OCNでサーバレス導入してみたけど、通信サービスでクラウド使うのってどうなのよ？

NTTコミュニケーションズ株式会社
伊藤 良哉/松田 丈司

自己紹介

伊藤 良哉
NTTコミュニケーションズ株式会社

松田 丈司
NTTコミュニケーションズ株式会社

JANOG43

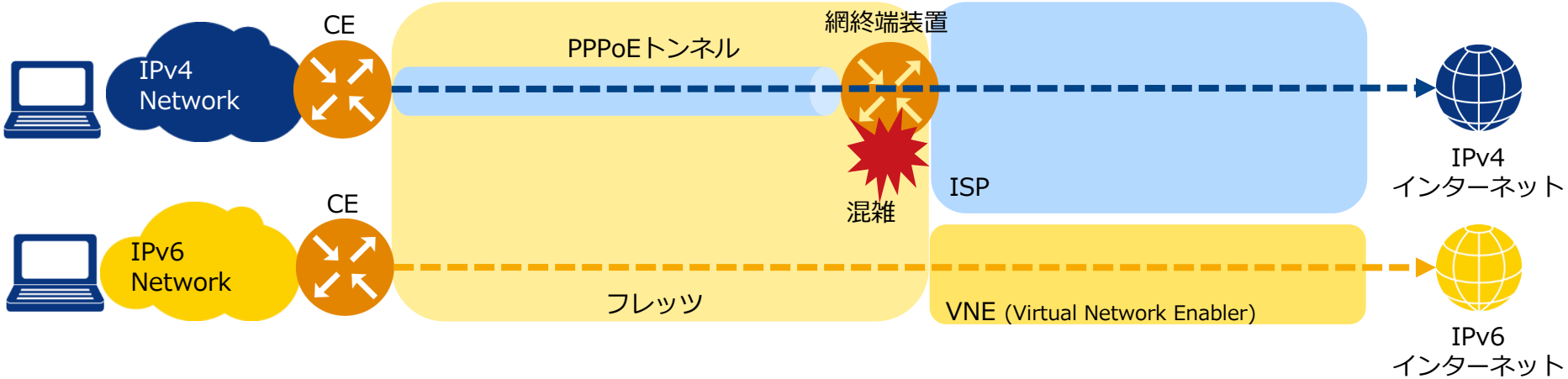
BKNIX2019

イントロダクション

IPoEのおさらい

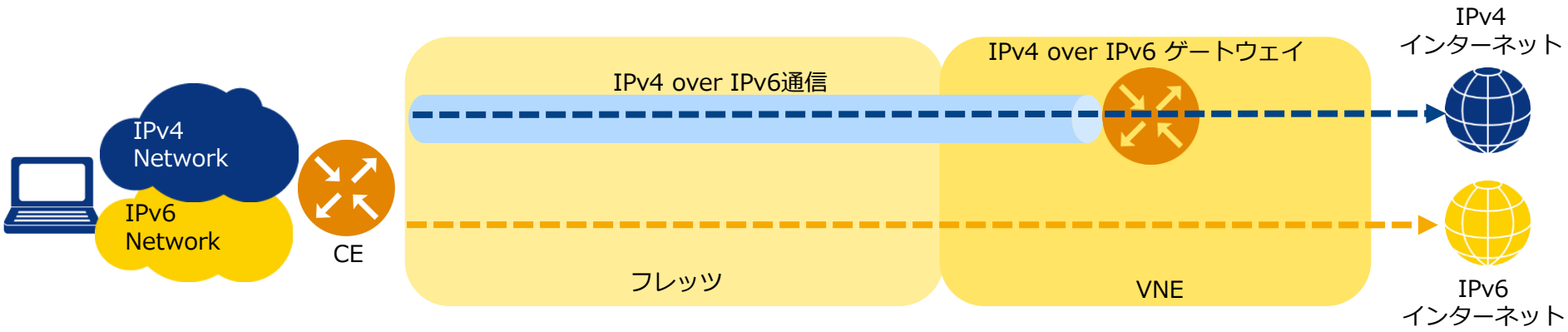
■ PPPoE (Point-to-Point Protocol over Ethernet)
- 最も普及しているフレッツの通信方式
- 近年、時間帯によってはフレッツ網終端装置が混雑し、速度が低下する傾向

■ IPoE (IP over Ether)
- IPv6インターネットを利用する場合の方式のひとつ(ネイティブ方式)
- 近年はPPPoEよりも速度が出るため、推奨する事業者が多い

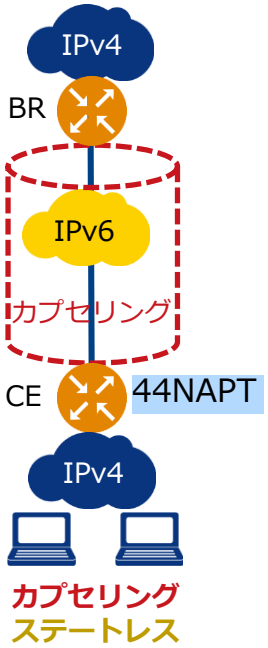


IPoE方式でIPv4通信するには？

- IPoE方式の場合、フレッツはIPv6接続のみ提供している
- VNEがIPv6からIPv4通信するためのGWを用意する
- そのゲートウェイを通過することでIPv4通信を可能とする
- NTTComでは、IPv4 over IPv6方式にMAP-Eを採用している



MAP-E (Mapping of Address and Port with Encapsulation)



- IPv4 over IPv6カプセル化技術
- CEでステートフルな44NAPT/ステートレスなv4/v6カプセル化
- BR(VNE側)はステートレスなカプセル化のみ
- アドレスとポートマッピングがMAPに準拠

