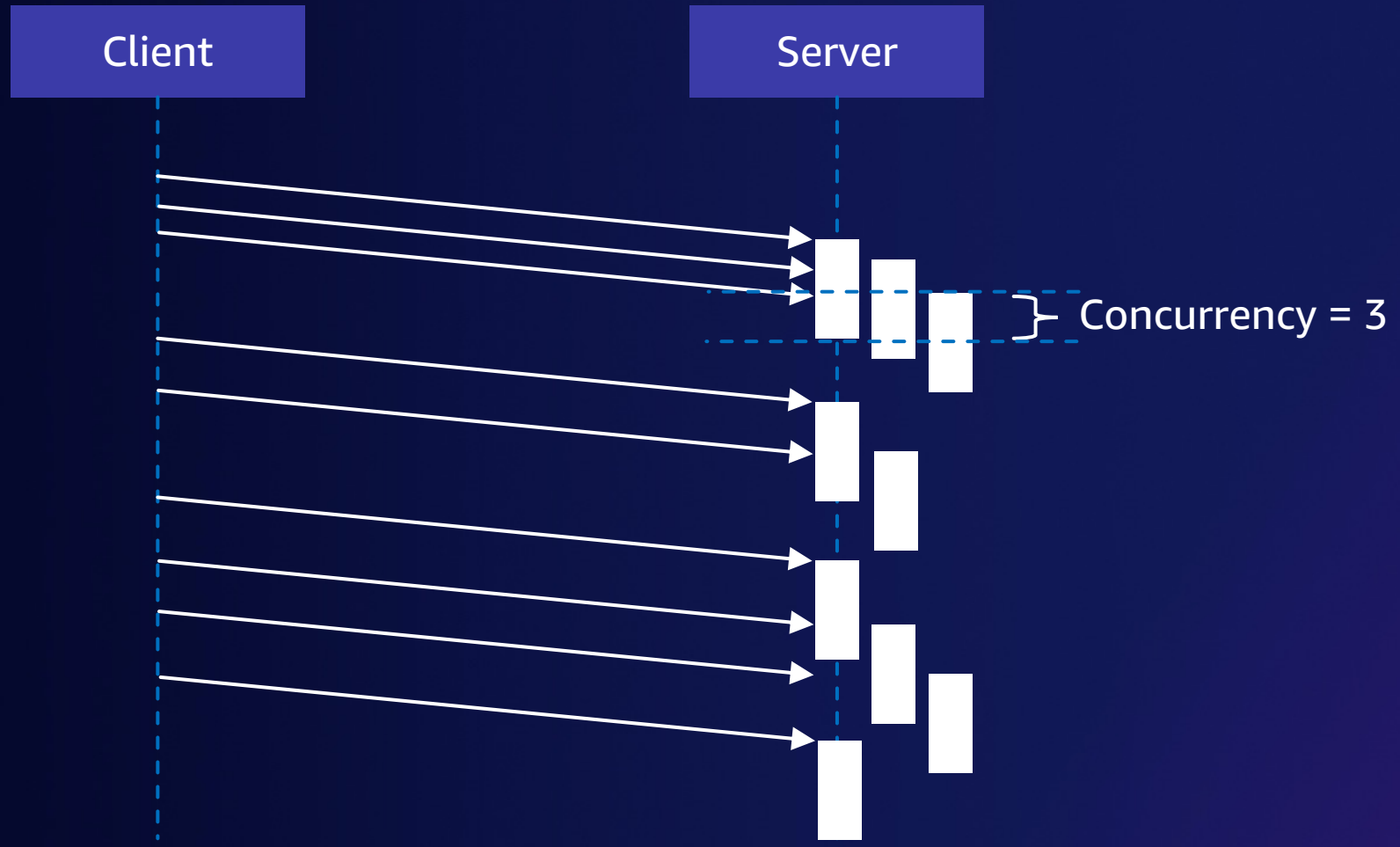
Stateless であること



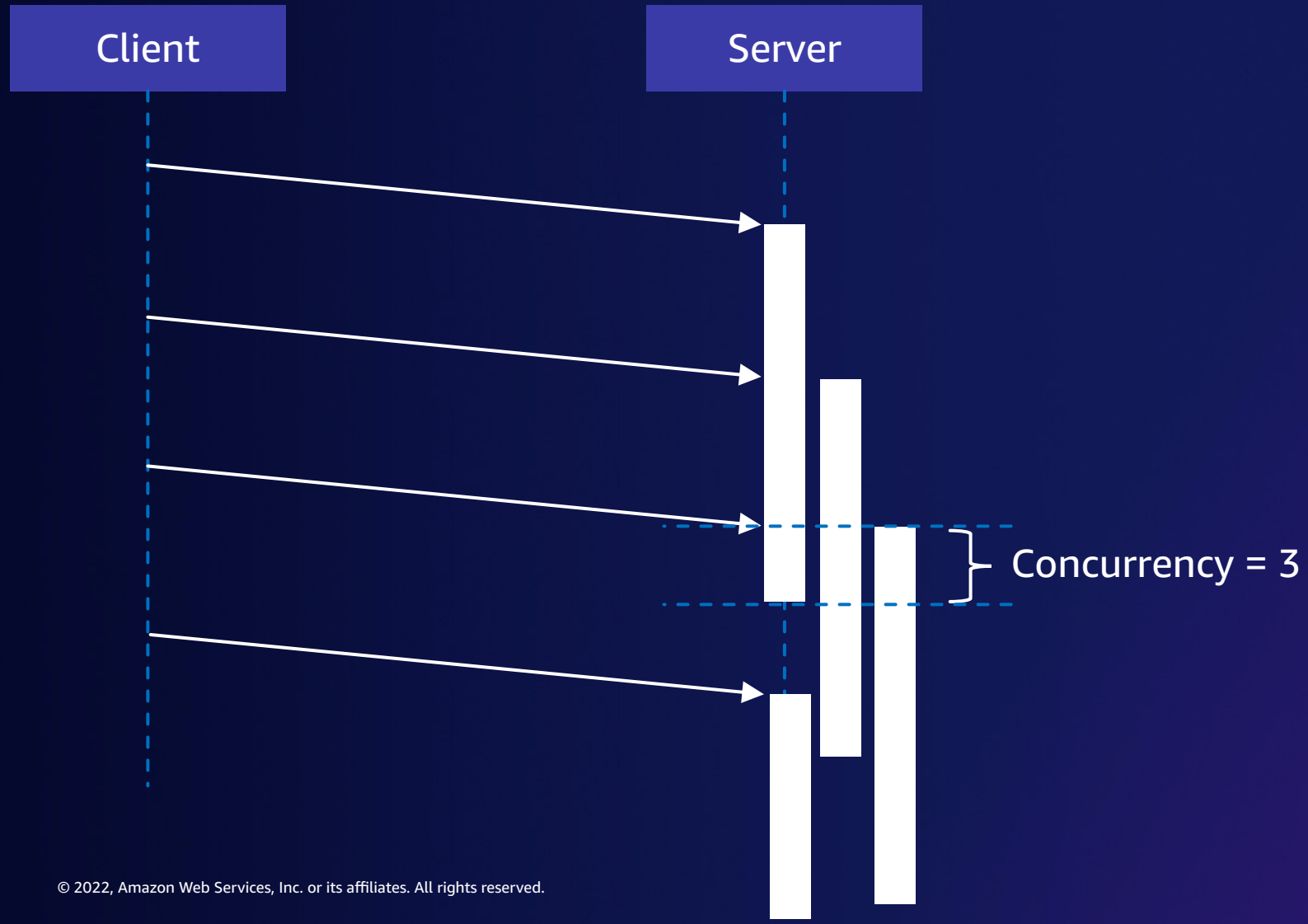
同時実行（Concurrency）



Concurrency増加：リクエストレート(rps)増加



Concurrency増加：実行時間(duration)増加



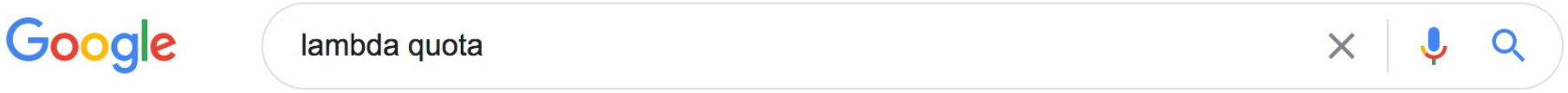
Little’s Law (リトルの法則)

ここを下げるか　　ここを下げる

Concurrency = rps x duration

(同時実行数)　　(リクエストレート)　　(実行時間)

AWS Lambda の Concurrency Quota



AWS Lambda limits

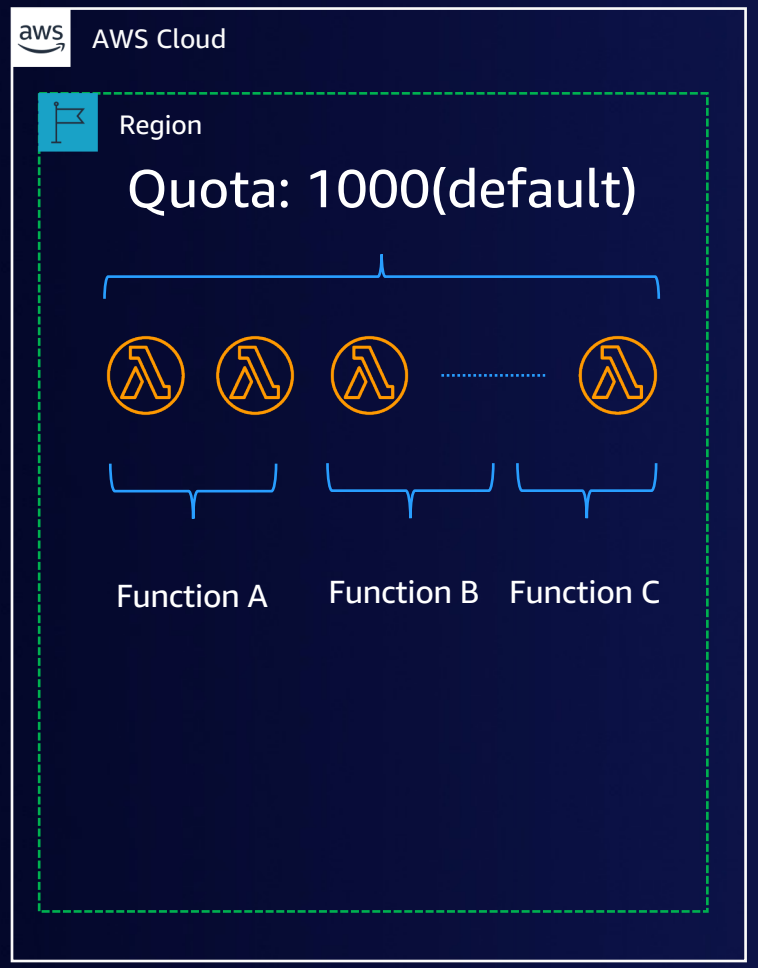
PDF | Kindle | RSS

AWS Lambda limits the amount of compute and storage resources that you can use to run and store functions. The following limits apply per-region and can be increased. To request an increase, use the Support Center console.

Resource	Default limit
Concurrent executions	1,000
Function and layer storage	75 GB
Elastic network interfaces per VPC	250

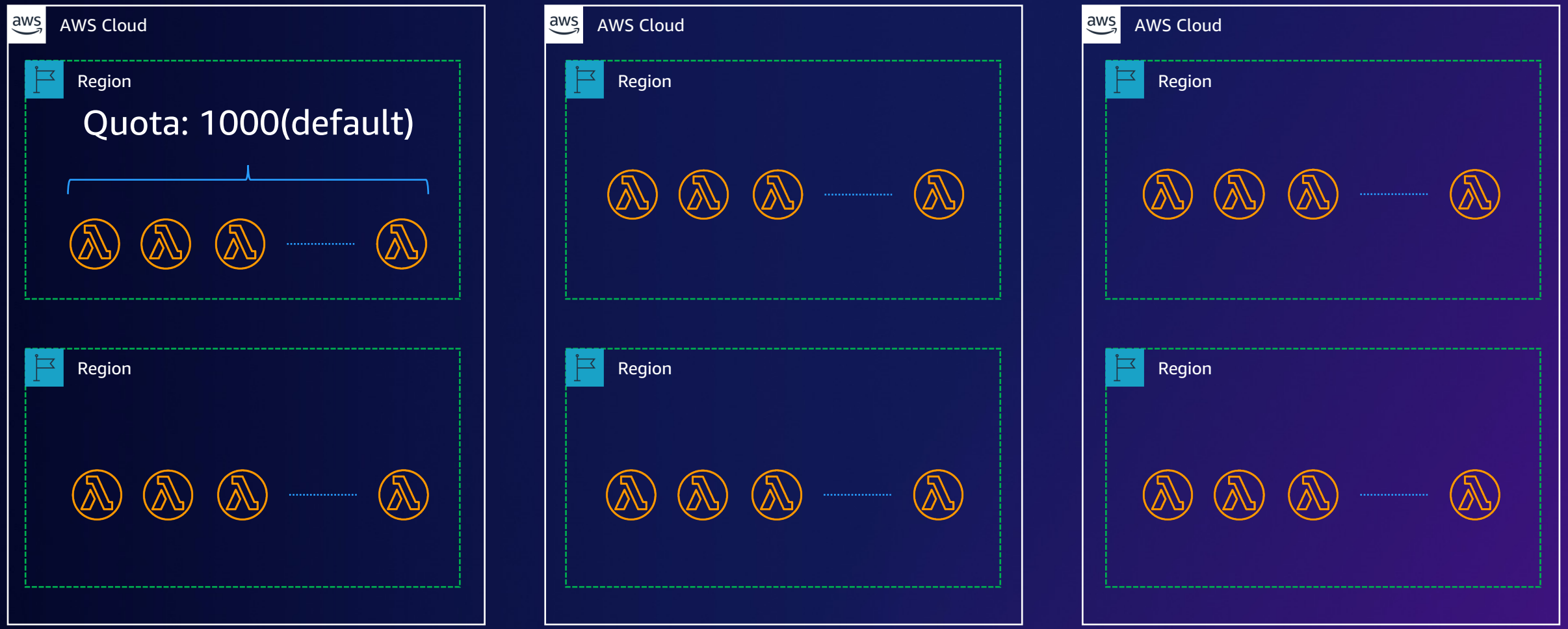
アカウント、リージョン毎

同時実行数はアカウント、リージョン単位で共有



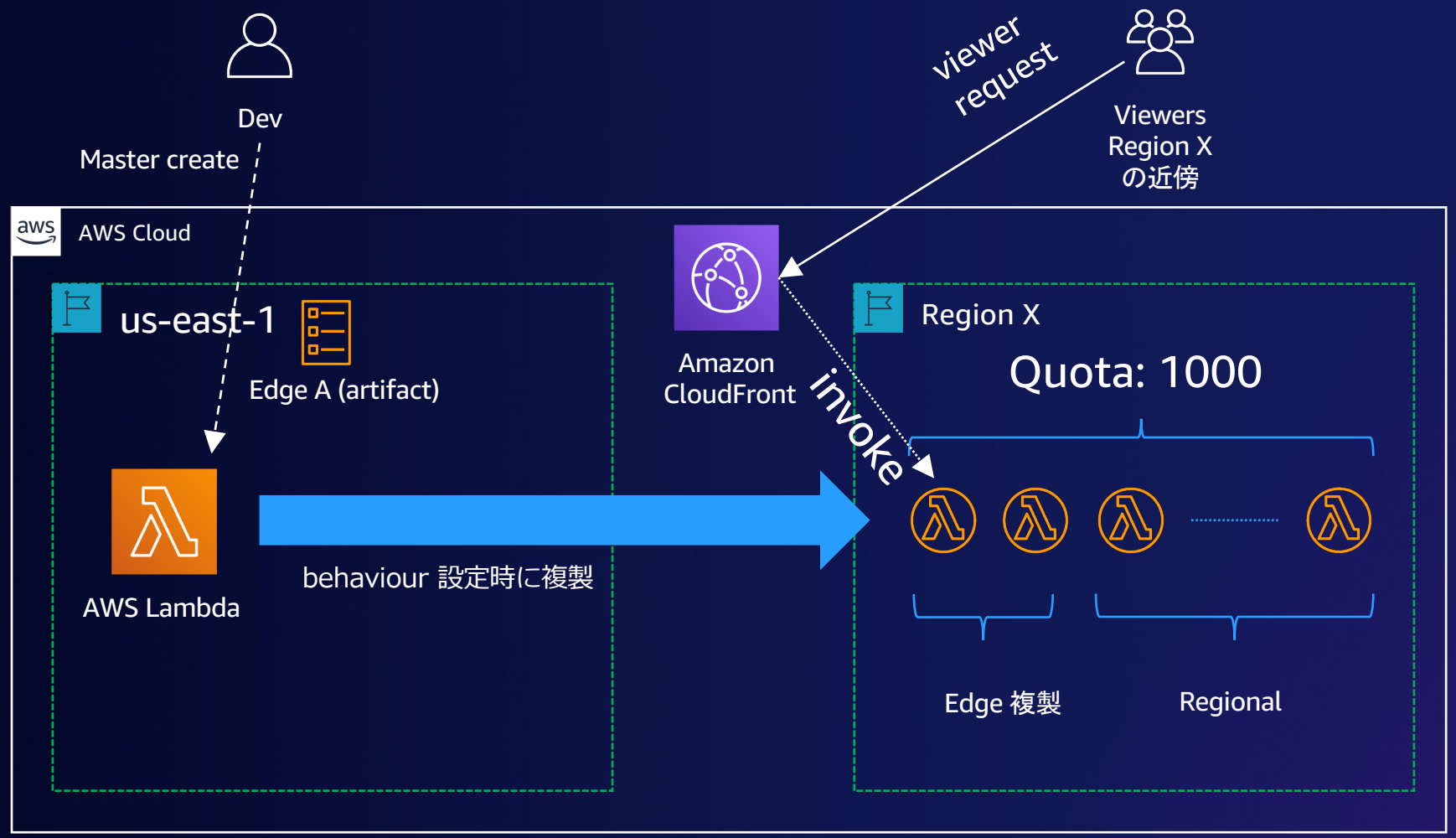
- 各Lambda関数が、Quotaで設定された同時実行数を共有
- Lambda関数の Concurrency の合計が 、同時実行数の制限に達した場合に、Throttlingが発生

同時実行数はアカウント、リージョン単位で共有



Tokyo, Osaka で別のQuotaを持つため、負荷の分散に利用可能
プロダクトごとに AWS アカウントを分離するのも有効

Lambda@Edge の同時実行について

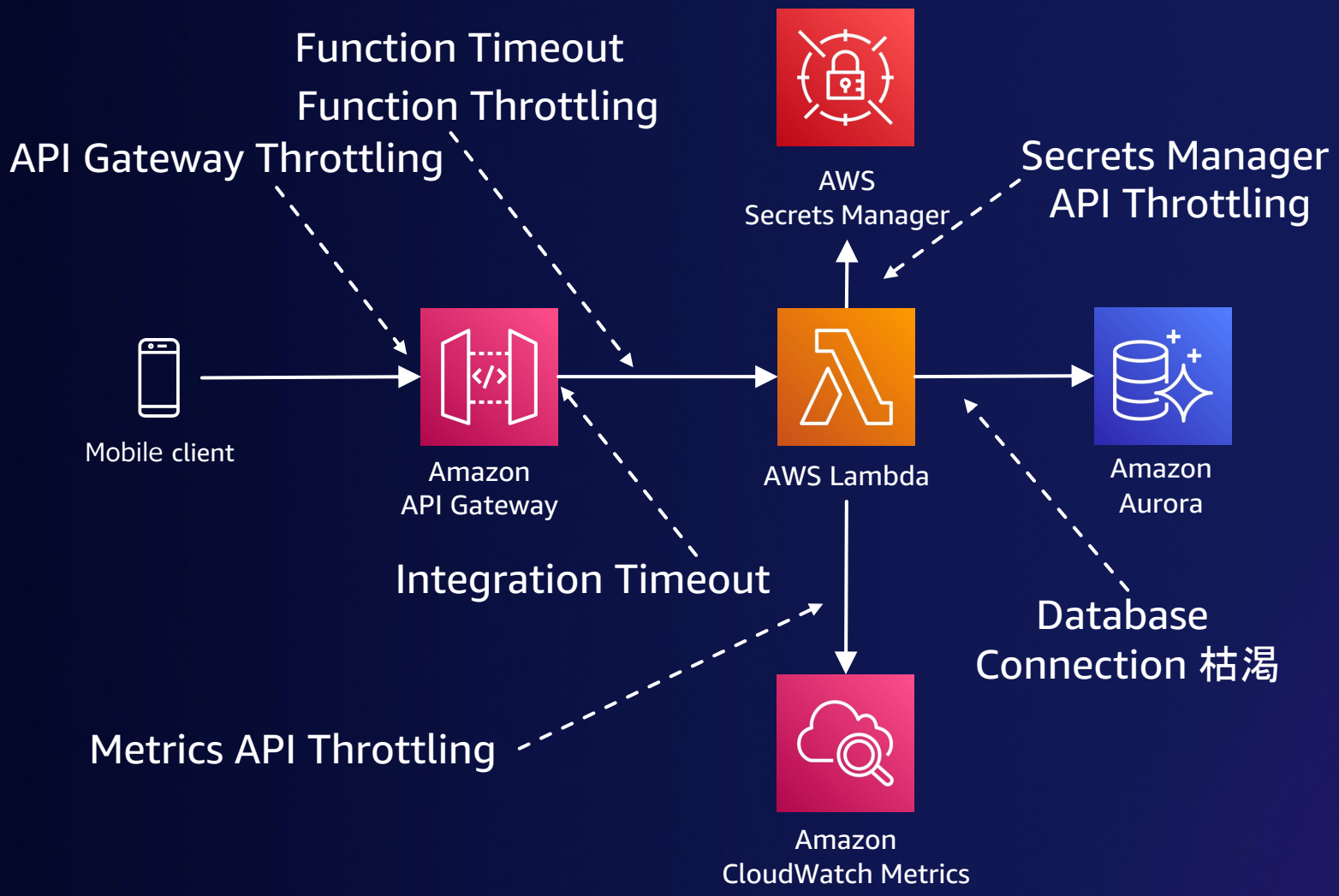


- Lambda@Edgeは、us-east-1のマスター関数を他のRegionに複製
- CloudFrontへのアクセスにより、適切な近傍 RegionでLambda関数の複製を実行
- 複製先でのConcurrency QuotaはRegional Lambdaと共有
- 軽量な処理は CloudFront Functions にオフロードも考慮

https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/lambda-edge-how-it-works.html
https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/cloudfront-limits.html#limits-lambda-at-edge

Tuning の 観測

同期的な Serverless API



[負荷観測ポイント]

- DBへの負荷により、接続性やQuery応答性、Lambdaの実行時間に影響
- Lambda関数の実行時間が延びることにより、コストや統合サービスも遅延タイムアウトを誘発
  - API Gateway 統合タイムアウト = 29s(default)
- イベント駆動バッチとして処理している場合、Lambdaの15mタイムアウトを考慮
- Lambda関数から呼び出すサービスのThrottling
  - Secrets Manager
  - CloudWatch Metrics

Throttling とは

What is Throttling?

Throttling は、リソースとダウンストリームを保護することが目的
Lambda 関数は受信トラフィックに合わせて自動的にスケーリングするが、さまざまな理由で関数が Throttling 状態になる

https://aws.amazon.com/premiumsupport/knowledge-center/lambda-troubleshoot-throttling

Lambda 関数の 同時実行 (Concurrency) 制御

- Concurrency は、共有プール (Shared Pool)
- 関数単位の Concurrency の設定が可能
  - Reserved Concurrency
    - Concurrency の確保
    - Concurrency の最大値の定義としても機能

Throttling トラブルシュートパス

https://aws.amazon.com/jp/premiumsupport/knowledge-center/lambda-troubleshoot-throttling

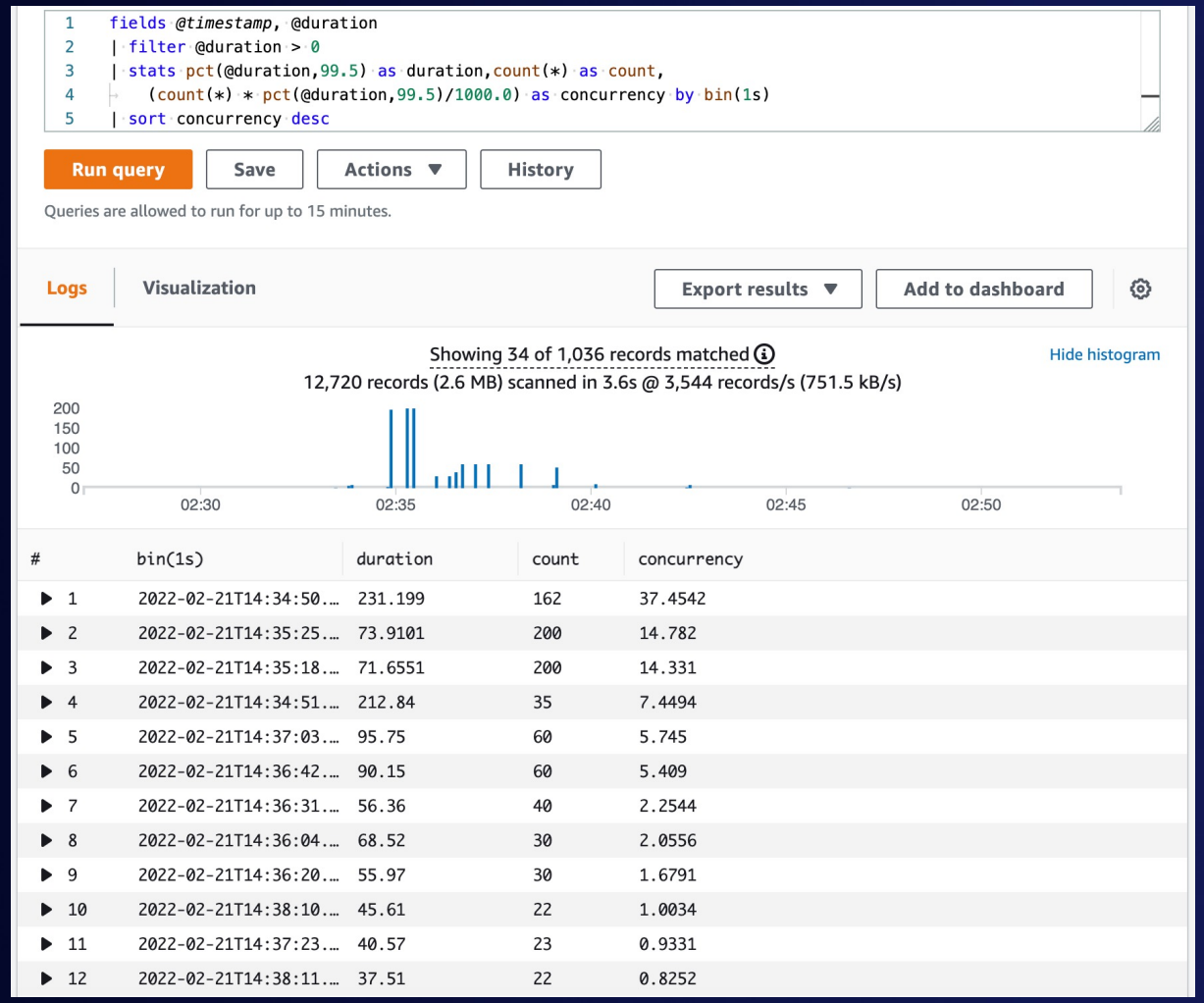
Concurrency を観測する

Amazon CloudWatch Metrics

AWS Lambda の標準 Metrics の分解能は 1分、これより精度の高い 1秒 単位の パフォーマンスチューニングが必要になる場合どうすればよいか

※ Custom Metrics では 1秒 単位の分解能

CloudWatch Logs Insights：ログデータの分析



#	bin(1s)	duration	count	concurrency
1	2022-02-21T14:34:50.…	231.199	162	37.4542
2	2022-02-21T14:35:25.…	73.9101	200	14.782
3	2022-02-21T14:35:18.…	71.6551	200	14.331
4	2022-02-21T14:34:51.…	212.84	35	7.4494
5	2022-02-21T14:37:03.…	95.75	60	5.745
6	2022-02-21T14:36:42.…	90.15	60	5.409
7	2022-02-21T14:36:31.…	56.36	40	2.2544
8	2022-02-21T14:36:04.…	68.52	30	2.0556
9	2022-02-21T14:36:20.…	55.97	30	1.6791
10	2022-02-21T14:38:10.…	45.61	22	1.0034
11	2022-02-21T14:37:23.…	40.57	23	0.9331
12	2022-02-21T14:38:11.…	37.51	22	0.8252

CloudWatch Logs Insights を使用すると、Amazon CloudWatch Logs のログデータをインタラクティブに検索し分析可能

特定の期間でのピーク Concurrency を 1s の分解能で抽出できる

※ Map/Reduce Cluster等の解析エンジンは不要、スケーラブルに分析

Amazon CloudWatch Logs Insights

https://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/logs/AnalyzingLogData.html

CloudWatch Logs Insights：ログデータの分析

1秒の分解能

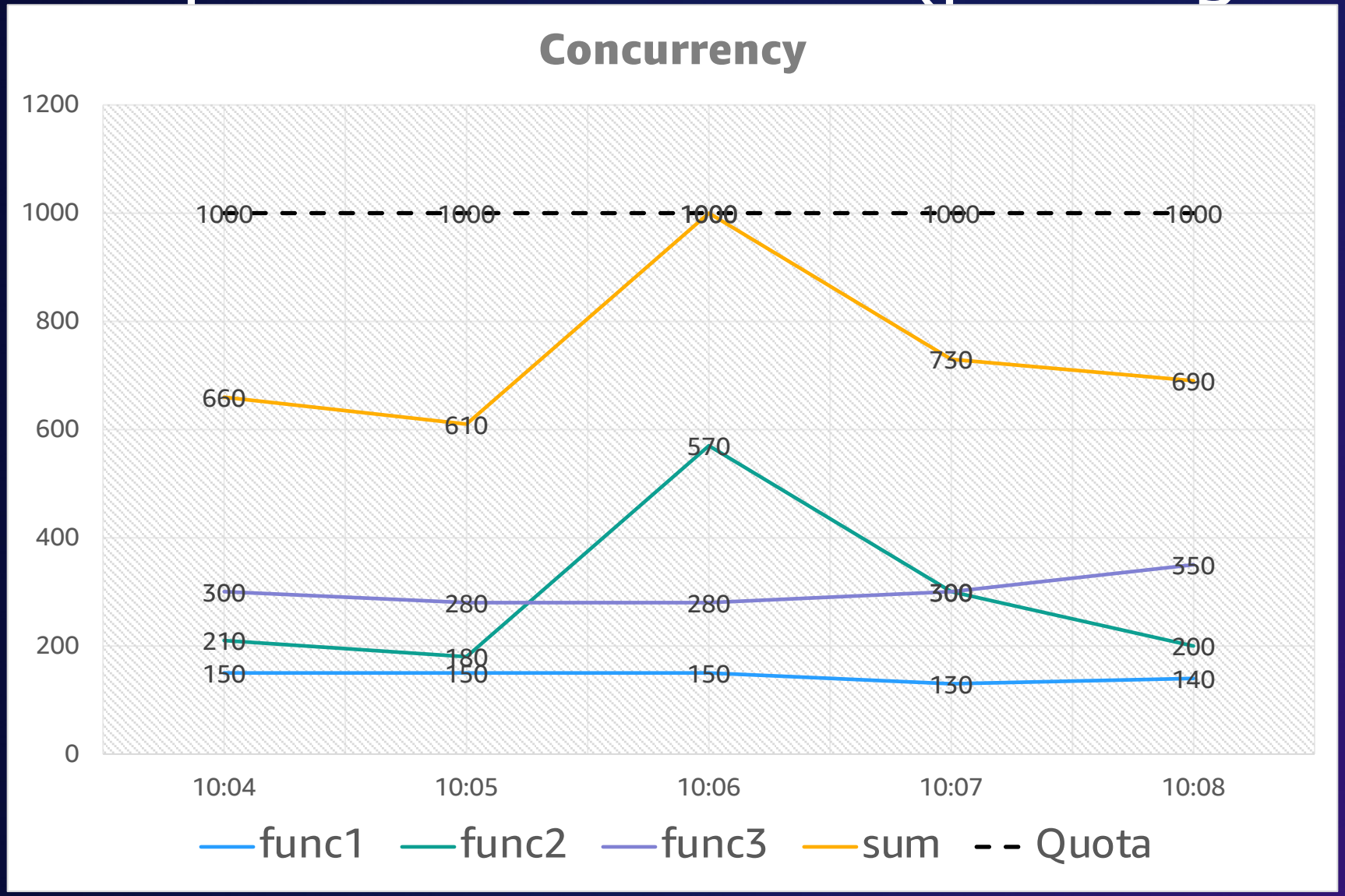
```
fields @timestamp, @duration
| filter @duration > 0
| stats pct(@duration,99.5) as duration, count(*) as count,
        (count(*) * pct(@duration,99.5)/1000.0) as concurrency by bin(1s)
| sort concurrency desc
```

※ 分析要件に応じて、パーセンタイル(pct) と平均(avg)を利用可能

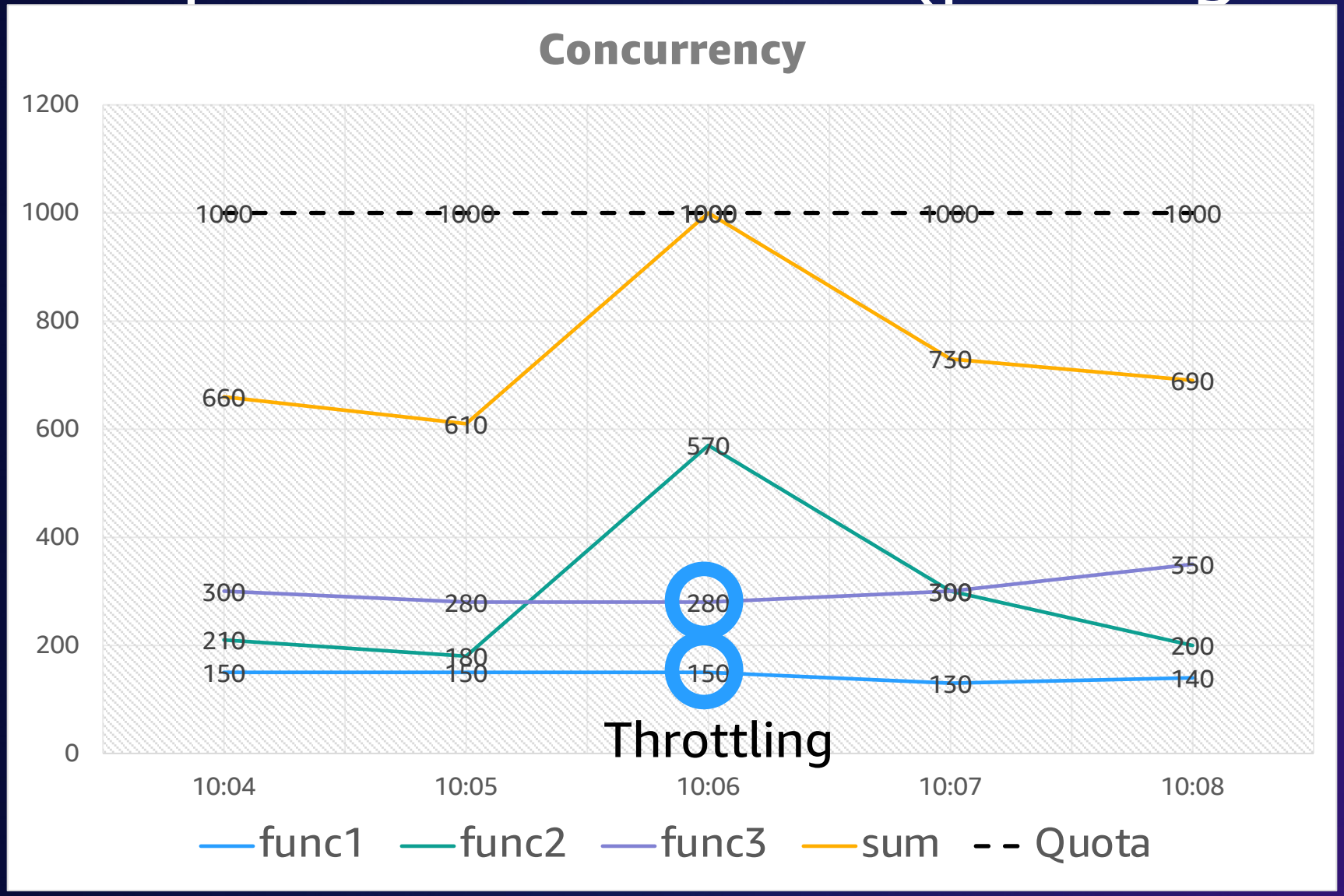
Concurrency = rps x duration

(同時実行数)　(リクエストレート)　(実行時間)