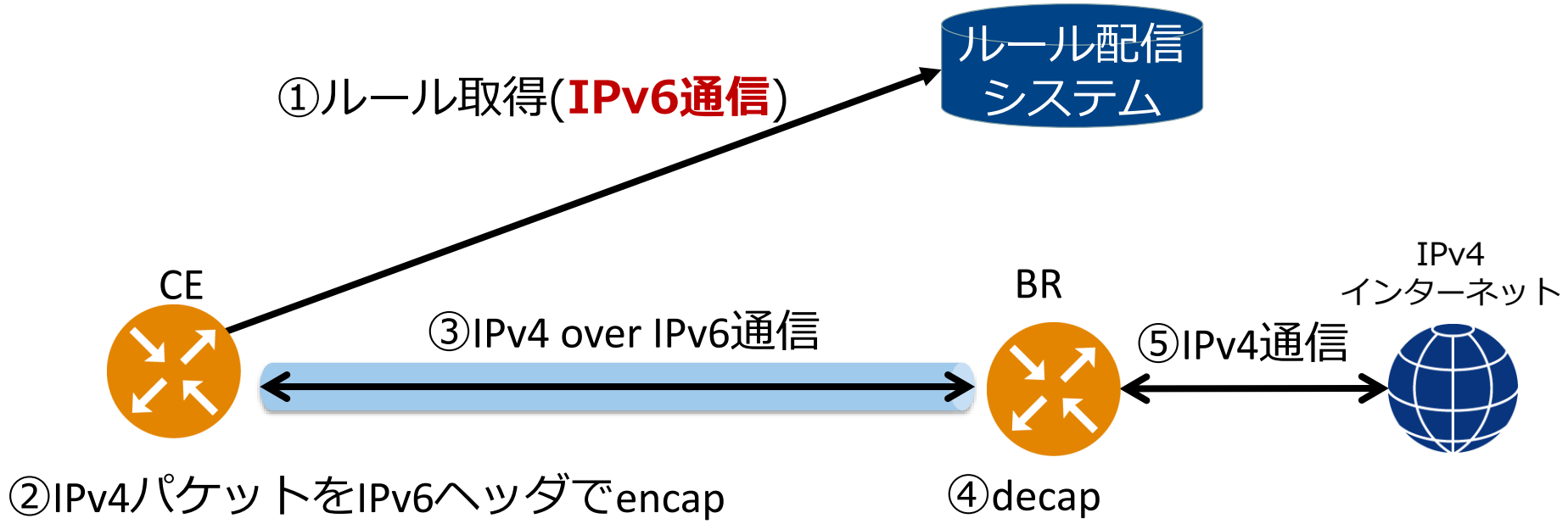
他にもIPv4 over IPv6技術はたくさんあるが割愛

BR: Border Relay

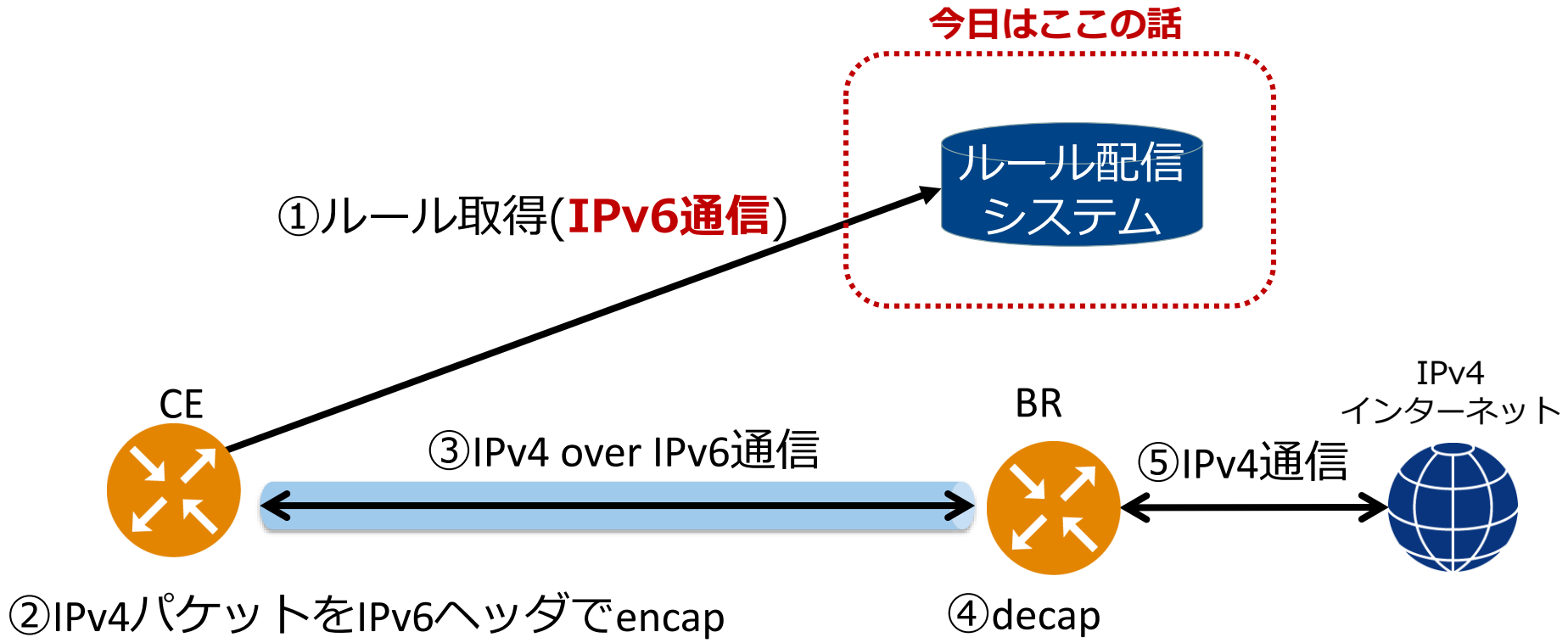
MAPはステートレスというが…

- BRとCE間は同一のMAPルールを共有する必要がある
- BRはVNE側にあるので事前プロビジョニング可能
- CEのプロビジョニングはどう実施するか？
  - rfc7598 - フレッツ仕様として存在しない
  - 残念ながら、独自で作るしかない
  - IPv6普及・高度化推進協議会 IPv6家庭用ルータSWG
    「家庭用ルータ向けIPv6移行技術プロビジョニング方式検討分科会」で共通仕様策定中

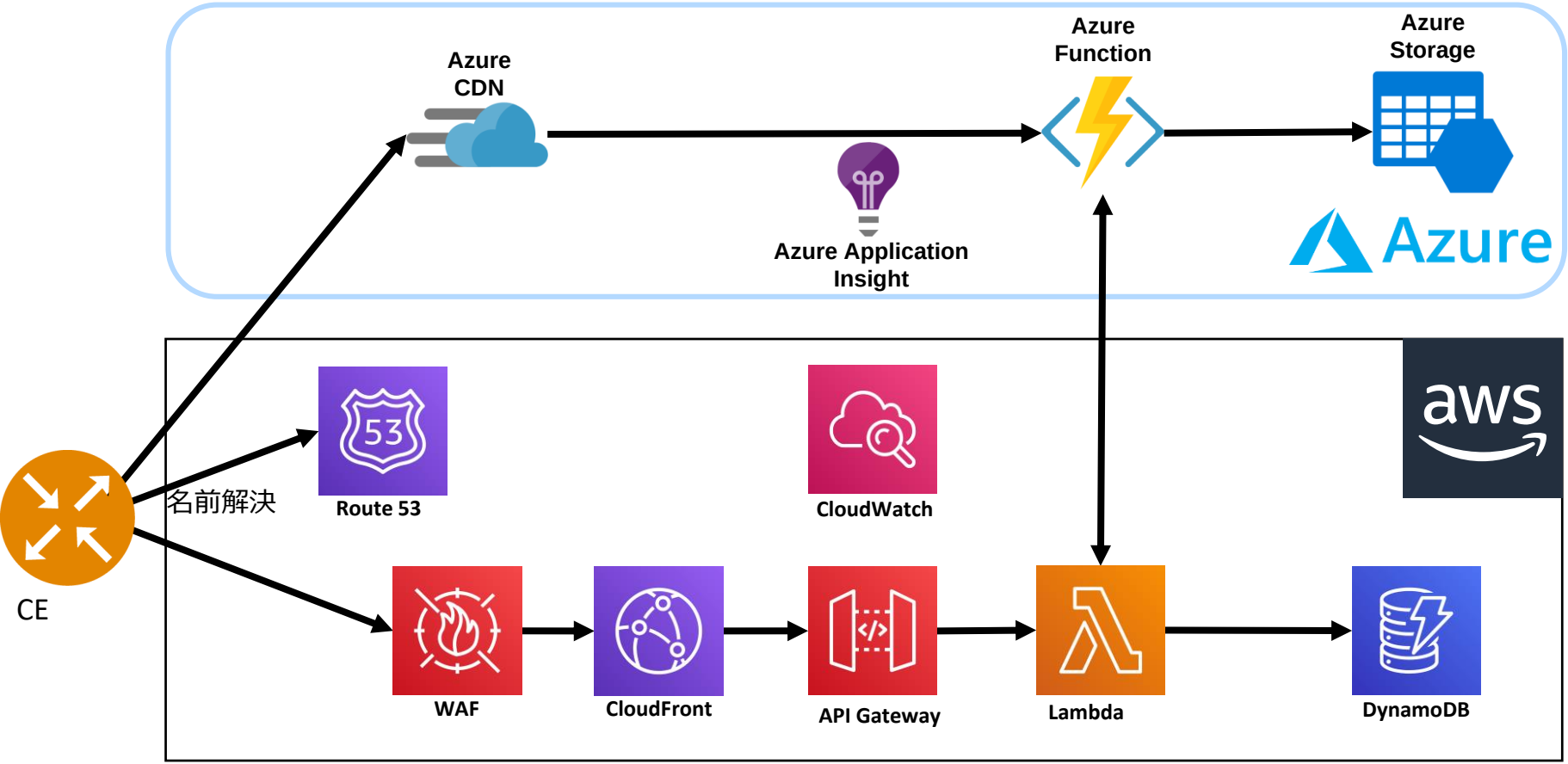
MAP-E通信を始める流れ



MAP-E通信を始める流れ



MAPルール配信システム概要 - 初期 -



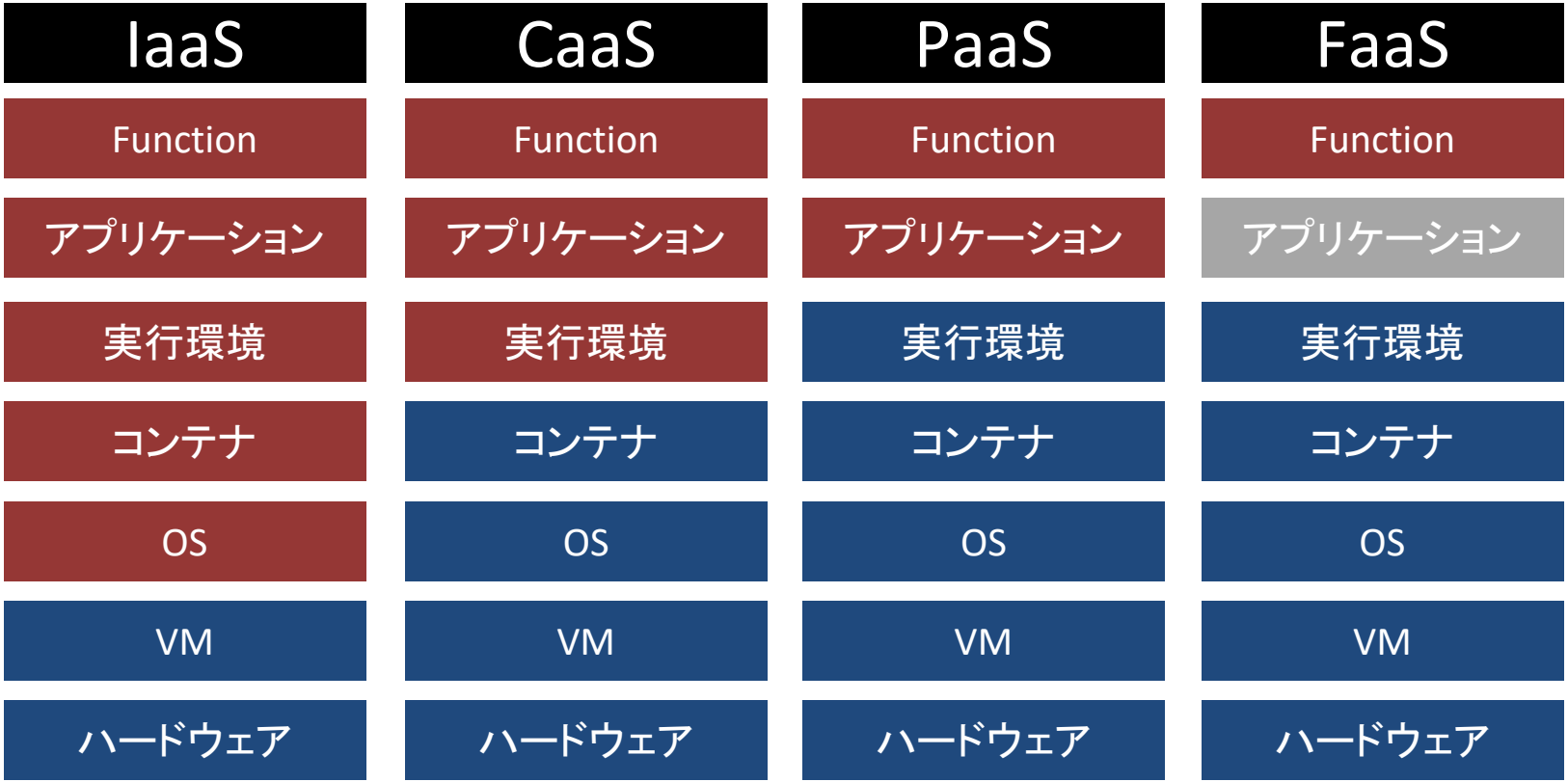
サーバレスアーキテクチャ

- ~~「サーバーが本当に不要」~~
- 「サーバーに直接アクセスしなくても仕事ができる」アーキテクチャ
- 究極の目標は、開発者がサーバーやインフラを気にかけずにコードに全力を注げるようにすること
- もたらす効果
  - スケールを気にしなくて良い（auto-scaling）
  - 利用した分のみ課金