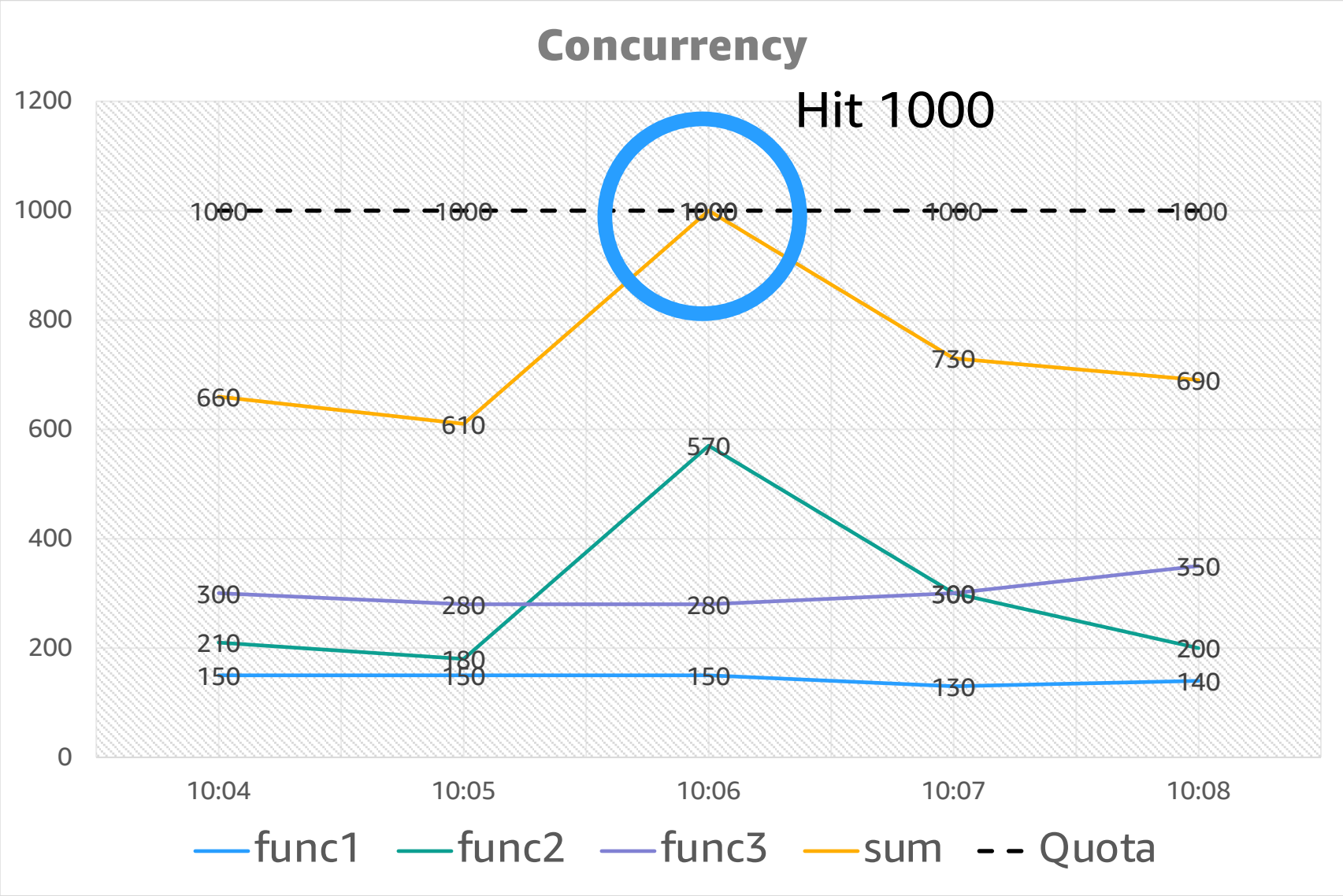
どの関数がSpikeしているか確認(per Region)



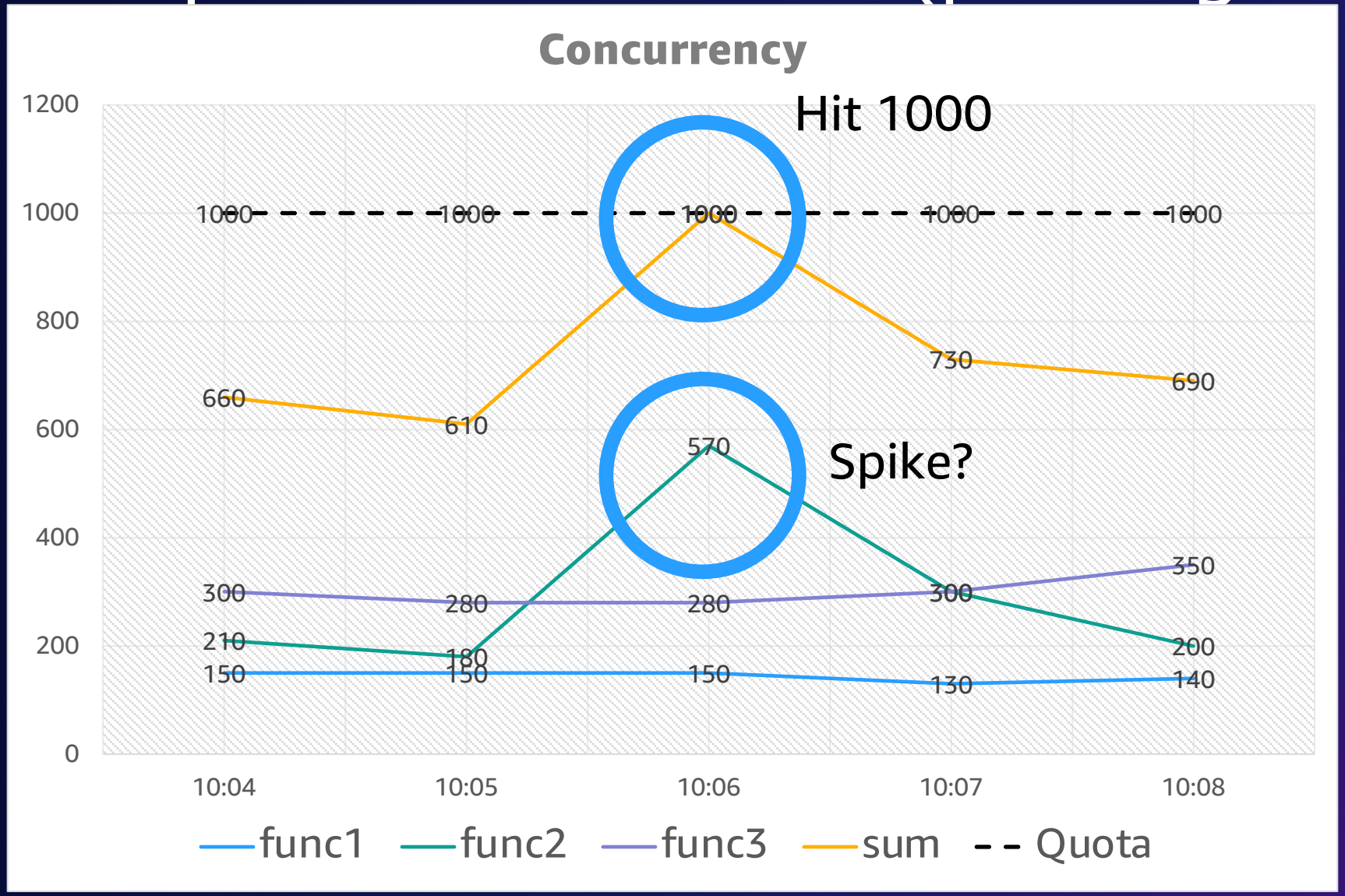
どの関数がSpikeしているか確認(per Region)



どの関数がSpikeしているか確認(per Region)



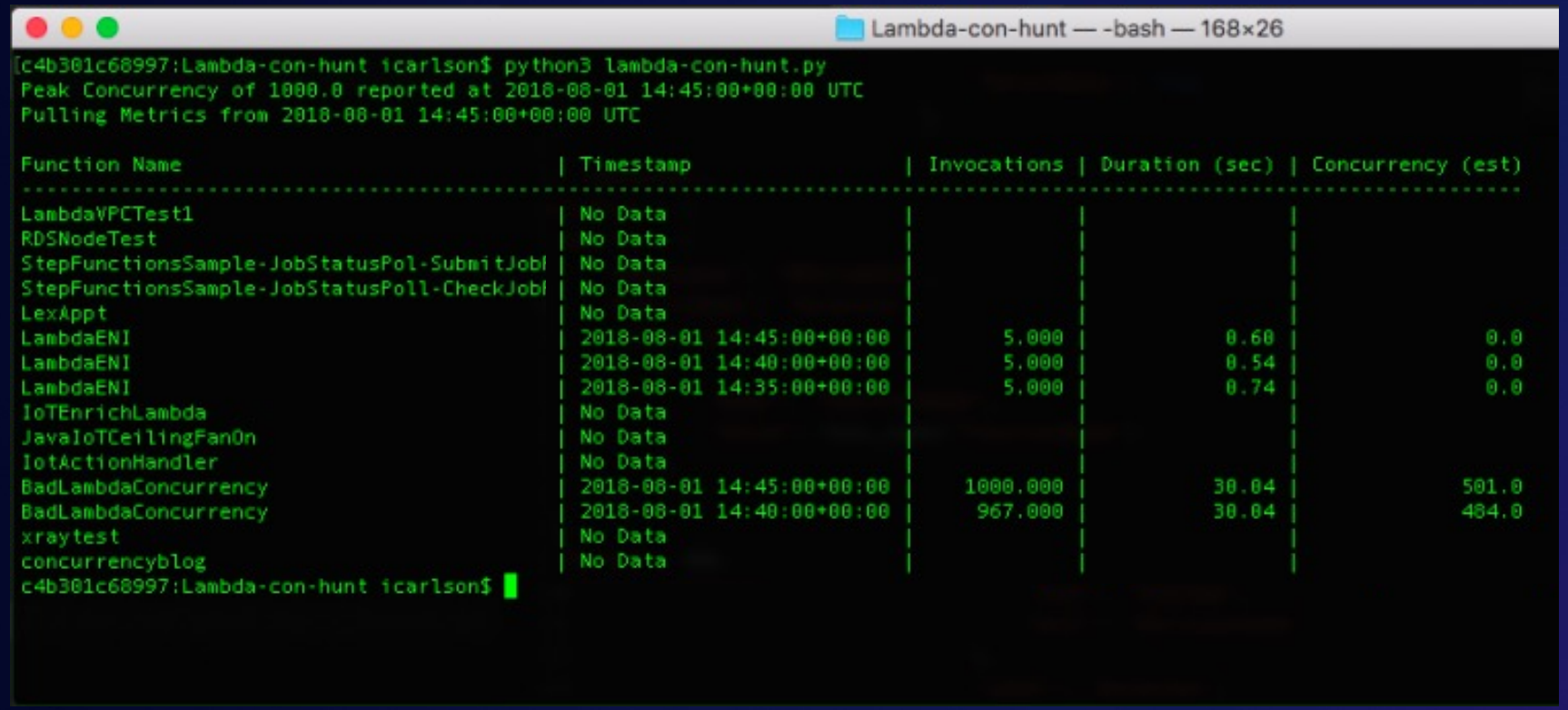
どの関数がSpikeしているか確認(per Region)



Throttling の対処

- Duration Metrics に着目
  - Duration が一時的に伸びている場合、同時実行数を消費することも
    - 外部リソースへのアクセスが正常かを確認
      - 外部リソースへのアクセスに対して Exponential Backoff が実装されている場合
        - DynamoDB の キャパシティ枯渇時(On-Demand の検討も)
        - 関数内の呼び出し先 On-prem HTTP サーバーのダウン時 (Cacheの検討も)
    - コードチューニング
      - AWS Lambda 関数を使用する際のベストプラクティス
        - https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/best-practices.html

Lambda Concurrency Hunt (Tips)



過去7日間のMetricsに対して同時実行数が最も高い期間を見つけ、そのSpikeの前の6分間の情報を出力する。

- 関数の呼び出し回数
- 平均処理時間
- 同時実行数

※ Lambdaの1 分間隔Metricsにおいて、15 日間は1 分の分解能を持つ。以降も使用可能だが、5 分に集約された分解能となる
https://aws.amazon.com/jp/cloudwatch/faqs/

$ curl https://raw.githubusercontent.com/aws-samples/aws-lambda-concurrency-hunt/master/lambda-con-hunt.py -o lambda-con-hunt.py
$ python3 lambda-con-hunt.py

Lambda 実装プラクティス

効率的な関数コード

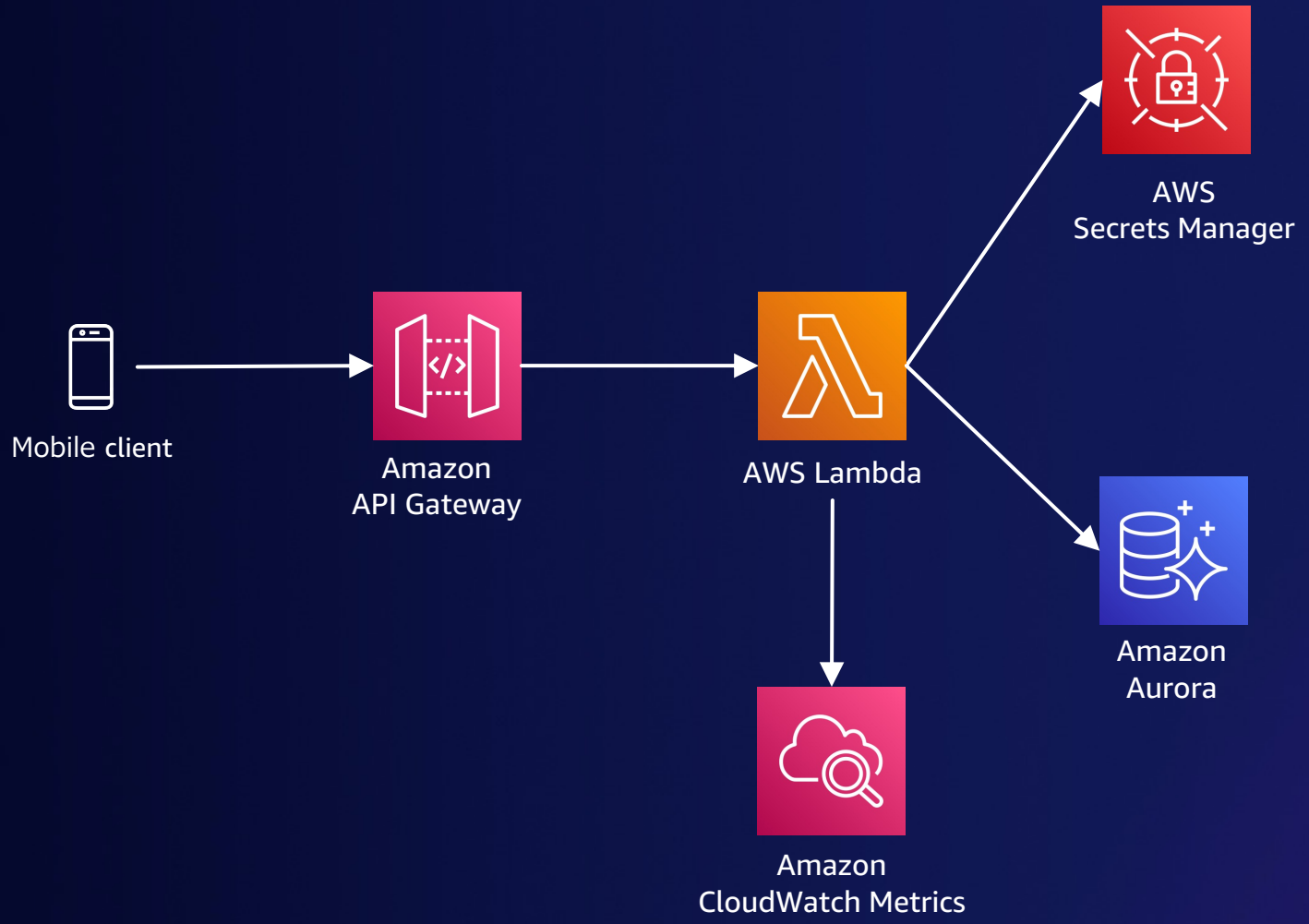
- FAT/monolithic な関数実装を避ける
- 依存パッケージを最小化
  - デプロイメントサイズを小さくしダウンロード時間の短縮
  - 依存モジュールのロード時間を短く
- パッケージマネージャによる TreeShaking
  - e.g.) Node.jsの場合、webpackを利用して、ES module の静的解析を実行