FaaS (Function as a Service)

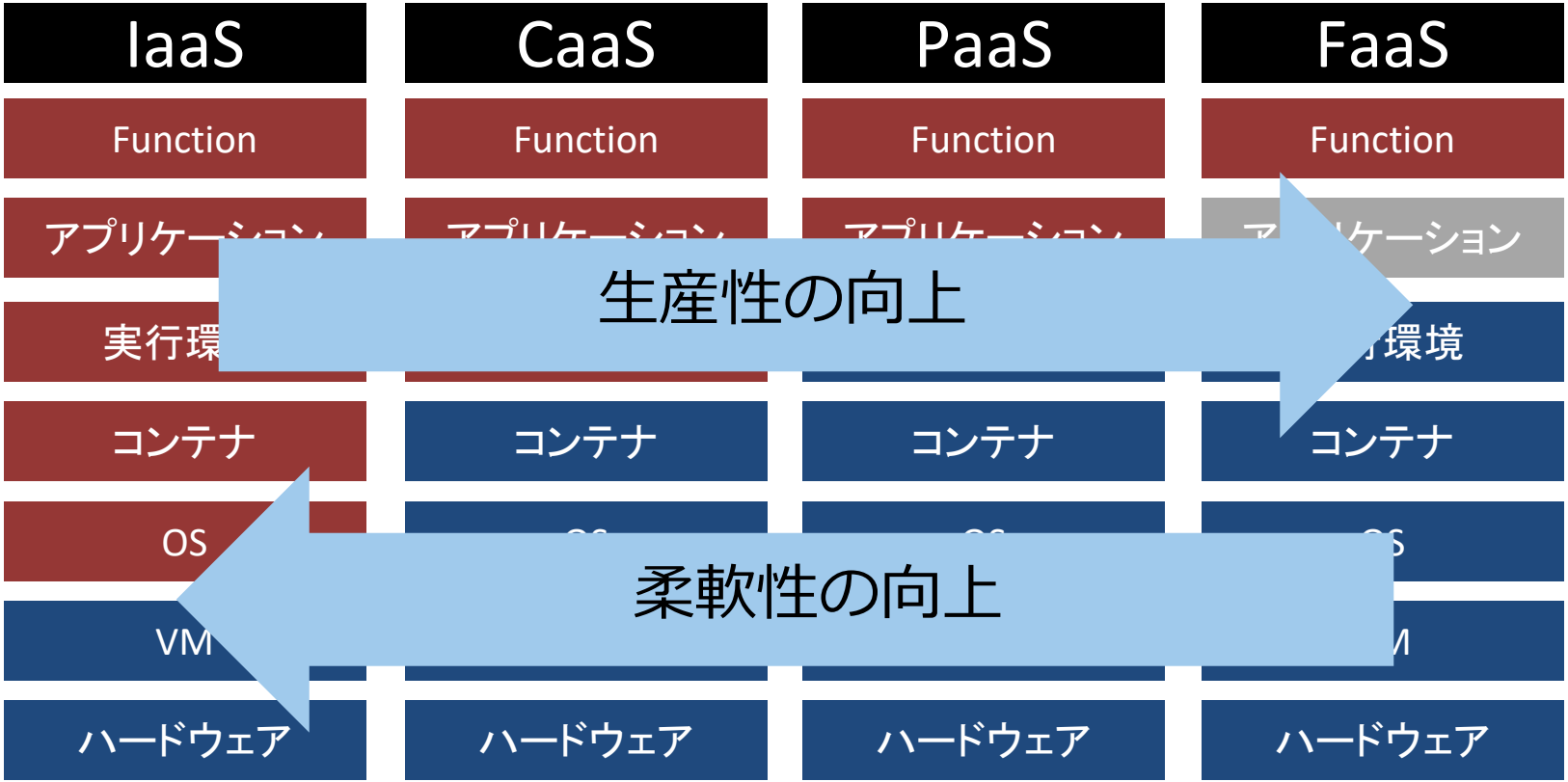
- 関数をコードで定義するだけ
- 使用した分だけ課金される
- イベントドリブン
- サーバダウンの概念がない
- ステートをもたない(≒持てない)

XaaS



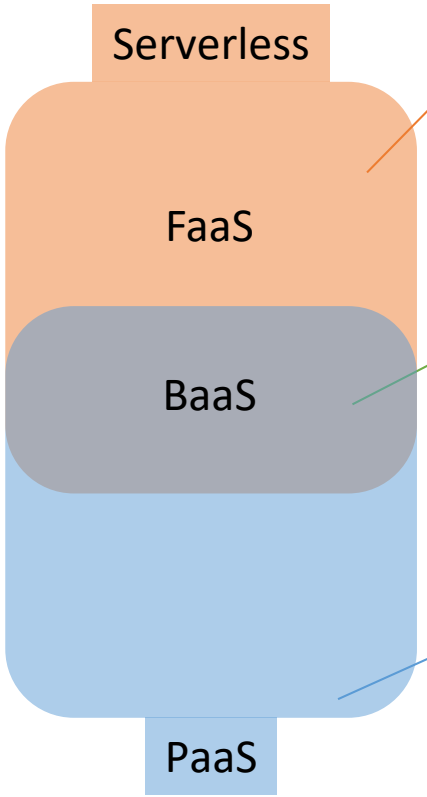
IaaS: Infrastructure as a Service
CaaS: Container as a Service
PaaS: Platform as a Service

XaaS



IaaS: Infrastructure as a Service
CaaS: Container as a Service
PaaS: Platform as a Service

サーバレスとFaaSの関係



- AWS Lambda
- Google Cloud Functions
- Azure Functions
- Open FaaS

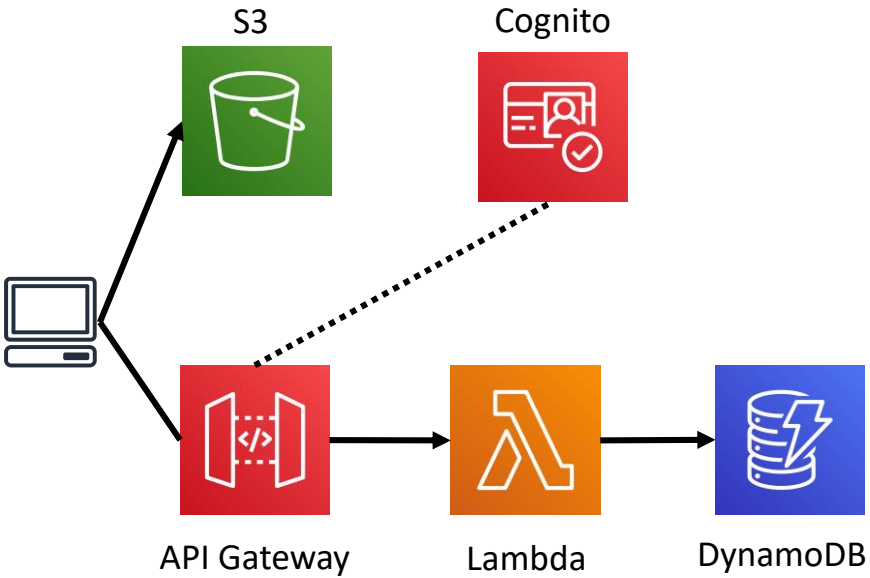
- AWS DynamoDB
- Google BigQuery
- Azure CosmosDB

- GKE
- Azure Container Service
- AWS EKS, ECS

BaaS: Backend as a Service

サーバレスのよくある構成

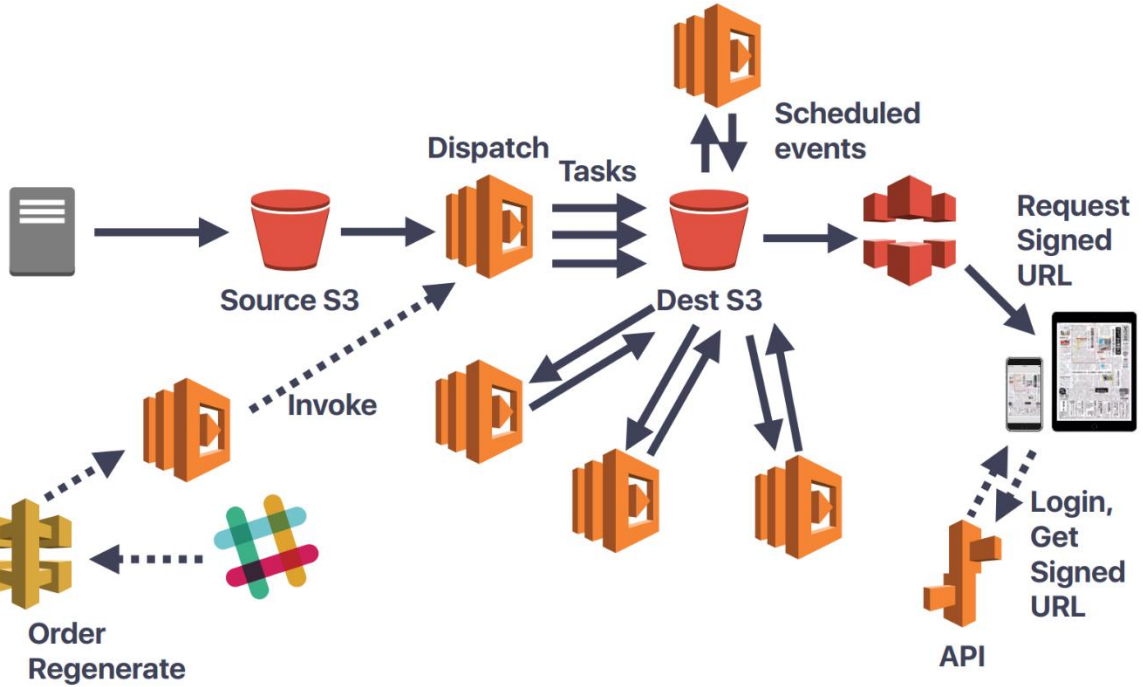
- Webサイト(SPA: Single Page Application)



- フロントエンドはS3にhtml/jsを置いて、SPA構成
- バックエンドはAPI Gateway + Lambda
- データはDynamoDBで保存
- Cognitoで認証

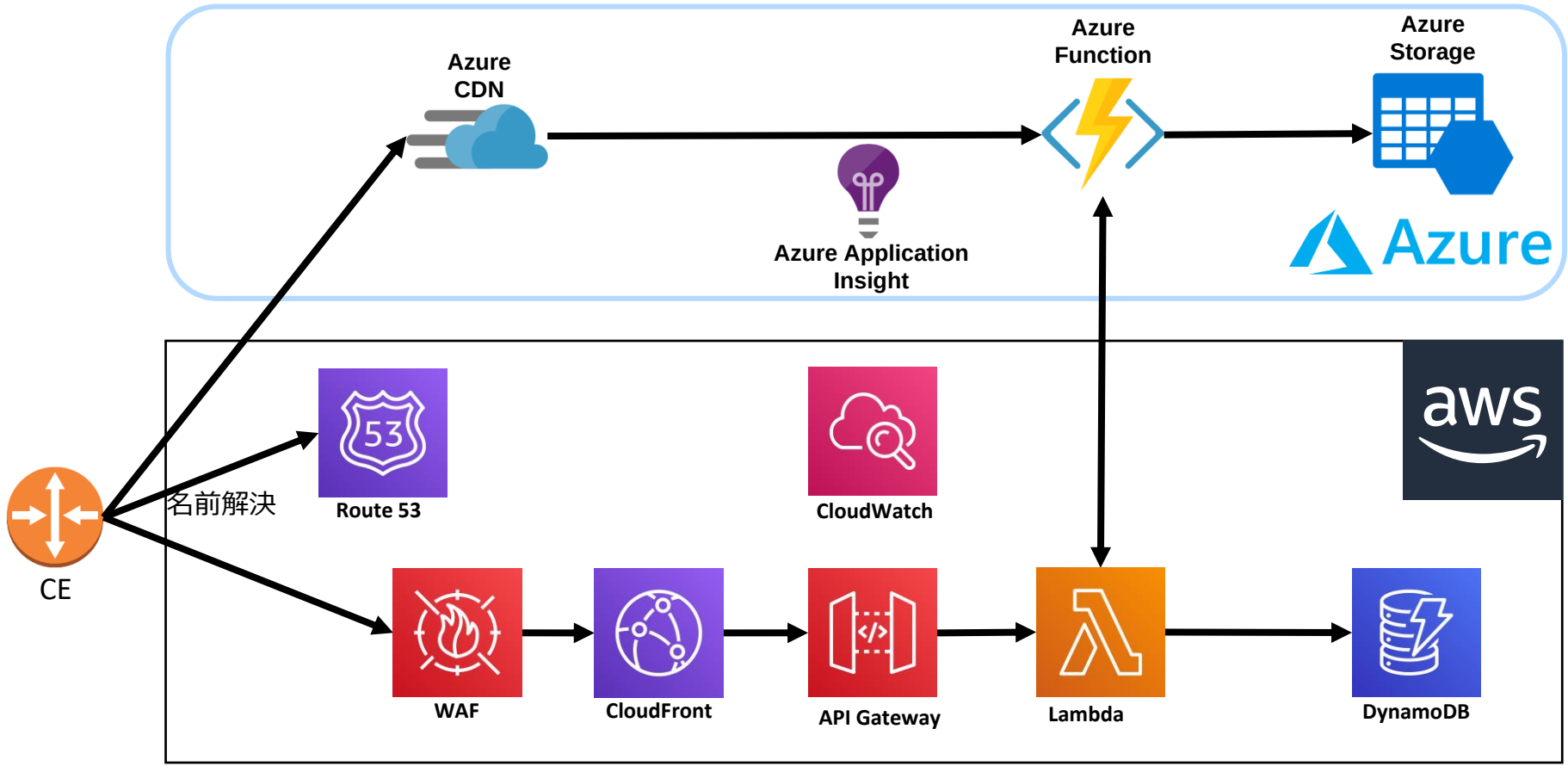
サーバレスならでは事例

■ 画像アップロードトリガー -> リサイズや切り出し等



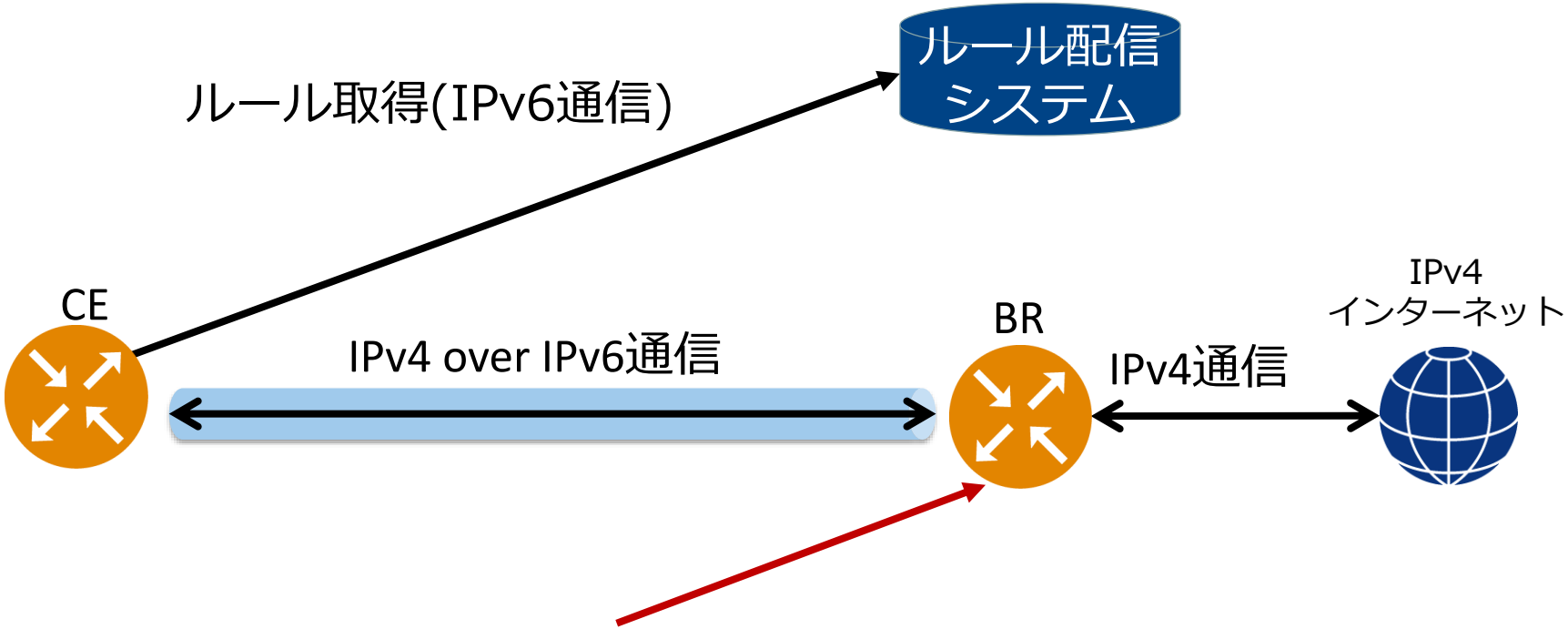
出典: 紙面ビューアーを支えるサーバーレスアーキテクチャ
(https://speakerdeck.com/ikait/serverless-architecture-supports-nikkeis-paper-viewer)

MAPルール配信システム概要 - 初期 -



なぜこんなことを始めたのか？

当時の状況



- 僕は当時BR設計・検証担当
- MAPルール配信システムは誰かが作ってくれると思ってた

誰も作ってなかった

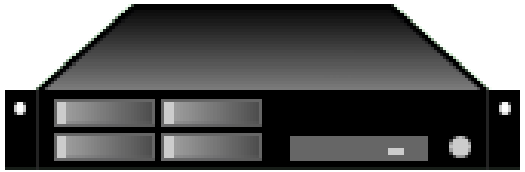
サービスリリースまで3ヶ月…

どう作るか…

- アプリケーションだし、アウトソースする？
  - 要件定義
  - アウトソース先選定
  - 意思決定会議
  - アウトソース先との調整
- 結局、時間がかかる -> 間に合わない可能性大
- 非エンジニアリング業務

自分たちで
作っちゃえ！

我々の選択肢



物理サーバ

これまでの第一選択肢
→到底間に合わない

Enterprise Cloud
自社IaaSサービス

弊社内で仮想化したかったら、これ
→IPv6未対応

aws Azure

Google Cloud Platform
他社IaaSサービス

自社サービス競合で
選択しにくかった

k8sってのもあるよね？

Google
Kubernetes Service

Amazon Elastic
Kubernetes Service

Azure
Kubernetes Service

- Kubernetesのアプリケーション初心者には難しい
- 抽象度が低いため、エンジニアのカバー範囲が大きい
- 僕たちはコンテナを管理したいのではない、Appが作りたかった

今なら
AWS Fargate/Google Cloud Run/Azure Container Instancesかな

FaaS (Function as a Service)

- 関数をコードで定義するだけ
- 使用した分だけ課金される
- イベントドリブン
- サーバダウンの概念がない
- ステートレスをもたない(≒持てない)

FaaS (Function as a Service)

- **関数をコードで定義するだけ**
- 使用した分だけ課金される
- イベントドリブン
- サーバダウンの概念がない
- ステートレスをもたない(≒持てない)