揮発性(Ephemeral) を意識

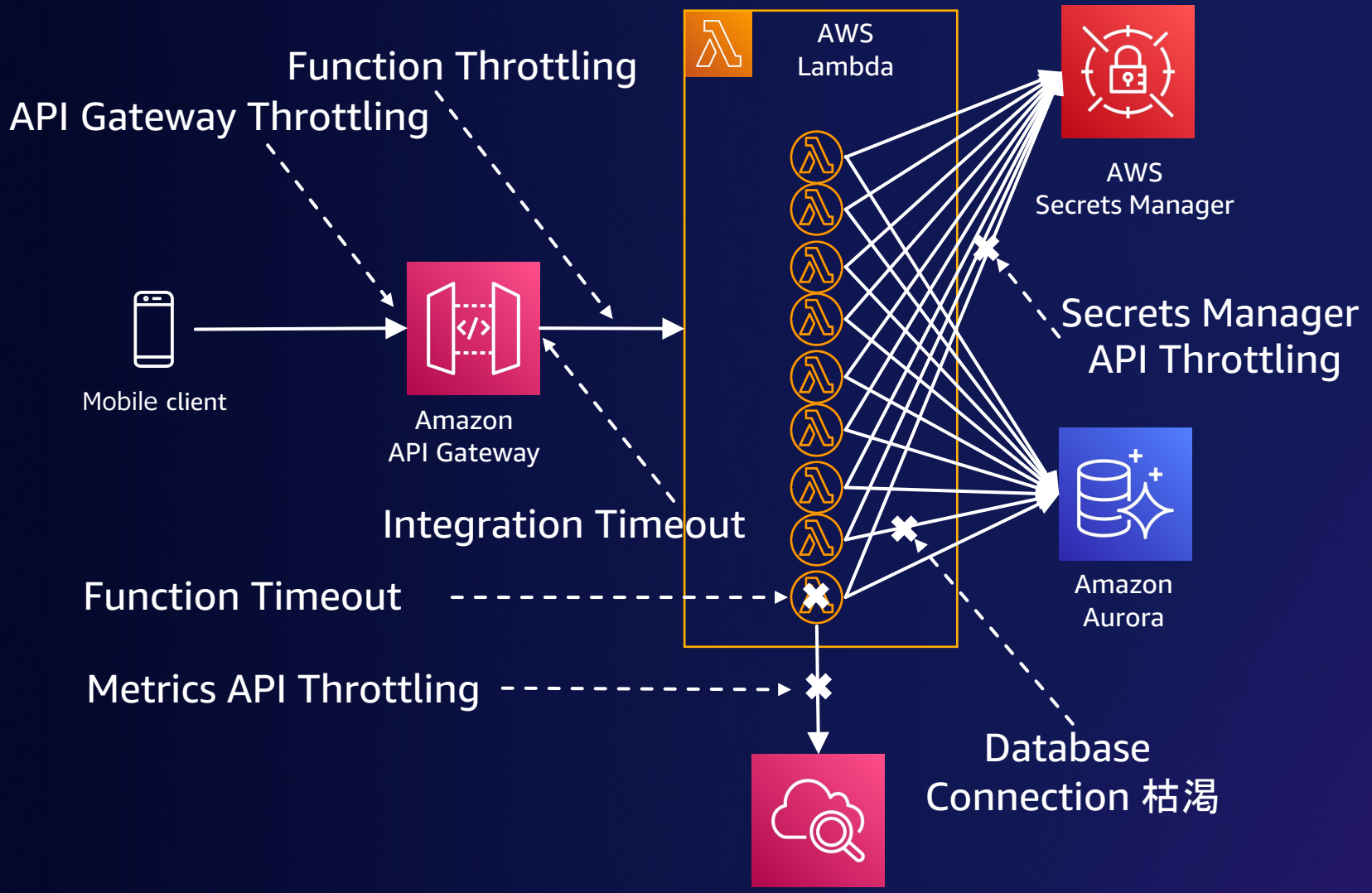
- Lambdaサービスのトラフィックルーティング
  - イベント処理終了まで、次のイベントを受け付けない
    - 揮発性はあるが、Lambda 関数インスタンスは再利用される
  - Non-blocking なリクエスト受付モデルは本来不要

- 実行環境は、次のイベント受付で再利用
  - オブジェクトのロードを遅延させる
    - 分岐条件が多数ある場合など、必要になるまでロードしない
  - グローバルスコープの利用
    - ロードされたオブジェクトはグローバルスコープで管理
  - Dynamic なオブジェクト生成をしない

同期的な Serverless API をスケール



同期的な Serverless API をスケール



[Function の詳細]
- avg duration: 100ms
- avg rps: 6000rps
- avg concurrency: 600

[負荷観測]
- DBへの負荷により接続性悪化
- 関数の実行時間が延びることにより、コストや統合サービスも遅延タイムアウトを誘発
  - API Gateway 統合タイムアウト
- Lambda関数から呼び出すサービスの Throttling
  - Secrets Manager
  - CloudWatch Metrics

Secrets Manager Throttling

```
import json
import os
import boto3

db = os.environ['RDS_DB_NAME']
user = os.environ['RDS_USERNAME']
pw_arn = os.environ['RDS_PASSWORD_ARN']

secretsmanager = boto3.client('secretsmanager')

def lambda_handler(event, context):
    secrets = secretsmanager.get_secret_value(SecretId = pw_arn)
    password = json.loads(secrets['SecretString'])['password']

    conn = openConnection(db, user, password)
    (snip)
```

関数実行毎に呼ばれる

[Secrets Manager の Quota]

Request type	Quota
GetSecretValue	5,000rps (緩和不可)

[Function の詳細]
- avg duration: 100ms
- avg rps: 6000rps
- avg concurrency: 600

[負荷観測]
- Lambda関数から呼び出すサービスの Throttling
  - Secrets Manager

※ SDK 実装で exponential backoff が設定されている場合、Lambda実行遅延に

Secrets Manager Throttling 緩和

```
import json
import os
import boto3

db = os.environ['RDS_DB_NAME']
user = os.environ['RDS_USERNAME']
pw_arn = os.environ['RDS_PASSWORD_ARN']

secretsmanager = boto3.client('secretsmanager')

secrets = secretsmanager.get_secret_value(SecretId = pw_arn)
password = json.loads(secrets['SecretString'])['password']


def lambda_handler(event, context):
    conn = openConnection(db, user, password)
    (snip)
```

[Secrets Manager の Quota]

Request type	Quota
GetSecretValue	5,000rps (緩和不可)

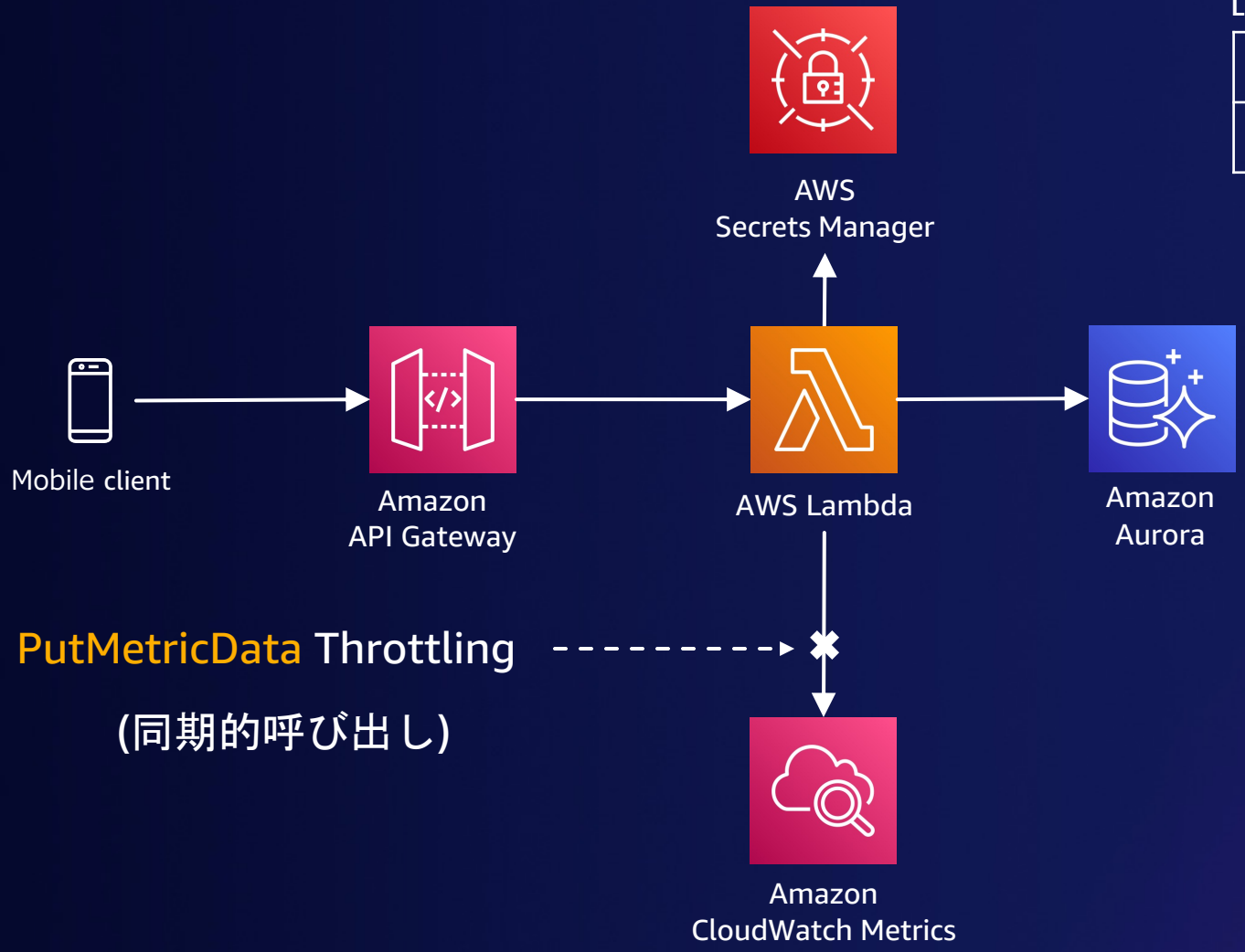
[Optimization]

- ハンドラーの外にSecrets取得を置くことにより、INIT phaseで処理可能
- 600 concurrency であれば、およそ600回の Secrets 取得に緩和

※Password のローテーション対策としては、connection 取得の例外を捕捉し、再取得後にGlobal 変数に再代入

Global Scope に移動

Metrics API Throttling



[CloudWatch Metrics の Quota]

Request type	Quota
PutMetricData	150rps (緩和可能)

[Function の詳細]
- avg duration: 100ms
- avg rps: 6000rps
- avg concurrency: 600

[負荷観測]
- Lambda関数から呼び出すサービスの Throttling
  - CloudWatch Metrics

※ custom metrics 利用時

Metrics API Throttling

```
import boto3

cloudwatch = boto3.client('cloudwatch')

def lambda_handler(event, context):

    results = conn.runQuery(sql)

    cloudwatch.put_metric_data(     // put one custom metric
        Namespace='Summit2022',
        MetricData=[{
                'MetricName': 'Conversion',
                'Dimensions': [{'Name': 'sales', 'Value': 'conv'}],
                'Value': results.Count,
                'Unit': 'Count',
                'StorageResolution': 1
            } ]
    )
```

[CloudWatch Metrics の Quota]

Request type	Quota
PutMetricData	150rps (緩和可能)

[Function の詳細]

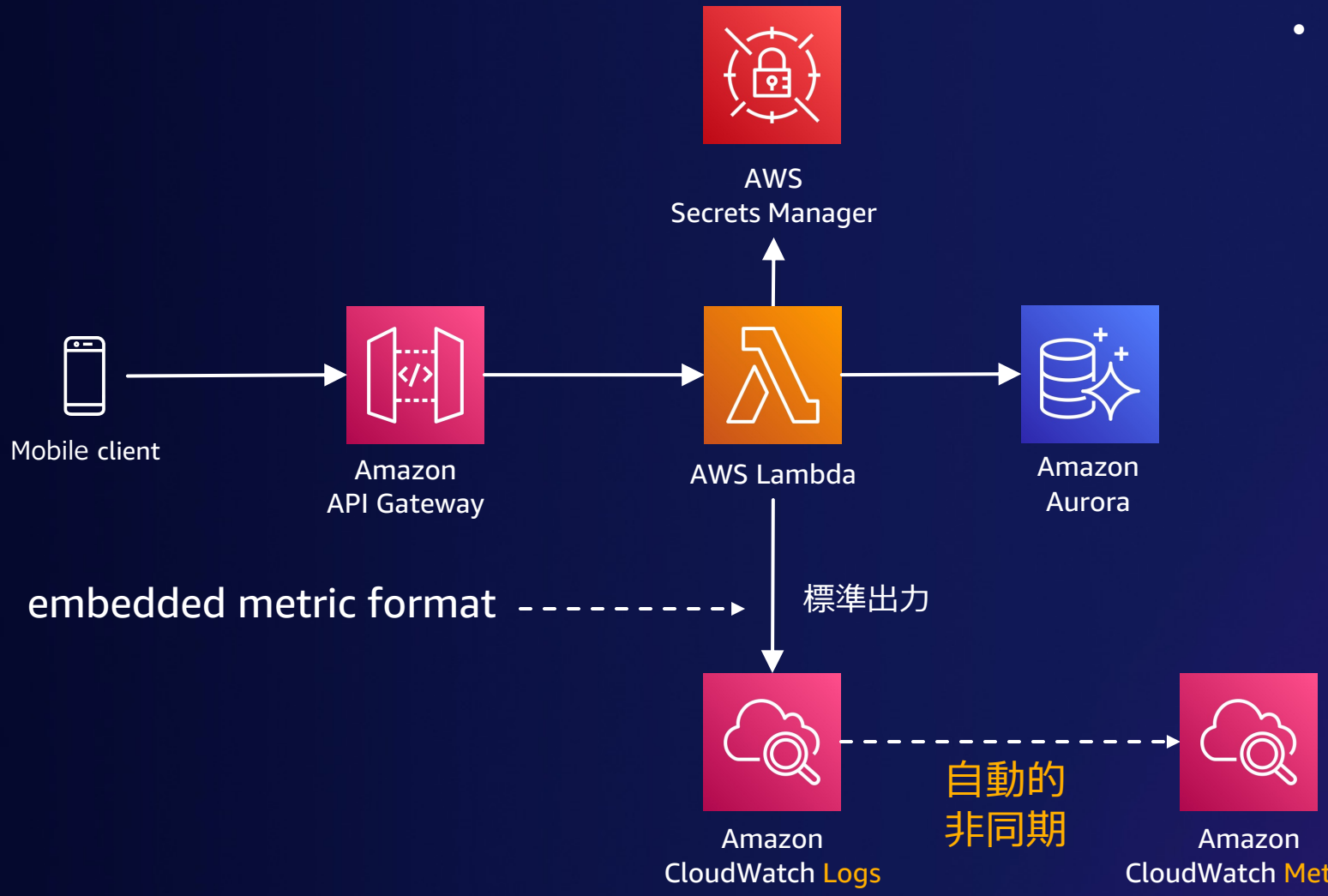
- avg duration: 100ms
- avg rps: 6000rps
- avg concurrency: 600

[PutMetricData の 価格]

1,000 リクエスト: 0.01USD

※ 0.01 USD を Lambda関数にたとえると、
1GBメモリ設定 duration 600ms の 1,000回実行分に相当

Metrics API Throttling 緩和



CloudWatch embedded metric format

- CloudWatch Logsに規定フォーマットで出力
  - オープンソースのクライアントライブラリを提供
    - Node.js
    - Python
    - Java
    - C#

    https://docs.aws.amazon.com/ja_jp/AmazonCloudWatch/latest/monitoring/CloudWatch_Embedded_Metric_Format_Libraries.html
- Lambda Powertools も利用可

  https://awslabs.github.io/aws-lambda-powertools-python/latest/core/metrics/
- CloudWatchサービスが自動的にMetricsを非同期に記録