サーバレスを選択した合理性

1. 別の業務の存在
2. アプリ開発経験不足
3. 売れないかも

サーバレスを選択した合理性

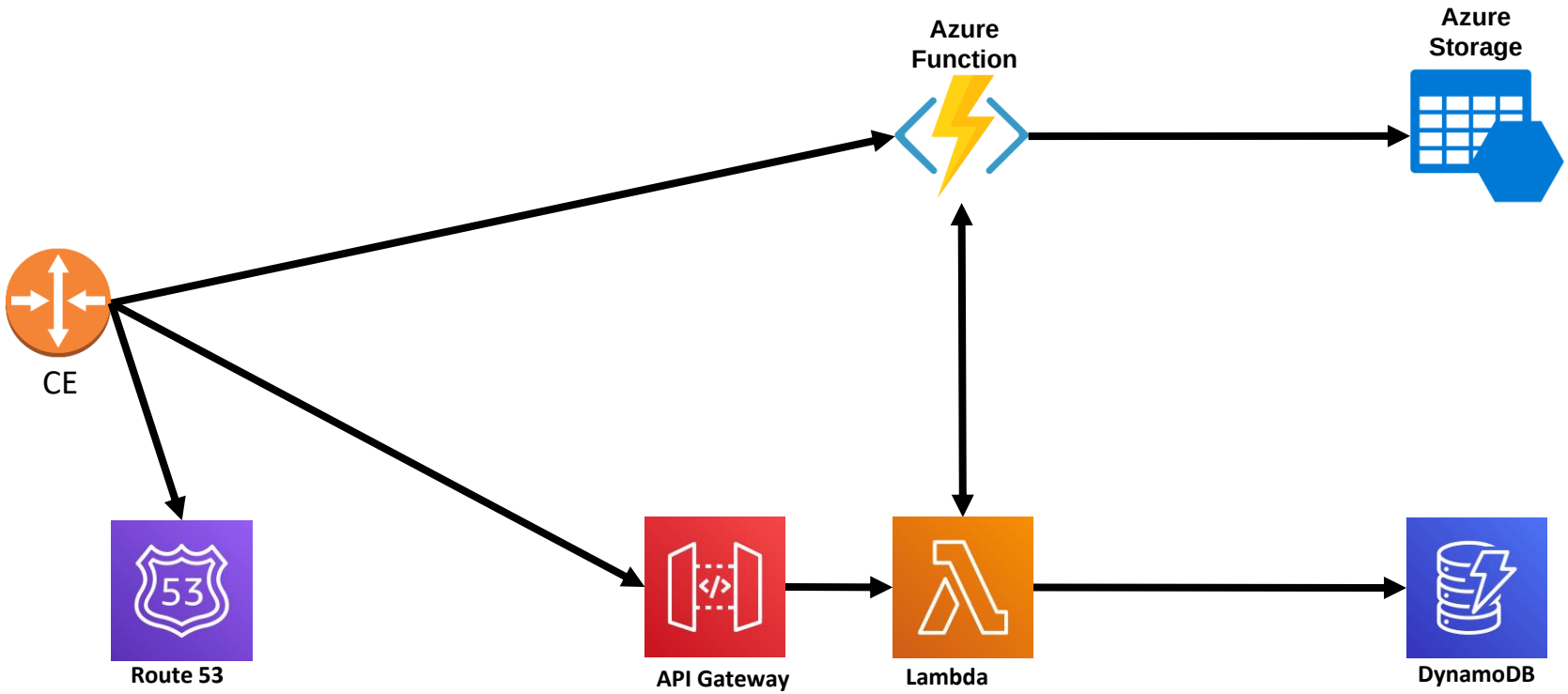
1. 別の業務の存在
2. アプリ開発経験不足
3. 売れないかも

1. マネージドサービス
2. コードのみに集中
3. 利用分のみの課金

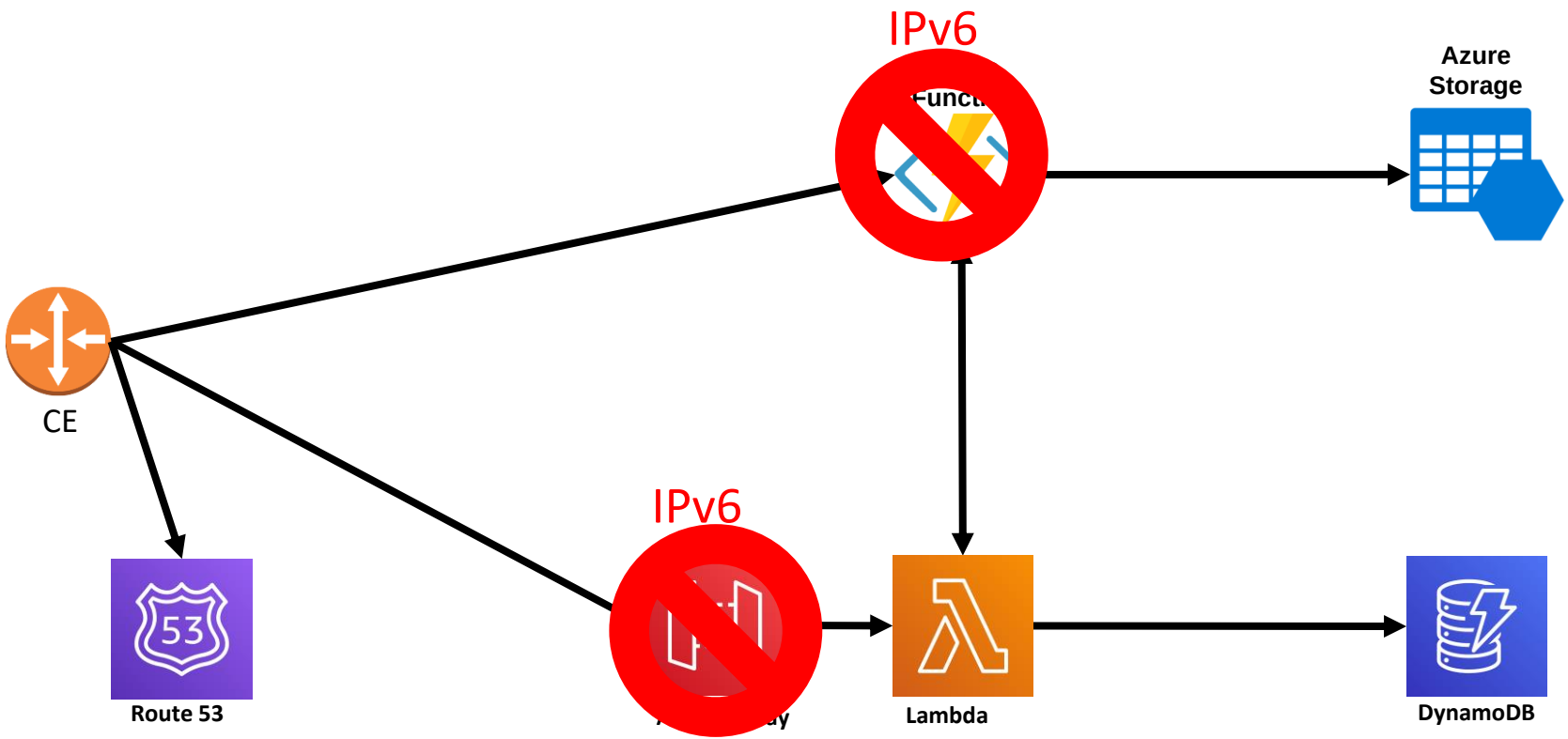
開発・運用の工夫と苦労

～システム構成の変遷に沿って～

システム構成の変遷（初期検討）

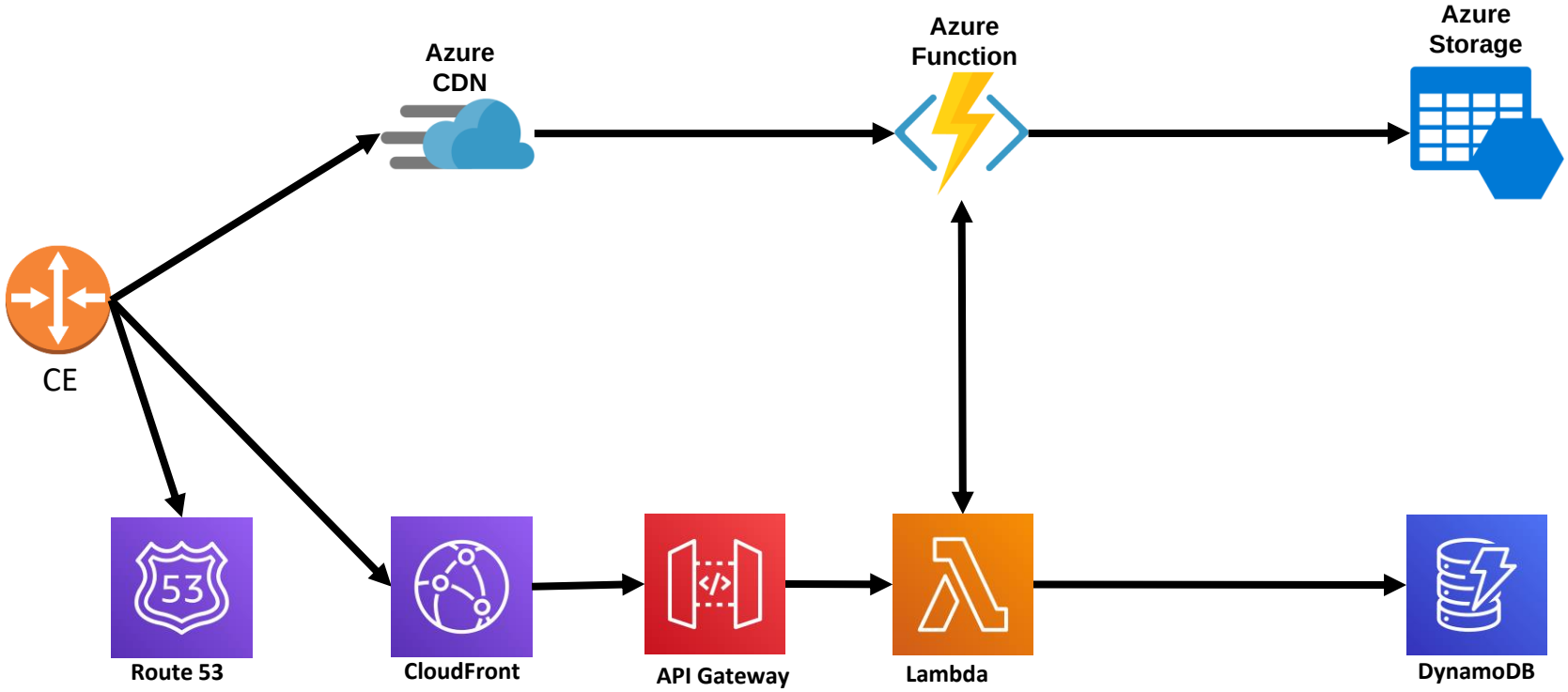


システム構成の変遷（初期検討）



システム構成の変遷（初期検討）

IPv6通信するために、CDN入れました



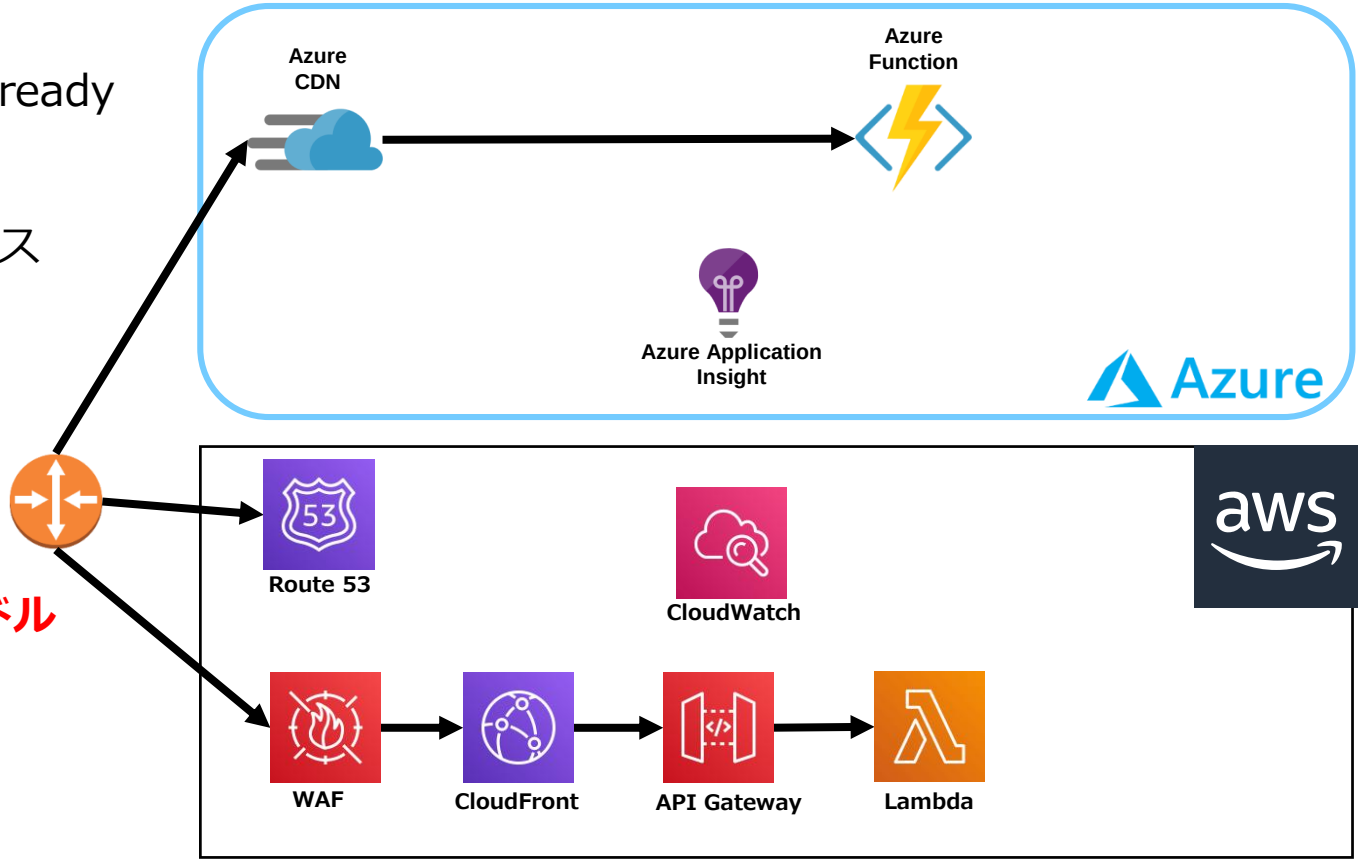
システム構成の変遷（FY2018 Q1）

Azure 単独でベータ提供
　開発・テストは1ヶ月でready
※事前調査、手続き除く

AWS も追加→正式リリース

○試行錯誤のしやすさ
　誰でも始めやすい手段

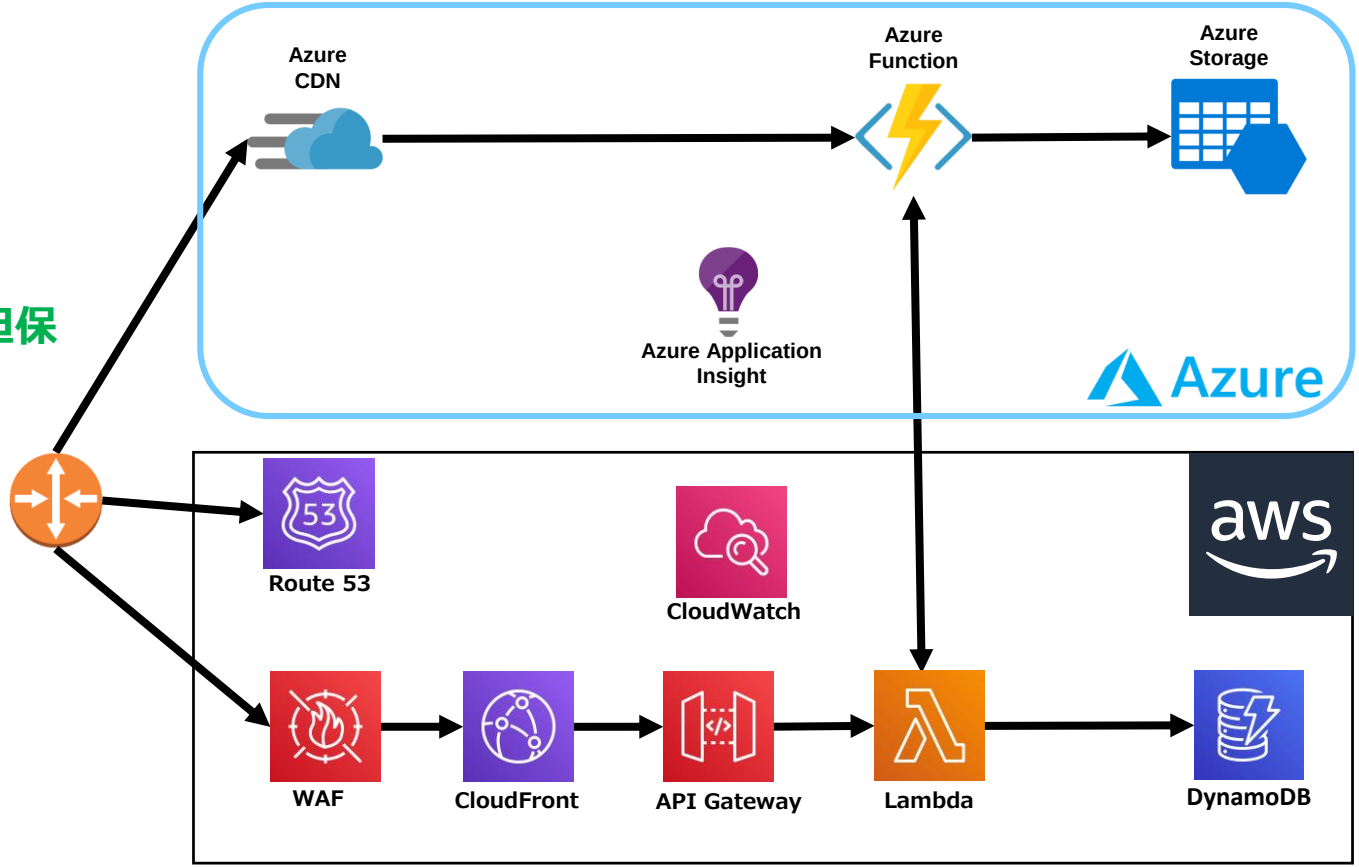
△他社クラウド導入ハードル
　社内手続き
　冗長化の圧力



システム構成の変遷（FY2018 Q2）

サービス機能追加と
クラウド間連携強化
　1.5ヶ月でリリース

システム複雑化
➔テスト自動化で品質担保



テスト自動化？
test automation?