CloudWatch embedded metric format

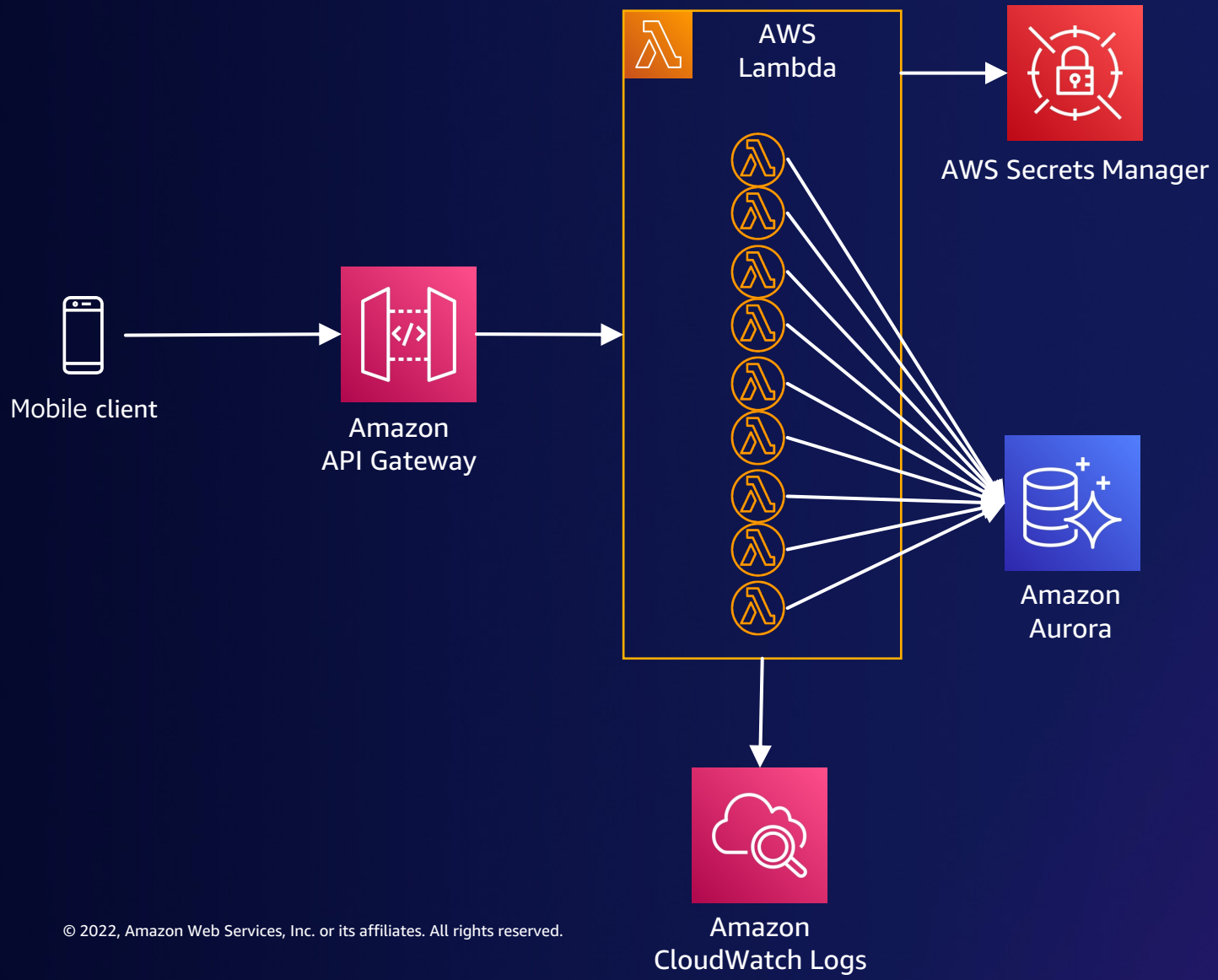
```python
from aws_embedded_metrics import metric_scope

@metric_scope
def lambda_handler(event, context, metrics):

    results = conn.runQuery(sql)

    metrics.set_namespace("Summit2022")
    metrics.put_dimensions({"dim": "sales"})
    metrics.put_metric("Conversion", results.Count, "Count")
```

CloudWatch Logs に標準出力

```
{
  "LogGroup": "embedded-log",
  (snip)
  "_aws": {
    "Timestamp": 1645369641214,
    "CloudWatchMetrics": [
      {
        "Dimensions": [["LogGroup",
            ”ServiceName", "ServiceType", "dim" ]],
        "Metrics": [{
            "Name": "Conversion",
            "Unit": "Count”
        }],
        "Namespace": "Summit2022"
      }
    ]
  },
  "Conversion": 100
}
```

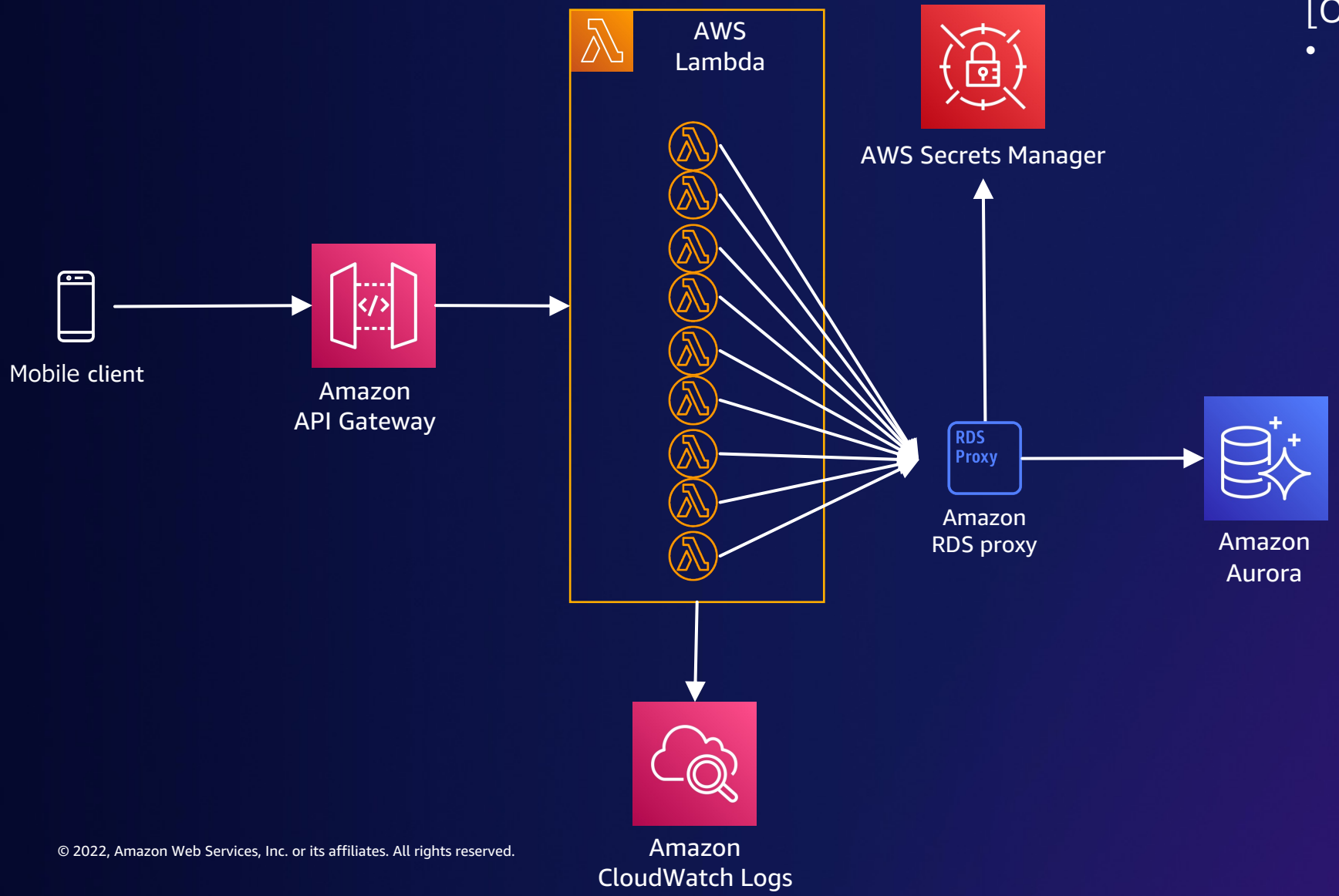
Amazon CloudWatch Logs

Database Connection 枯渇



[負荷観測]
- DBへの負荷により接続性悪化
- 関数の実行時間が延びることにより、コストや統合サービスも遅延タイムアウトを誘発
  - API Gateway 統合タイムアウト

Database Connection 枯渇：Proxy による緩和



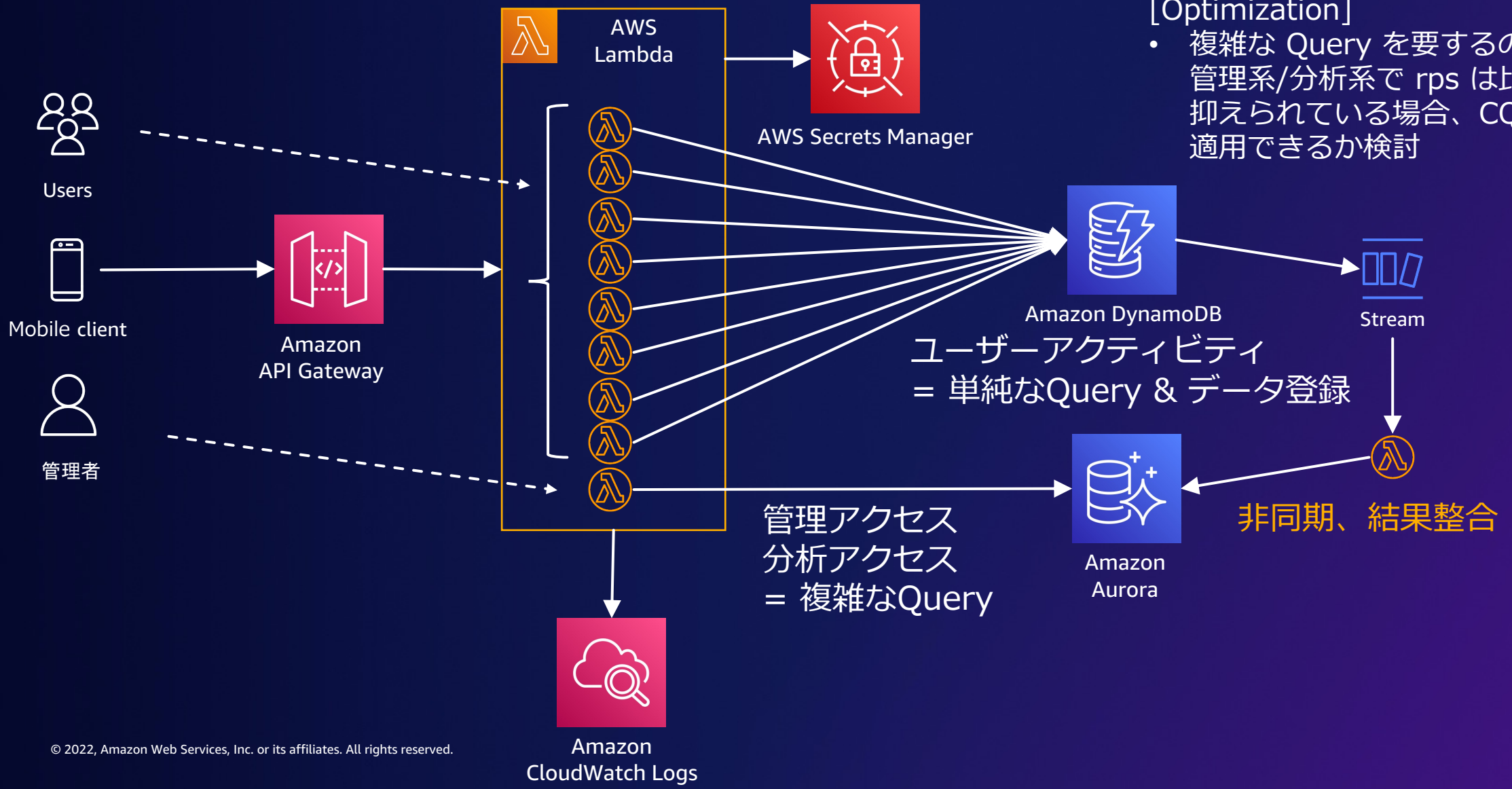
[Optimization]

- Connection Pooling 層として RDS Proxy を配置

※ RDS Proxy に Secrets 取得設定が可能なため、アプリケーションからは、IAM認証可能

※ RDS Proxy は DB instances を、アクティブに監視、クライアントを適切なターゲットに自動接続 (DNS伝搬遅延がない)

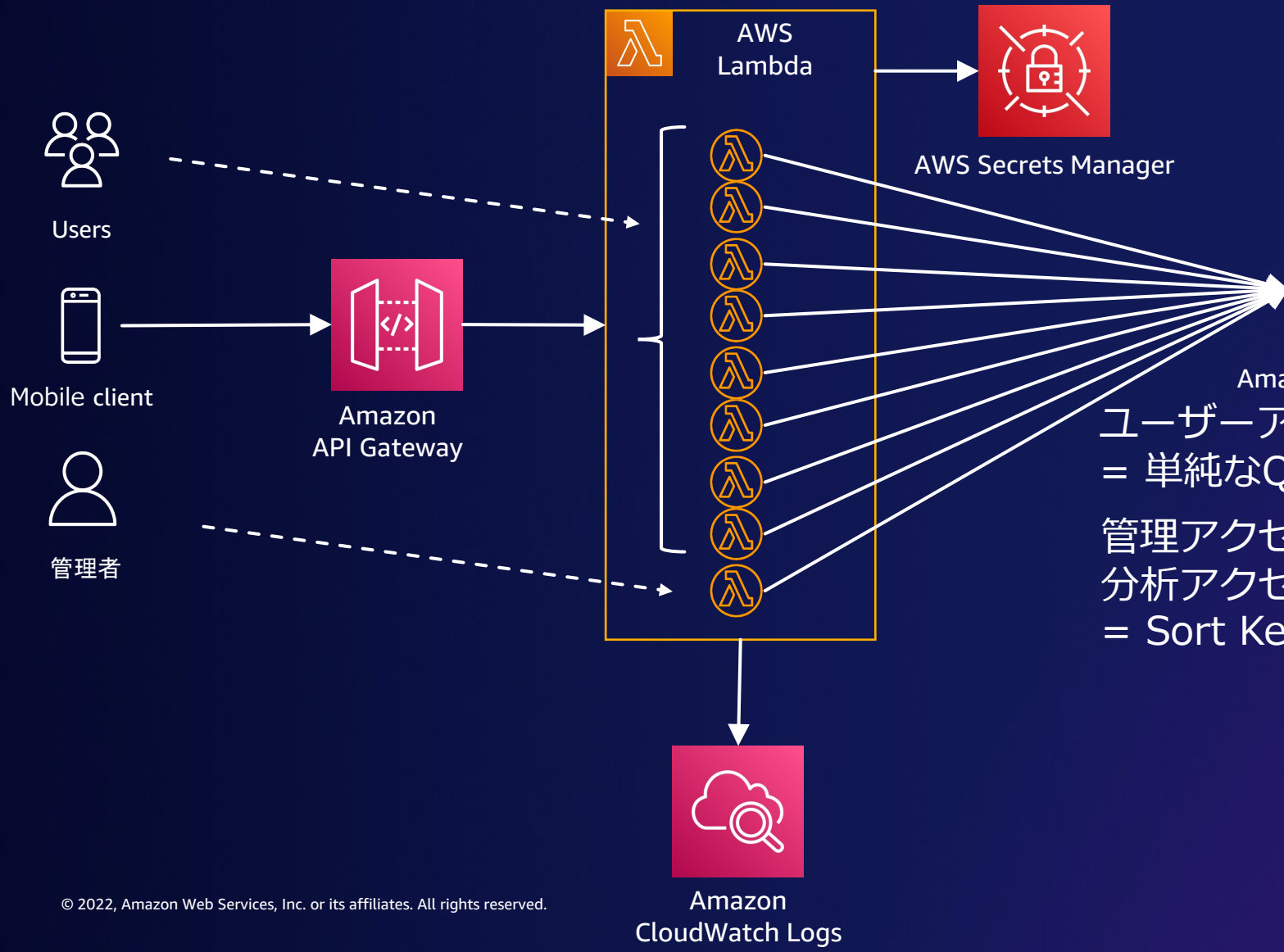
Database Connection 枯渇：CQRS による緩和



[Optimization]