「テスト自動化（テストじどうか）とは、テスト支援ツール等を使うことにより、ソフトウェアテストを自動化することである～」 *(Wikipedia)*

ここで大事なこと：

自分たちが書いたコードが**期待通り**動作することを
常に簡易にチェックできる機構

↓

心理的な安心

正常に動くコード

```
63      # get task
64      def get(self, id):
65          # read from DynamoDB
66          result = self.table.get_item(Key={'id': id})
67
68          if 'Item' in result.keys():
69              task = result['Item']
70              return task
71          else:
72              return None
```

テストコード

```
89  def test_GET_0(event_init):
90      event_init['httpMethod'] = 'GET'
91      event_init['path'] = '/tasks/0'
92      event_init['resource'] = '/tasks/{id}'
93      event_init['pathParameters'] = {'id': '0'}
94      event_init['body'] = None
95
96      _r = lambda_handler(event_init, context)
97
98      assert _r['statusCode'] == 200
```

テスト実行結果

```
george@gpro2016.local$ pytest
========================== test session starts ==========================
platform darwin -- Python 3.6.5, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
rootdir: /Users/george/shugyo/2019-sw-training/faas/application/backend
collected 7 items

tests/test_sls_offline.py .......                                  [100%]

========================== 7 passed in 3.89 seconds ==========================
george@gpro2016.local$
```

変更されたコード

```
63      # get task
64      def get(
65          #
66                                      : id})
67
68
69
70
71          else
72              return None
```

加えられた変更

テストコード（同じ）

```
89  def test_GET_0(event_init):
90      event_init['httpMethod'] = 'GET'
91      event_init['path'] = '/tasks/0'
92      event_init['resource'] = '/tasks/{id}'
93      event_init['pathParameters'] = {'id': '0'}
94      event_init['body'] = None
95
96      _r = lambda_handler(event_init, context)
97
98      assert _r['statusCode'] == 200
```

テスト実行結果

？？？？

テスト実行結果

```
george@gpro2016.local$ pytest -v
========================= test session starts ==========================
platform darwin -- Python 3.6.5, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -
- /Users/george/.pyenv/versions/3.6.5/bin/python3.6
cachedir: .pytest_cache
rootdir: /Users/george/shugyo/2019-sw-training/faas/application/backend
collected 7 items

tests/test_sls_offline.py::test_POST PASSED                      [ 14%]
tests/test_sls_offline.py::test_GET_0 FAILED                     [ 28%]
tests/test_sls_offline.py::test_PUT FAILED                       [ 42%]
tests/test_sls_offline.py::test_GET_1 FAILED                     [ 57%]
tests/test_sls_offline.py::test_LIST FAILED                      [ 71%]
tests/test_sls_offline.py::test_DELETE_0 PASSED                  [ 85%]
tests/test_sls_offline.py::test_GET_404 PASSED                   [100%]
```

FAILED!!!!

（エラー出力の続き）

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <src.todo.Todo object at 0x110693048>, id = '0'

    def get(self, id):
        # read from DynamoDB
        result = self.table.get_item(Key={'id': id})

        if 'Item' in result.keys():
>           task = result['FAKEFAKE']
E           KeyError: 'FAKEFAKE'

src/todo.py:69: KeyError
```

こいつだ→

テストを常に回しながら開発→異常にすぐ気付いて修正

テスト自動化の導入

■ 成功体験

- 一度テストが動けば安心して変更を加えられる
- コーディング下手くそでも自信を持てる
- 圧倒的スピードアップ

■ 気をつけたこと

- 「テスト駆動開発」はすぐ取り入れず、テスト記述は後から始めた
  ➔ まずは「動きそうなこと」の体験から始めたかった
- クラウドを「手元のMacで再現してテスト」する工夫
  ➔ 網羅性 x 速度の向上
- 継続的インテグレーション (CI) も加えてより一層効率化
  ➔ 誰が開発しても必ずテストを通る仕組みづくり

システム構成の変遷（FY2019 Q2）

追加機能の検討

発生した課題

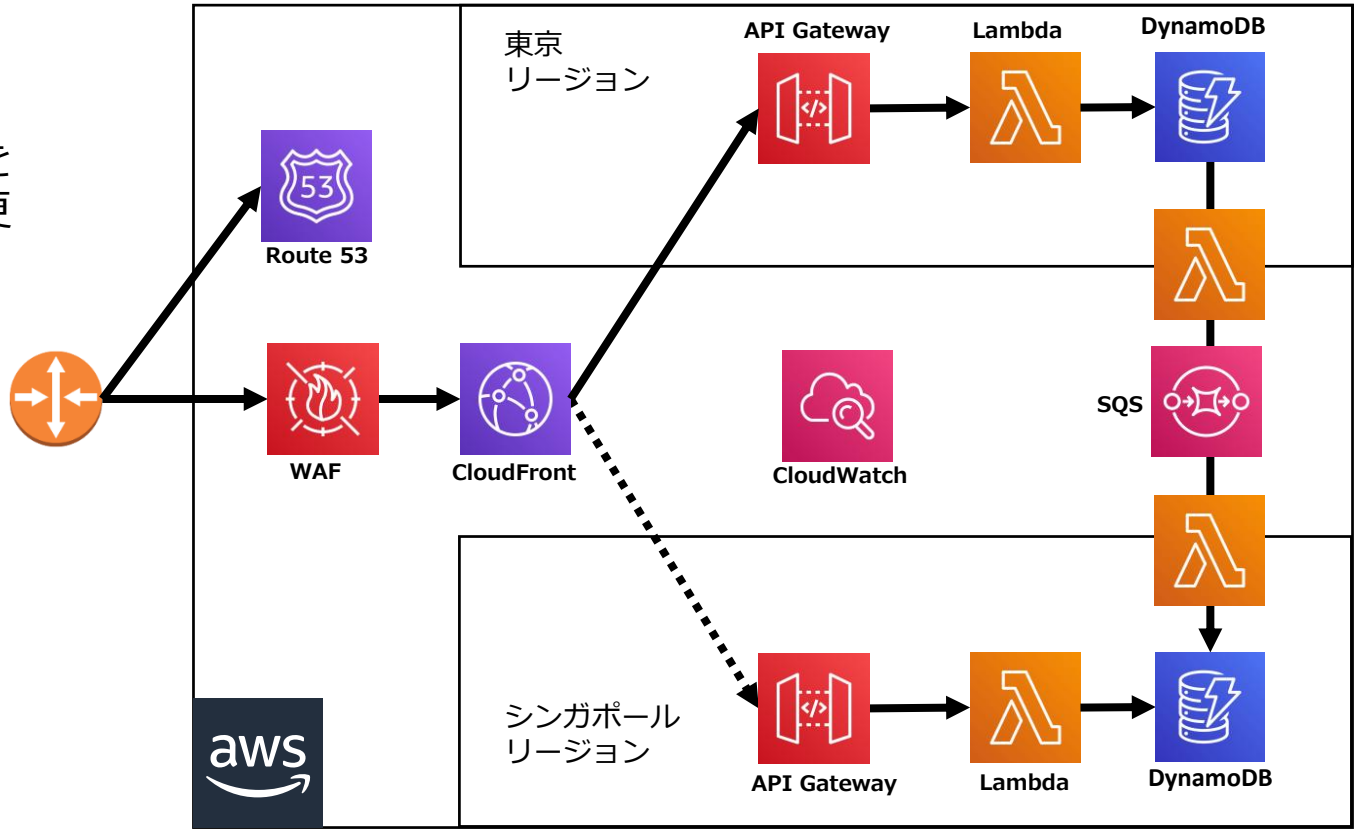
クラウド毎の固有仕様

同じ機能をマルチクラウドで維持するコスト（開発・運用）も拡大…

➔ 大幅な構成見直し

システム構成の変遷（FY2019 Q2）

大幅な構成見直し
- マルチクラウド脱却
- クラウド内で信頼性を担保できる構成に変更



マルチクラウドの難しさ

- 開発・運用の労力が1.7倍（とても概算）
  - 覚えること1.8倍　（それぞれ機能単位も特徴も違う）
  - メンテナンス量２倍　（全然オペレーション共通化出来ない）
  - メンテするコードは1.3倍くらい　（共通化に限界）

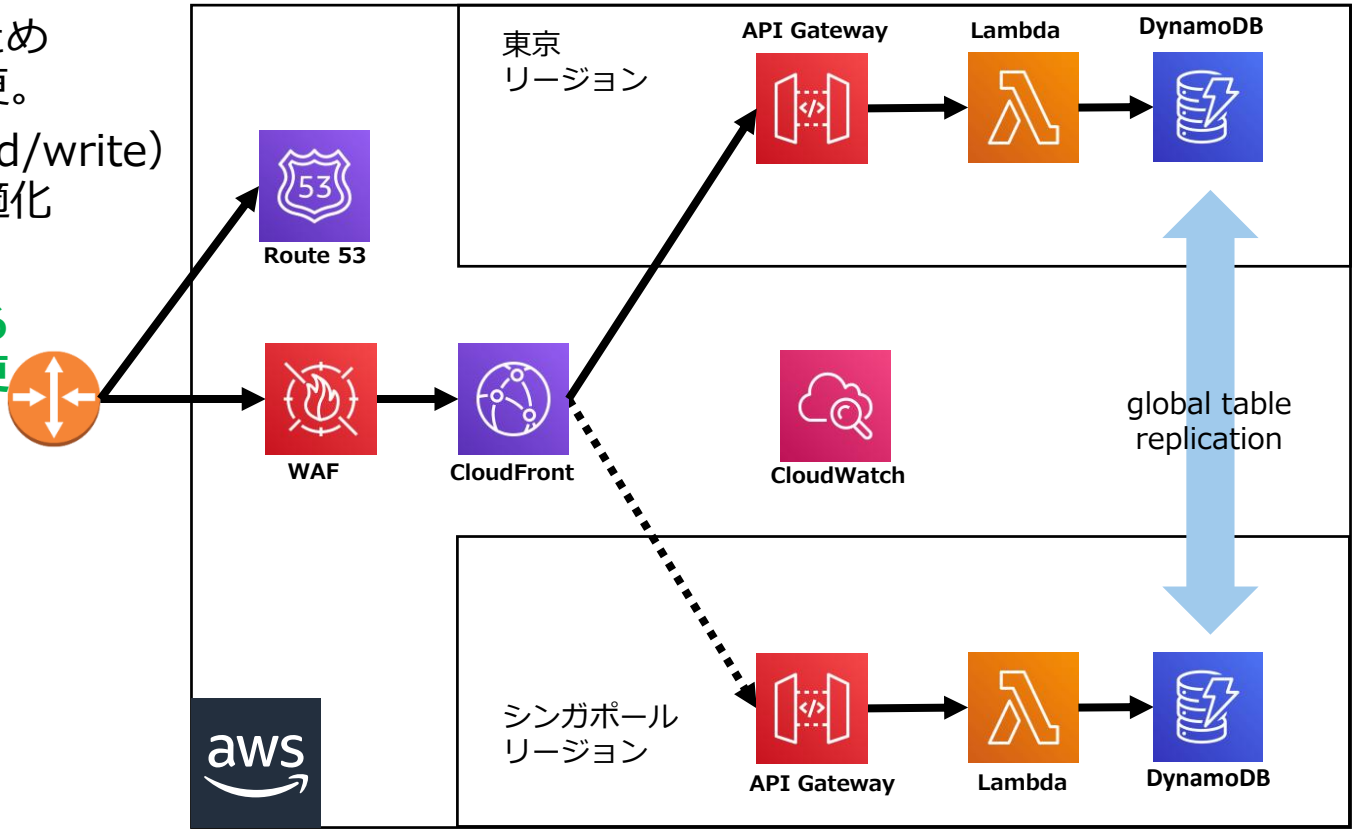
  - その他
    - ✓ クラウド間の連携・権限を適切に切る難しさ

システム構成の変遷（FY2019 Q3）

運用性・安定性 up のため
データベースも構成変更。

➔サービス無停止（read/write）
を保ちながら裏で最適化

○デプロイツールによる
堅牢・容易な構成変更



デプロイツール？
deployment tool?

デプロイツール – Serverless Framework

- 複数クラウドに対応した、サーバレスのコンポーネントをコード管理できる OSS フレームワーク

- 運用
  - yaml ファイルに構成を定義し、コマンド一発でデプロイ
    ➔ 構成変更作業で大活躍
- 開発
  - 豊富なプラグイン
    ➔**手元に各コンポーネントを再現**できるプラグインがあり、テスト駆動開発も可能

Serverless Framework: http://serverless.com/

（参考）デプロイのイメージ

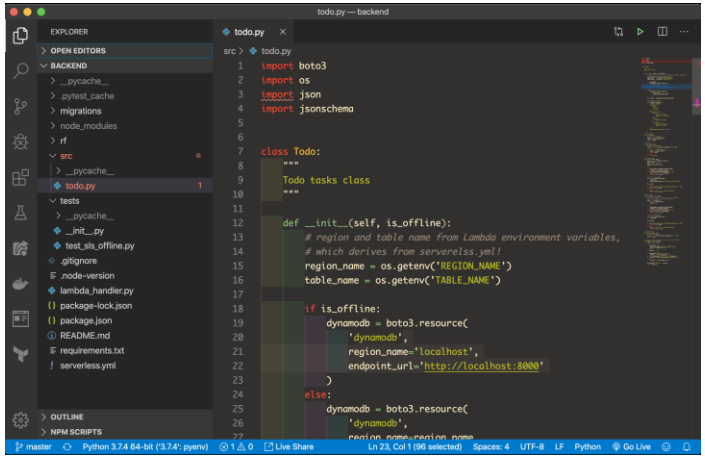
1. 使用するコンポーネント
とその設定を記述（yaml）