- 複雑な Query を要するのは、管理系/分析系で rps は比較的抑えられている場合、CQRSを適用できるか検討

Database Connection 枯渇：Store選択による緩和

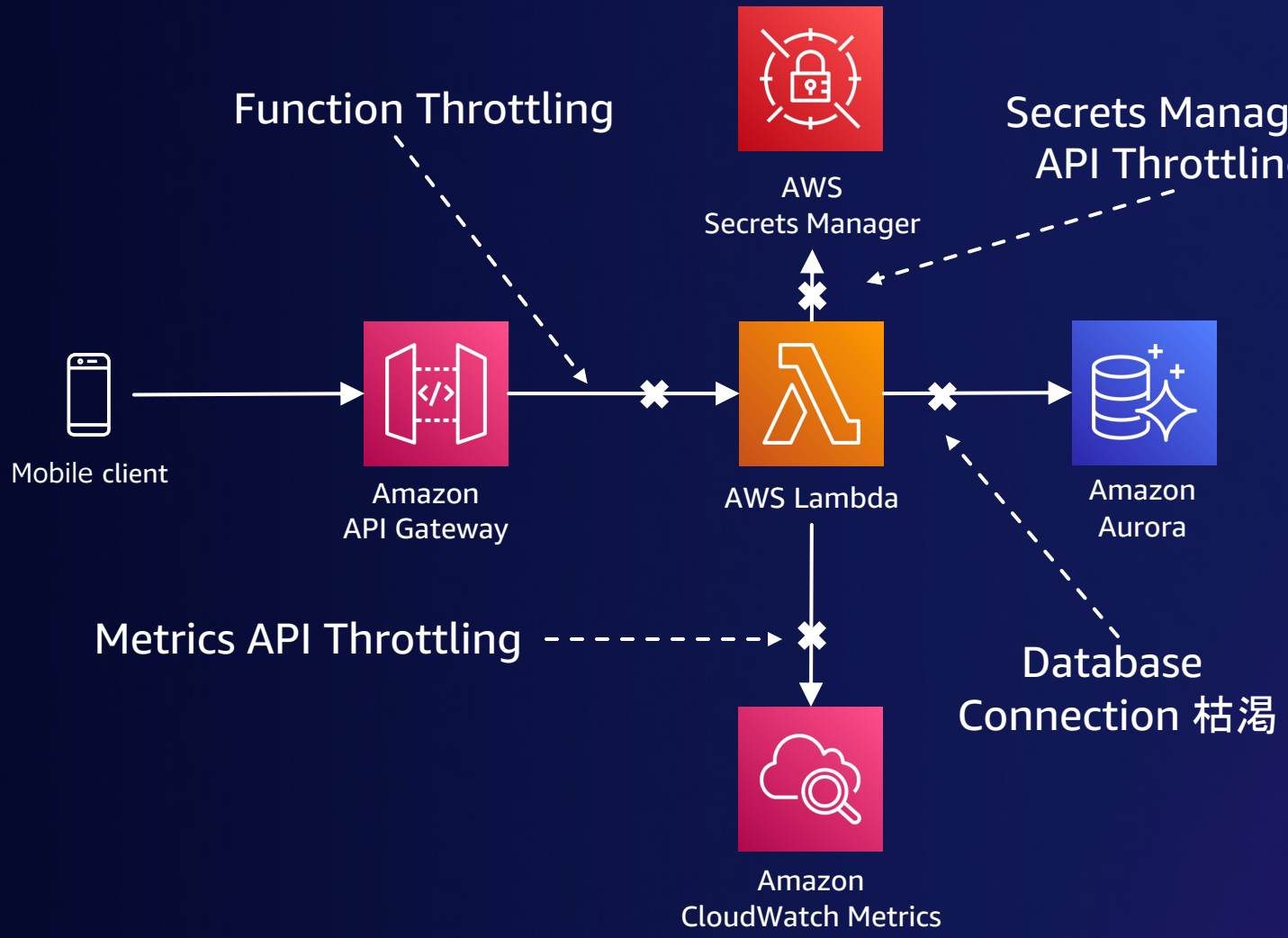


[Optimization]
- Query 特性が DynamoDBの Index設計で対応可能な場合、よりスケーラブルな構成も可能に

ユーザーアクティビティ
= 単純なQuery

管理アクセス
分析アクセス
= Sort Key、Composite Key設計、GSIの追加

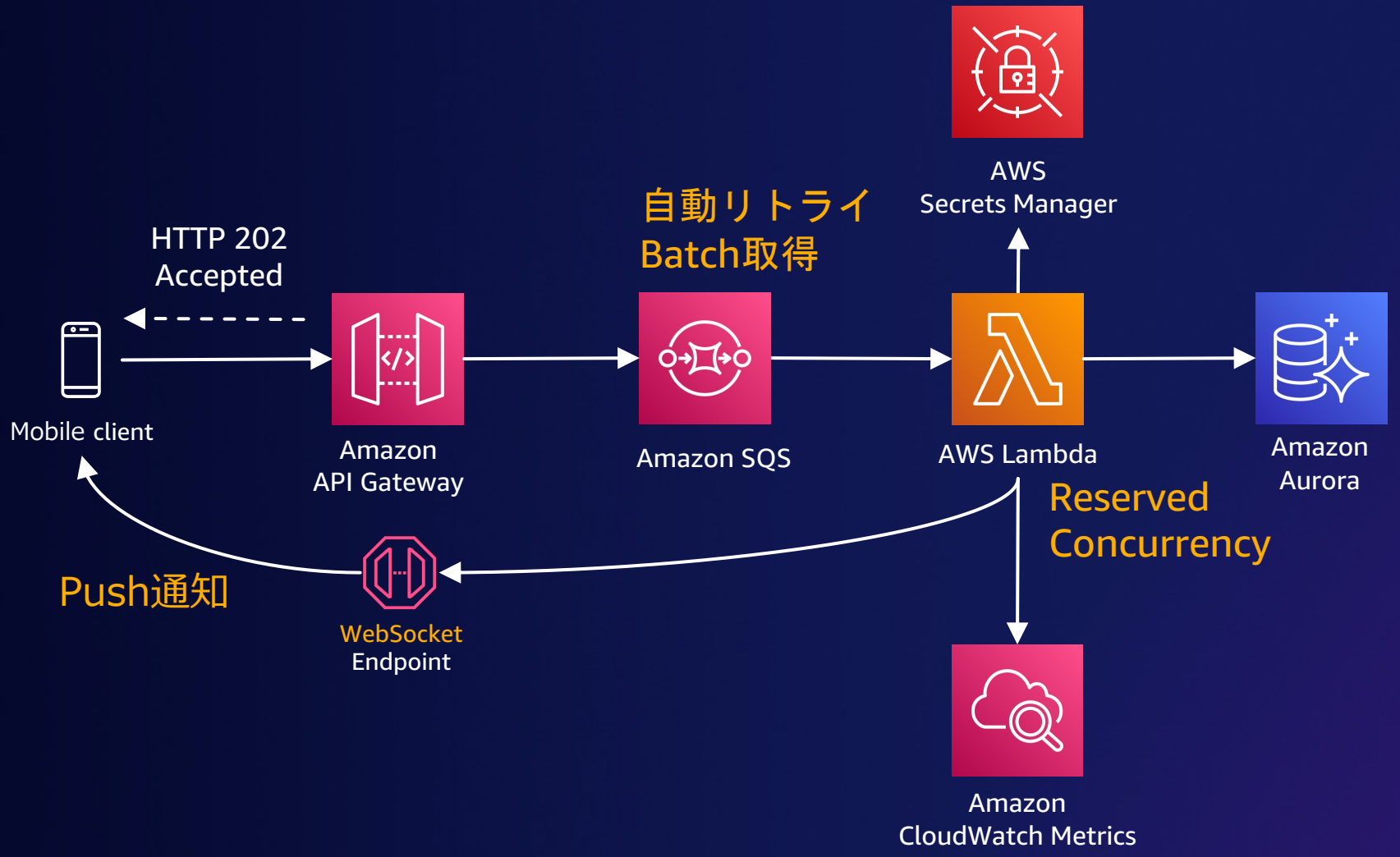
Throttling に包括的に対処



ここを下げるか　ここを下げる

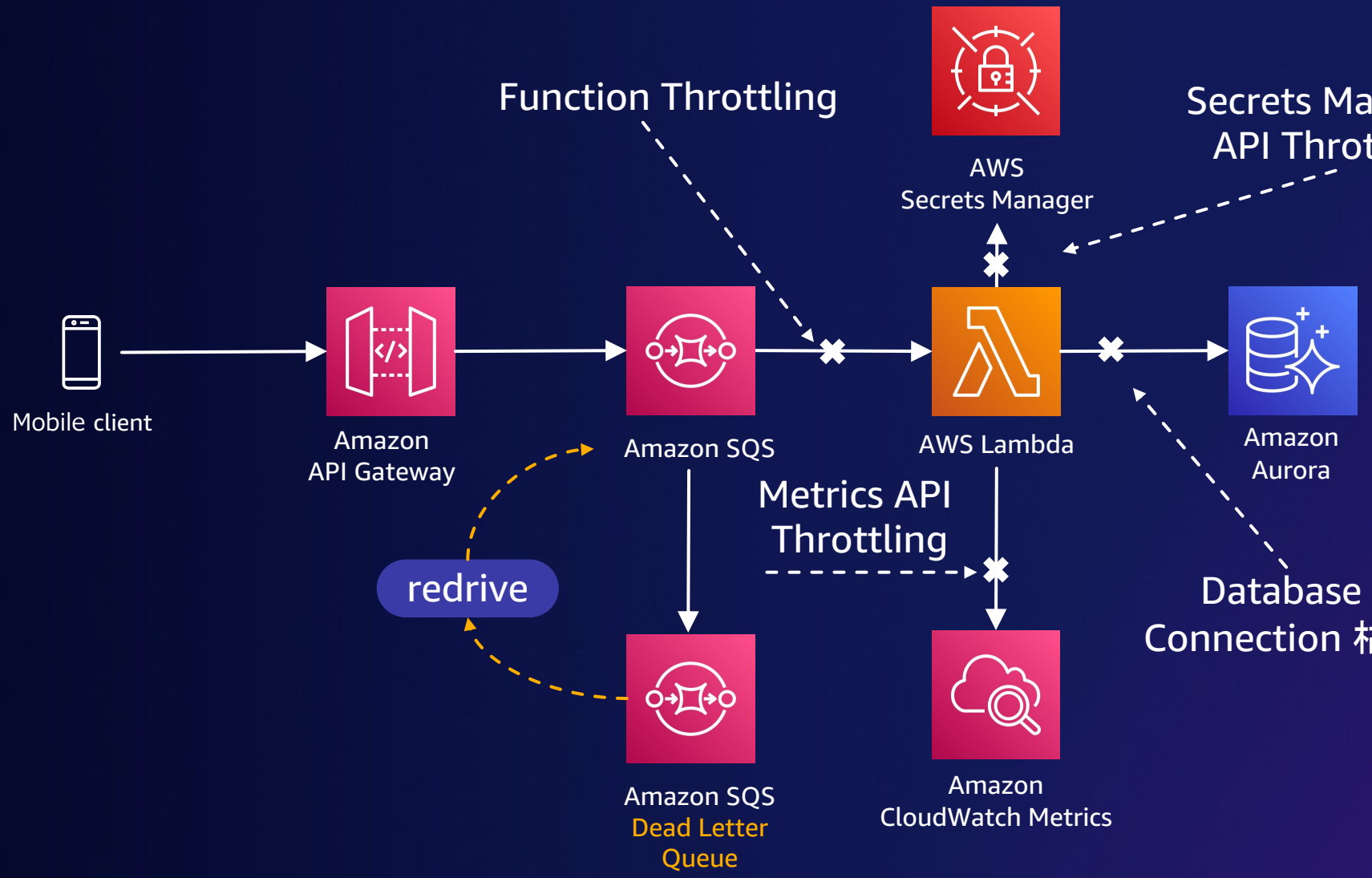
Concurrency = rps x duration

Throttling に包括的に対処：Queue による緩和



1. API GatewayとLambdaの間に SQS を挟む
2. Lambda関数には、同時実行数を予約しておき、データベースに高負荷を与えないようにする
3. API Gatewayとクライアント間で WebSockets の接続を確立しておく
4. Lambda関数は処理が終われば、WebScoket を介して結果をクライアントに返却

Throttling に包括的に対処：Queue による緩和



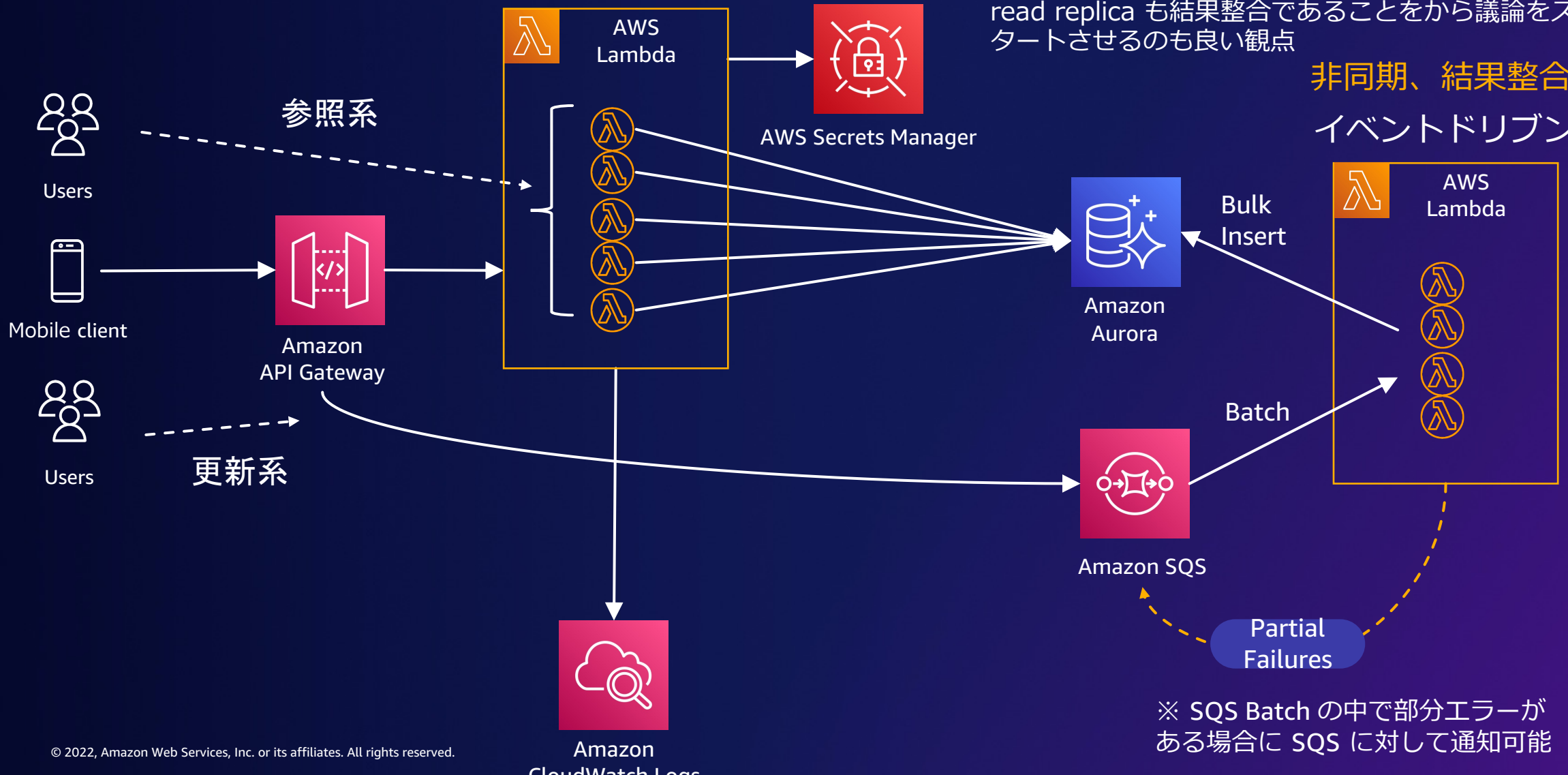
- それでも何らかのエラーが関数実行で発生する場合、DLQにmessageを待避
- デバッグした後、マネージドな redrive 機能を利用し、ソース Queue に message を再投入

Throttling に包括的に対処：Queue による緩和

※ 結果整合を許容できるかを考える場合、Aurora read replica も結果整合であることをから議論をスタートさせるのも良い観点

非同期、結果整合

イベントドリブン

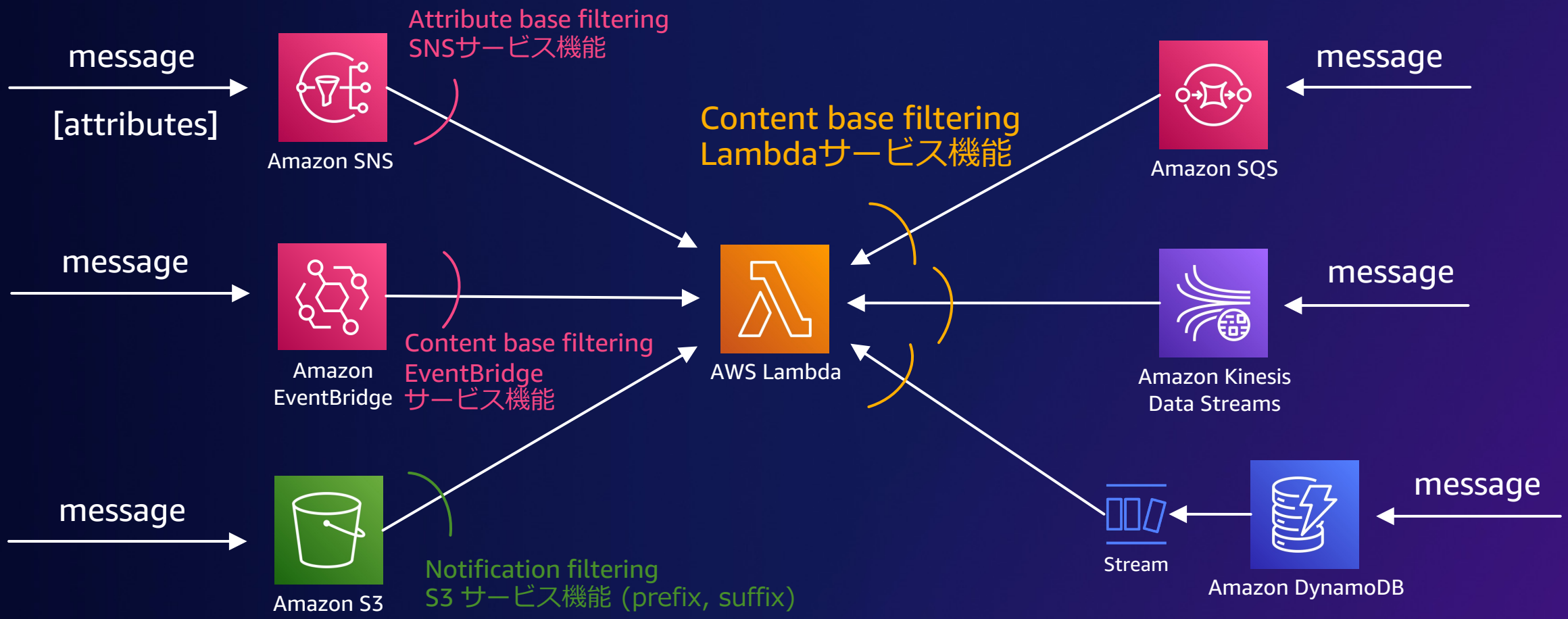


※ SQS Batch の中で部分エラーがある場合に SQS に対して通知可能

簡潔な Lambda 関数

- Lambda 関数のコアロジックと Handler entry point の分離
  - イベントソースサービスの情報をロジックに混入させない
  - イベントソースサービスの変更を用意に
  - 単体テストをローカルで高速に実行可能に
- Lambda 関数の利用は TRANSPORT ではなく、TRANSFORM
  - Network intensive な処理は別サービスにオフロードし CPU intensive を目指す
- 必要な情報だけを取得
  - Lambda がアクセスする永続層は 適切に、Index されている
  - Stream、Queue、Event Driven アーキテクチャは適切に Filter する
  - S3 から Objectを取得する際は、GetObject よりも、SelectObjectContent

Event Filtering



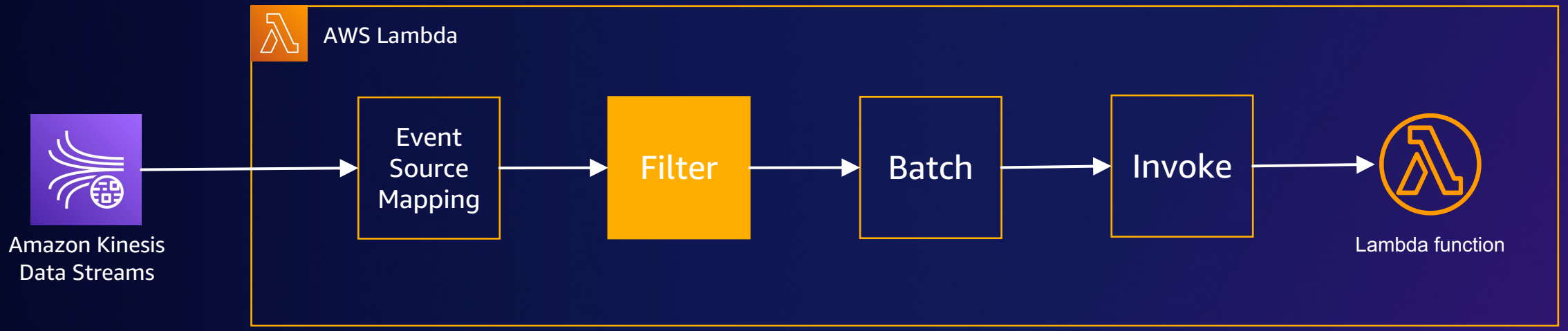
Event Filteringにより、Lambdaの実行数を削減可能

AWS Lambda 機能の Event Filtering

- Lambda関数実行前のフィルタリング
  - EventBridge と同じ形式のマッチング
    - イベントソースごとに、最大5パターン
    - (組み合わせは OR 条件)
    - パターンごとに最大2048文字
    - フィルタリングの後に、バッチ収集実施
- Lambda 関数実行コストの低減
- サポートイベントソース:
  - Kinesis Data Streams
  - DynamoDB Streams
  - SQS

```
Events:
  TirePressureEvent:
    Type: Kinesis
    Properties:
      BatchSize: 100
      StartingPosition: LATEST
      Stream: "arn:aws:kinesis:[region]: - :stream/hello"
      FilterCriteria:
        Filters:
          - Pattern: '{
              "data": {
                  "tire_pressure": [{"numeric": ["<", 32]}]
              }
            }'
```

AWS Lambda 機能の Event Filtering



イベントがAWS Lambda サービスに届くと、

1. まず定義されたFilterに一致するか判断
2. その後、「Batch（Lambda 関数に渡すイベントまとまり）」を生成
3. Batch単位にLambda関数が起動

※このため、Amazon Kinesis Data Streams では、Batchに入らない場合でもキャパシティが消費される
e.g.) 数万レコード入れ、1件もフィルタ条件に一致しない場合もキャパシティユニットは消費される
※ Kinesis Data Streams の max read capacity は 2MB/s

AWS Lambda 機能の Event Filtering

Filter と不一致だったイベントに対する挙動

	Kinesis Data Streams / DynamoDB Strems	SQS
不一致の場合	対象レコードは処理されたとみなされ スキップして次に進む	メッセージがキューから削除される

※ SQS は Filter 不一致の場合、キューからメッセージが削除されるため、Queue の中身を 再処理 したい場合は、Lambda サービスへの受け渡しと同時に メッセージを別の Queue に退避できるように、SNS による Fan out をしておくなど工夫が必要。

S3 Select を利用

```
import boto3
s3 = boto3.client('s3')

def lambda_handler(event, context):

    result = s3.select_object_content(
        Bucket='summit2022',
        Key='accounting/data.json',
        ExpressionType='SQL',
        Expression="""
            SELECT
                substring(s.dir_name,1,5) as sub
            FROM s3object s
                where s.dir_name = 'other_docs'
        """,
        InputSerialization = {'JSON': {"Type": "Lines"},
            'CompressionType': 'NONE'},
        OutputSerialization = {'JSON': {}},
    )
```

JSON LINES形式

s3://summit2022/accounting/data.json

```
{"dir_name":"important_docs","files":[{"name":"."},{"n
{"dir_name":"other_docs","files":[{"name":"."},{"name
```

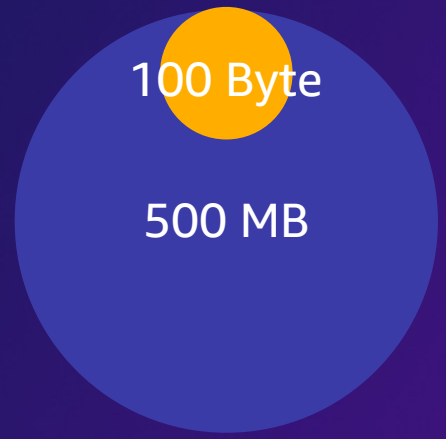
(snip)

result

```
{"sub":"other"}
```

S3 Selectを使うと

- 取得行数を削減
- 取得文字列長を削減
- S3 側で Map/Reduce

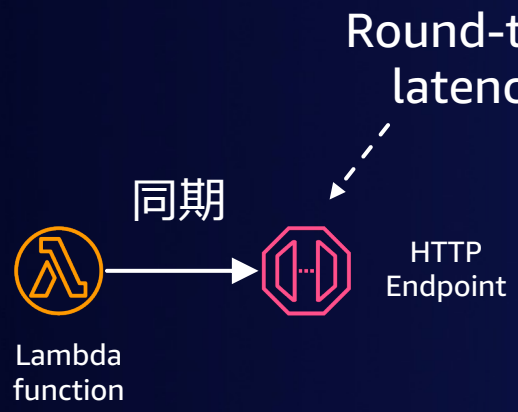


https://jsonlines.org/

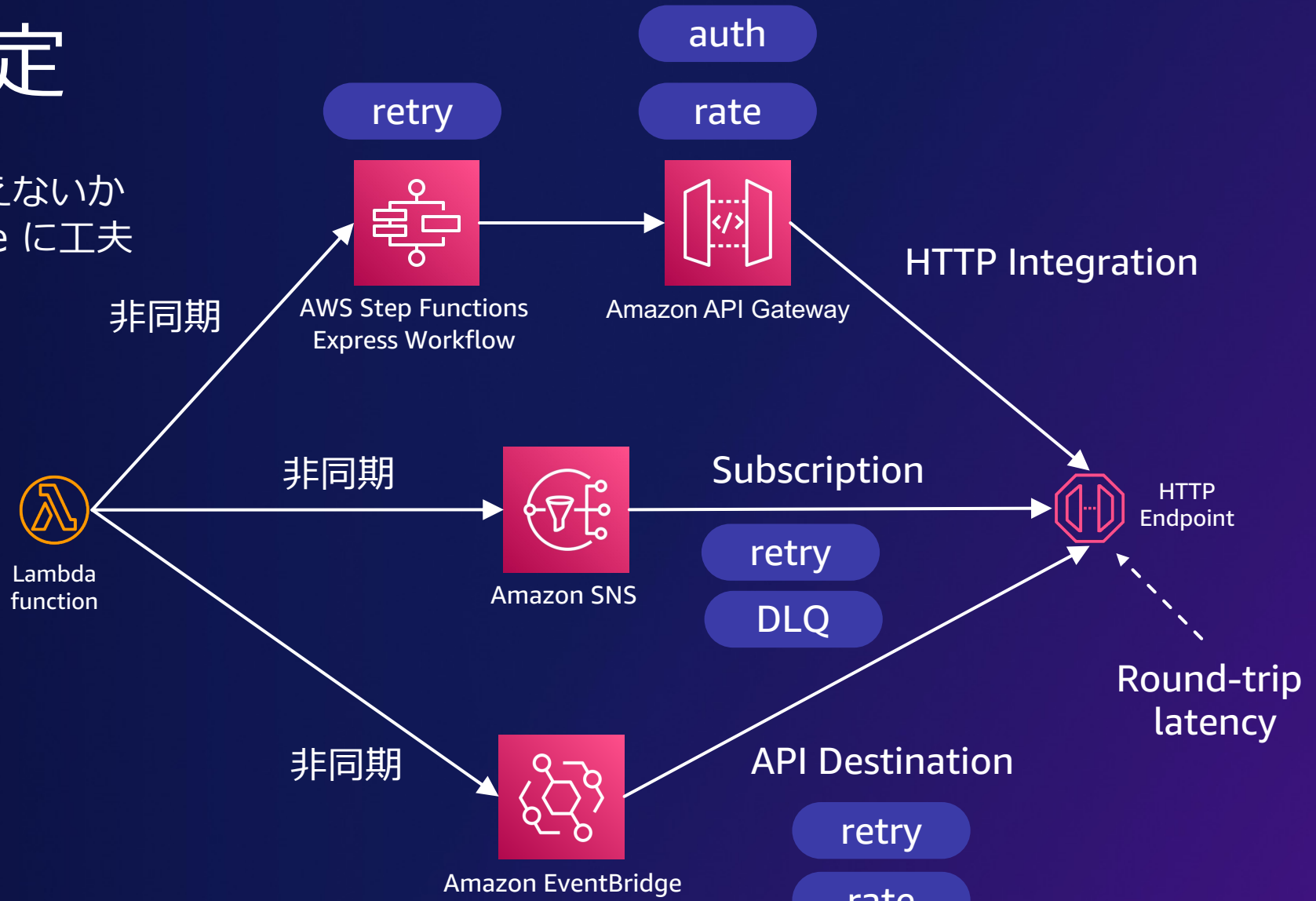
https://github.com/awslabs/lambda-refarch-mapreduce

API target の設定

外部 API 呼び出しを非同期的に扱えないか
Lambda 関数実装を CPU intensive に工夫




```
import requests
url = 'https://example.com/'
response = requests.get(url)
```



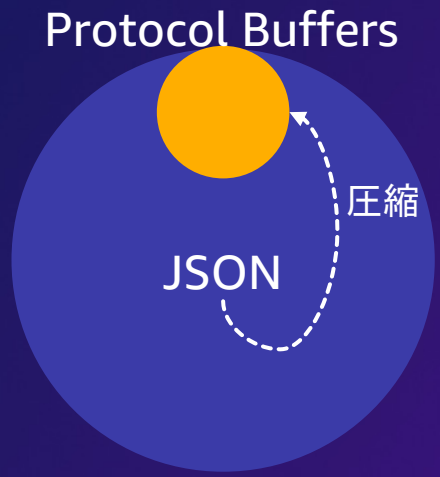
外部APIの同期呼び出しは、実行時間に加算されるため
同時実行数の増加に寄与する

Protocol Buffers による軽量通信

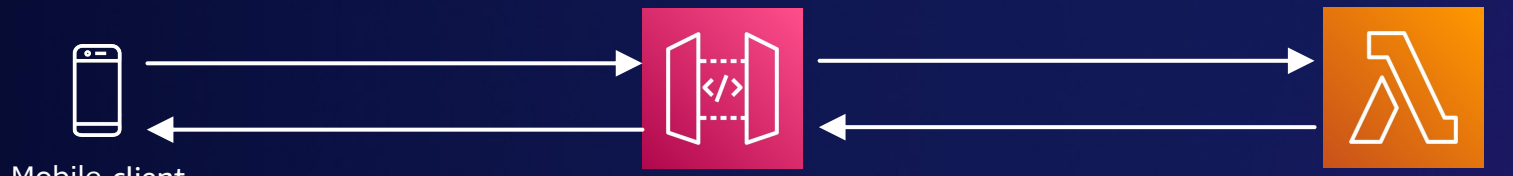
※ payload size により base64 がオーバーヘッドになることもあるため、ワークロードにより選択

protobuf (Protocol Buffers) はJSONの代替バイナリデータフォーマットとして通信Payloadに利用

base64.b64encode(person.SerializeToString())
protobuf エンコード



Accept: application/x-protobuf



protobuf デコード

BinaryMediaTypes:
- application/x-protobuf

```
"headers": {
    'Content-Type’:
        'application/x-protobuf’
},
"isBase64Encoded": True,
```

※ gRPC はHTTP/2をトランスポートとして利用し、Protocol Buffers をインターフェース記述言語およびエンコーディングとして利用するが API Gateway は gRPC および HTTP/2 を未サポート

Protocol Buffers による軽量通信

```
$ protoc --python_out=./ ./person.protols
#事前にmessage formatをコンパイル
```

https://developers.google.com/protocol-buffers/docs/pythontutorial

```
import base64
from person_pb2 import Person

def lambda_handler(event, context):
  person = Person()
  person.id = 9971
  person.name = ”Joseph"
  person.email = " joseph@example.com"
  byte_data = base64.b64encode(person.SerializeToString())

return {"statusCode": 200,
  "headers": {'Content-Type': 'application/x-protobuf’},
  "body": str(byte_data,'utf-8’),
  "isBase64Encoded": True,
}
```

protobuf (Protocol Buffers) の AWS SAM を利用した実装例

```
Resources:
  binary:
    Type: AWS::Serverless::Api
    Properties:
      BinaryMediaTypes:
       - application~1x-protobuf
  binaryFunction:
    Type: AWS::Serverless::Function
    Properties:
      Events:
        ApiEvent:
          Type: Api
          Properties:
            RestApiId:
              Ref: binary
```

※一部抜粋

Keep-Alive を利用したHTTP 接続再利用 (Node.js)

- デフォルトでは、デフォルトの Node.js HTTP / HTTPS エージェントは、新しいリクエストごとに新しい TCP 接続を再作成
  - DynamoDB へのクエリなど短時間操作の場合、TCP 接続を設定するためのレイテンシーオーバーヘッドは、操作自体よりも大きくなる傾向がある
  - 新しい接続を確立するコストを回避するために、既存の接続を再利用
- Node.js 以外の他の SDK ではデフォルトで Keep-Alive 設定されているものがほとんど

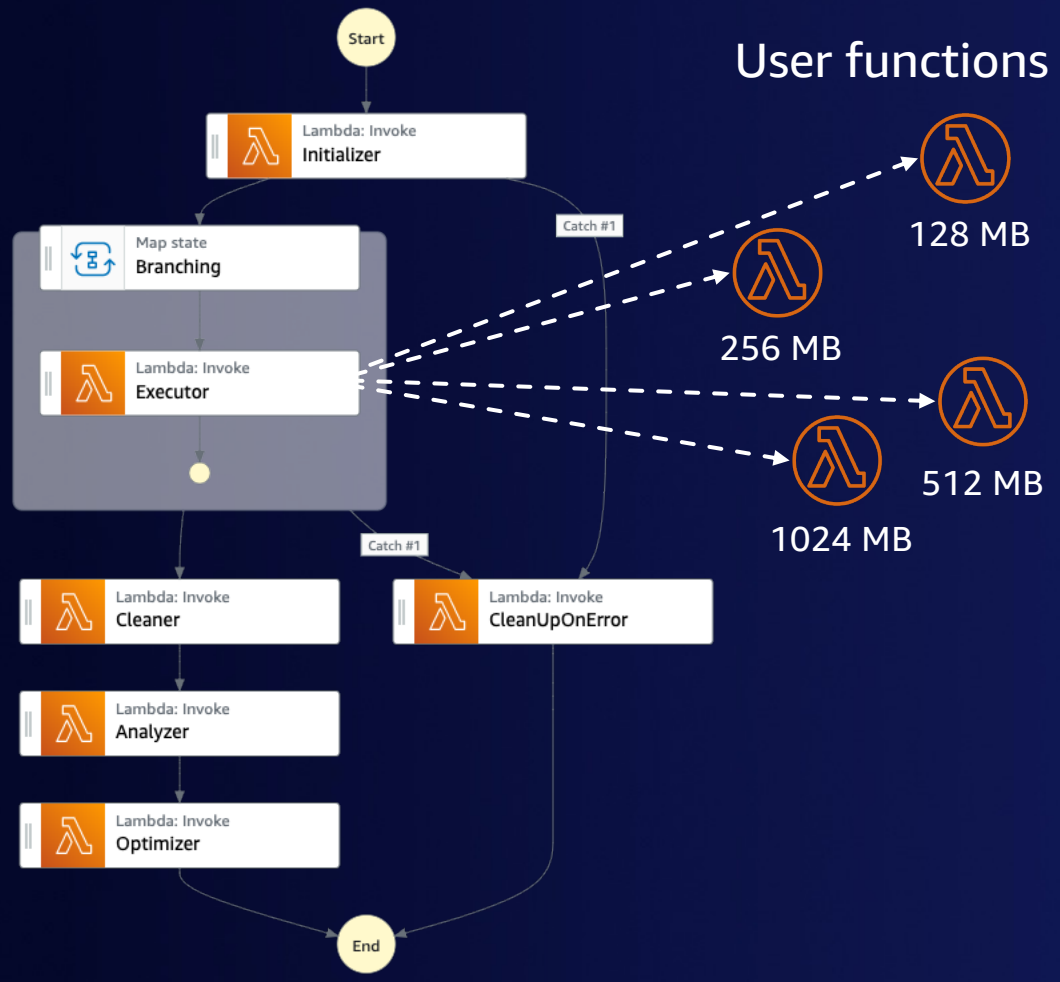
  (e.g. boto3)

https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/node-reusing-connections.html


```
const AWS = require('aws-sdk')
const https = require('https');
const agent = new https.Agent({
  keepAlive: true
});

const dynamodb = new AWS.DynamoDB({
  httpOptions: {
    agent
  }
});
```

CPU の上手な利用方法

AWS Lambda Power Tuning によるチューニング



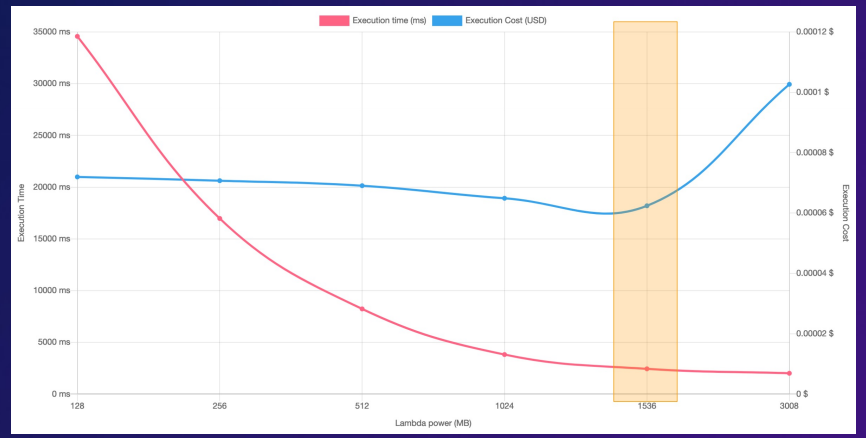
User functions

AWS Lambda Power Tuning は、
AWS Step Functions を利用したステートマシンであり、
Lambda 関数のコストやパフォーマンスを最適化

INPUT

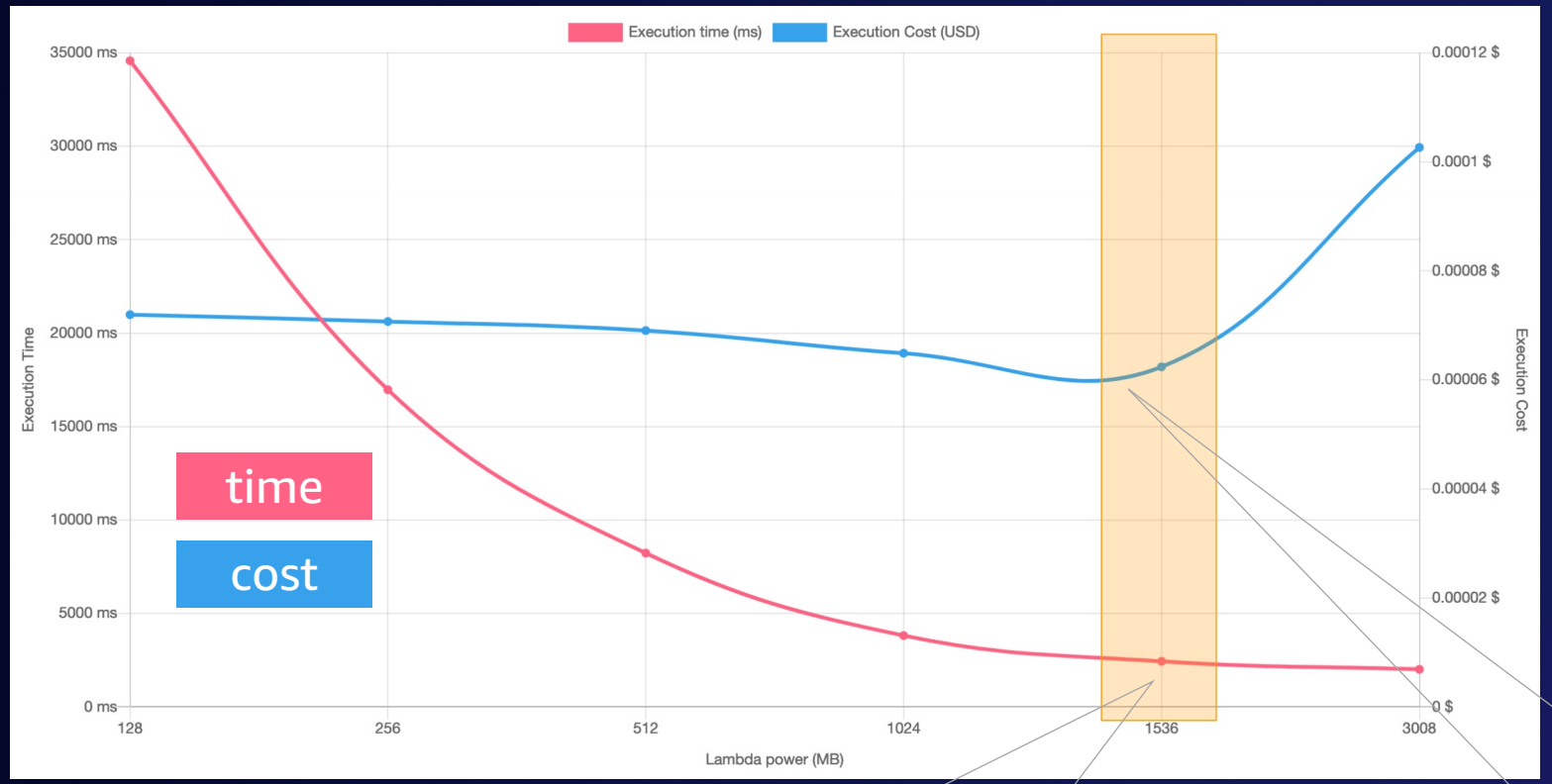
```
{
  "lambdaARN": "arn:aws:lambda: snip",
  "powerValues": [128,256, 512, 1024,1536,2048, 3008],
  "num": 1000,
  "payload": { },
  "parallelInvocation": true,
  "strategy": "speed | cost"
}
```

OUTPUT



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

CPU Intensive な処理系




```
def lambda_handler(event, context):

  heavy_CPU_job()

  return {
    'statusCode': 200,
    'body': json.dumps('CPU worked'),
  }
```

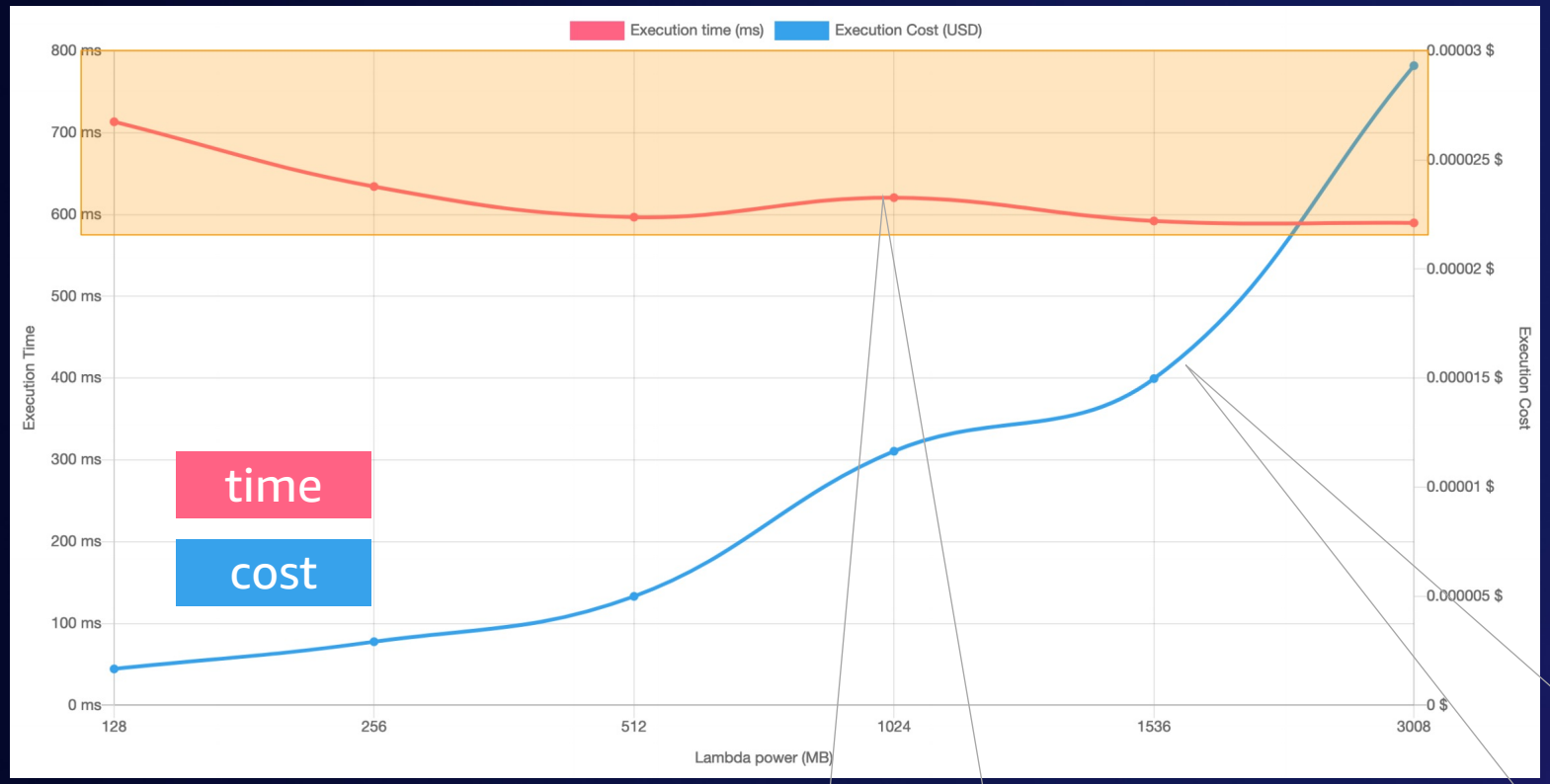
MB

メモリが1536MB付近の
時に処理時間が最小になり
その後は減少しない

メモリが1536MB付近の
時にコストも底を打つ

コスト = GB秒 x リクエスト件数

外部 APIコールに依存する処理系




```
def lambda_handler(event, context):

  external_API_call()

  return {
    'statusCode': 200,
    'body': json.dumps('API worked'),
  }
```

MB

メモリが上がっても
処理時間に対する
貢献が少ない

メモリを上げると
コストも上がる

コスト = GB秒 x リクエスト件数

INIT CPU boost と、Lambda Lifecycle

※ Provisioned Concurrency で与えられた INIT フェーズでは boostしない
※ Node.js 以外の言語では同期的な INIT 処理を組むことで利用可能

https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.html

ES modules の効能 (Node.js 14+)

- import/export オペレータによるモジュール利用可能
  - Node の CommonJS modules ではなく、JavaScript(ES6) 標準モジュール
  - webpack などのパッケージングで 静的解析や tree shaking (実行されないコードを削除する機能) の恩恵享受
  - ES modules は、デフォルトで strict mode 適用
  - 厳格な構文管理の適用を簡単に
- top-level await をサポートするように改善
  - INIT フェーズでの CPU boost の恩恵授与
  - Provisioned Concurrencyと組み合わせて Cold Start をより効果的に短縮

https://aws.amazon.com/blogs/compute/using-node-js-es-modules-and-top-level-await-in-aws-lambda/

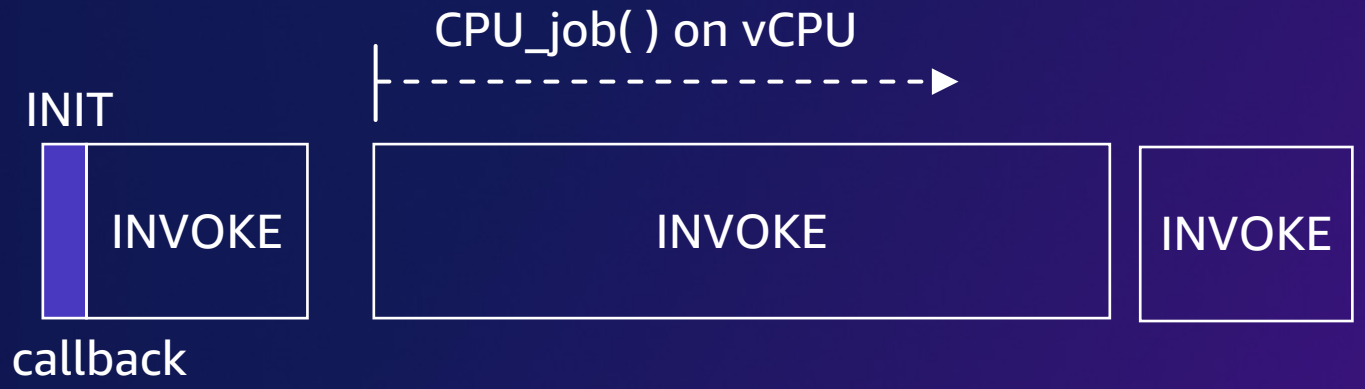
top-level await による CPU boost (Node.js 14+)

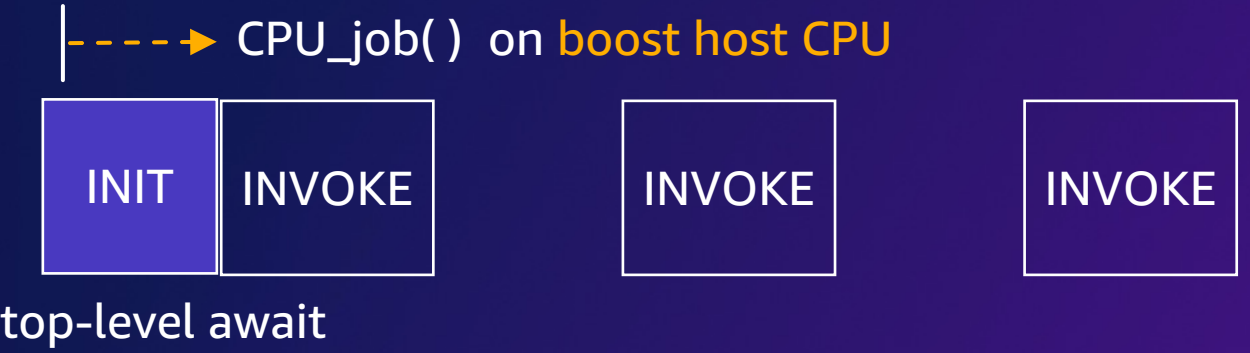
```
// app.js
import { CPU_job } from './lib.js';
await CPU_job(); // top-level await

export async function handler() {
    return "done";
};

// lib.js
export async function CPU_job( ) {
    return new Promise(resolve => {
        setTimeout(() => {
            // CPU intensive job here
            resolve('resolved');
        }, 1000);
    });
}
```

INIT

INVOKE

top-level await を利用しない場合、callback の event cycle 割り当て次第で、callback 実行のタイミングが次回の INVOKE タイミングになることも





※ Cold Start 遅延とのトレードオフにあることに注意

ES modules として Lambda 関数を扱うには

以下のどちらかの方法か、または組み合わせて利用可能

方法1

```
// package.json
{
    "name": "module-example",
    "type": "module",
    "description": "ES module.",
    "version": "1.0",
    "main": "index.js",
    "author": ”Steve Ryan",
    "license": "ISC”
}
```

type のデフォルトは “commonjs” だが、”module”を指定すると、パッケージ内のすべての .js 拡張子が ES modules 扱いに

<

個別に
拡張子指定で
override

方法２

```
// index.mjs
import { square } from './lib.mjs';

export async function handler() {
 let result = square(6); // 36
 return result;
};

// lib.mjs
export function square(x) {
 return x * x;
}
```

ES modules として扱いたいファイルの拡張子を .mjs とすることで個別指定

Multi thread を利用するか?

- Lambda は、設定されたメモリ量に比例して CPU パワーを割当
  - 関数のメモリを設定する際は、128 MB～10,240 MB の値を 1 MB 単位で設定
    - メモリをより多く割り当てれば、関数はより多くの CPU Cycle を獲得
    - 1,769 MB の割り当てが、1 つの vCPU (1 秒あたりのクレジットの 1 vCPU 秒分) に相当
    - Memory < 1,769 MB
      - CPU intensive な処理においても 、Multi thread (or Multi process) 効果が見えにくい
    - Memory >= 1,769 MB
      - Core 数が増加した時点で、Multi thread (or Multi process) の効果が期待

※ Lambda 関数 における Multi thread はプログラミングやエラーハンドリング
を複雑にするため、Lambda 関数の並列起動を推奨

AVX2 拡張命令 (x86_64)

- 計算集約型関数のパフォーマンス向上:
  - 機械学習の推論
  - マルチメディア処理
  - HPC
  - 金融モデル計算

- AVX2 拡張命令セット
  - 利用するためには自身のコードやライブラリが AVX2 命令セットに最適化されている必要があるため注意

Filter	Standard	With AVX2	Performance Improvement
1. Bilinear	105 ms	71 ms	32%
2. Bicubic	122 ms	72 ms	40%
3. Lanczos	136 ms	77 ms	43%

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

AWS Graviton2 Processor (arm64)

高パフォーマンス
vs x86

20% コスト削減
vs 同サイズの Lambda 関数

最大34%の
コストパフォーマンス向上

- インタープリター型およびコンパイル済みバイトコード言語は変更なしで実行可能
  - コンパイル言語は arm64 用にリコンパイルが必要
    - インタープリター型言語でもネイティブライブラリ利用の場合はリコンパイル
  - ほとんどの AWS ツールと SDK は Graviton2 を透過的にサポート
    - AWS CLI v1、AWS CLI v2
    - C/C++、node.js、Python、Go、Java、.NET用のSDK

Key Takeaways

Key Takeaways

- Lambda の ephemeral(揮発性)を理解し、statelessに実装を
  - 揮発性はあるが、Lambda 関数インスタンスは再利用される
- Lambda の コーディングプラクティスを理解し効率の良い実装を
  - コードアーティファクトサイズ最小化
  - Lambda関数から別サービス呼び出し時のレイテンシーに注意
- Lambda だけでは解決が難しい課題には、、、
  - アーキテクチャやアルゴリズムで解決を！

Happy Coding !

Thank you!

Kensuke Shimokawa

_kensh

Follow me on Twitter