```yaml
Properties:
  Name: todo-api-${self:provider.stage}-sls
  Description: ${self:provider.stage} API Gateway (Lambda-proxy integration)
ApiGatewayStage:  # for apigw logging
  Type: AWS::ApiGateway::Stage
  Properties:
    MethodSettings:
      - DataTraceEnabled: true
        HttpMethod: "*"
        LoggingLevel: INFO
        ResourcePath: "/*"
        MetricsEnabled: true

functions:  # Lambda configuration
  'todo':
    handler: lambda_handler.lambda_handler
    name: todo-function-${self:provider.stage}-sls
    description: ${self:provider.stage} Lambda function
    environment:
      REGION_NAME: ${self:provider.region}
      TABLE_NAME: ${self:resources.Resources.DynamoDbTable.Properties.TableName}
    events:  # define resources one by one to save Lambda from consumption
      - http:
          path: /tasks
          method: get
          cors: true
      - http:
          path: /tasks
          method: post
          cors: true
      - http:
          path: /tasks/{id}
          method: get
          cors: true
      - http:
          path: /tasks/{id}
          method: put
          cors: true
      - http:
          path: /tasks/{id}
          method: delete
          cors: true

package:  # exclude dev files first, and then include necessary files
  exclude:
    - '**'
  include:
    - 'lambda_handler.py'
    - 'src/todo.py'
    - 'requirements.txt'
```

2. Function (Lambda等) で
動作させるロジックを記述
(python, nodejs, c++, java, etc.)



3. CLI コマンド1発で
デプロイ！

```
$ serverless deploy
```

※オプション
region,
stage,
環境変数, etc.

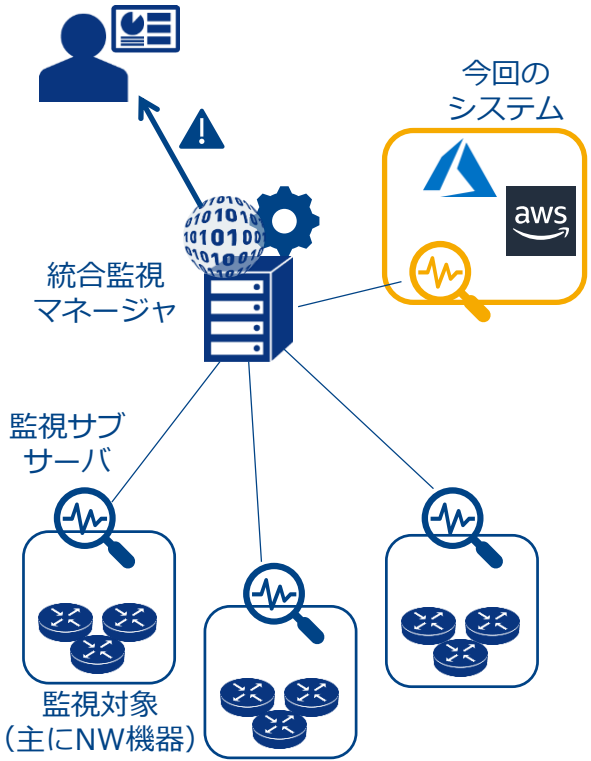
運用の話

○ほぼ障害が起きない
- AZ 障害も何のその

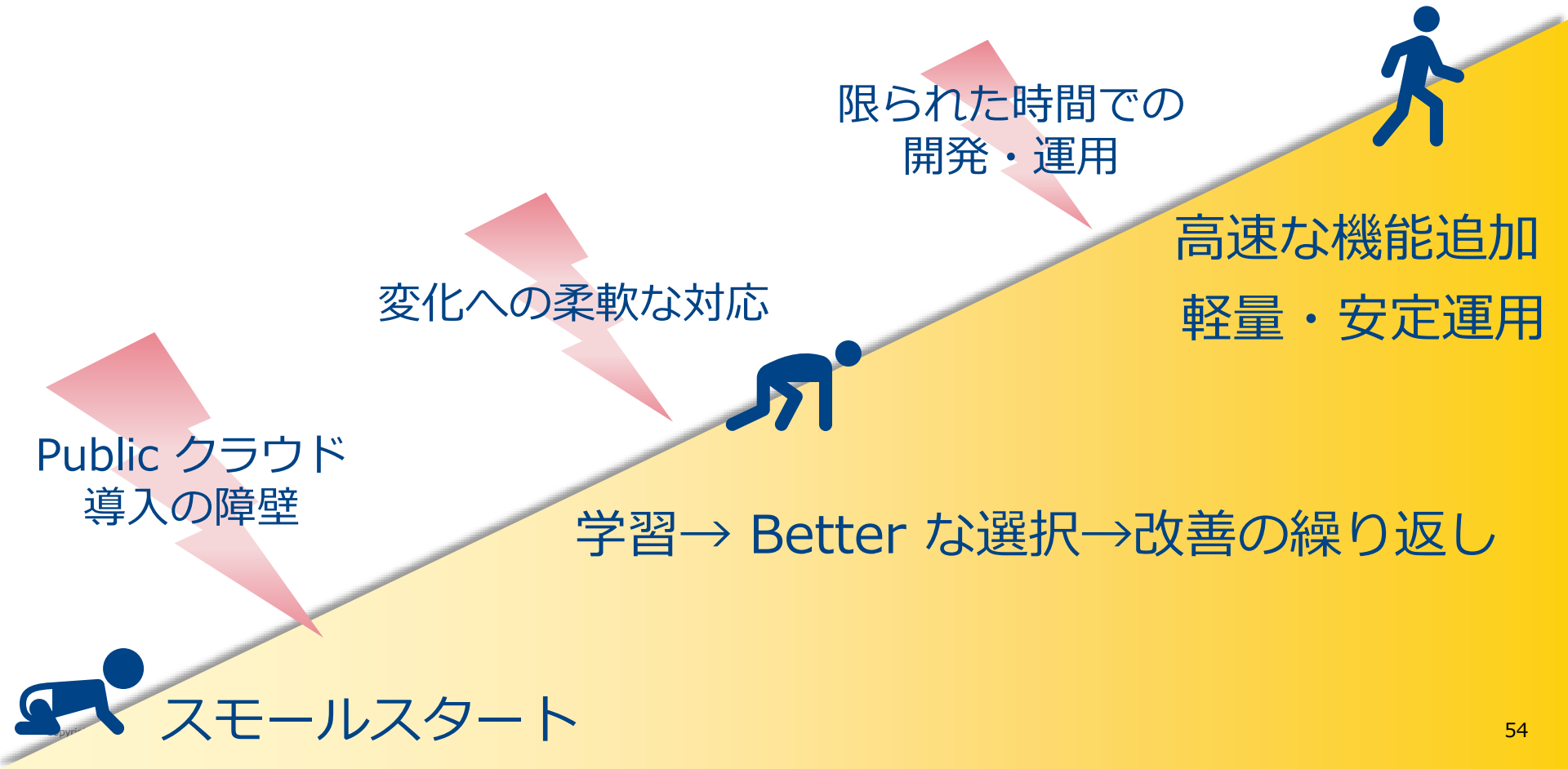
○運用稼働も極小

△少し歪んだ運用体制
- 監視統合: NW機器の監視と統合
- 1次運用の組織: NW運用のチーム

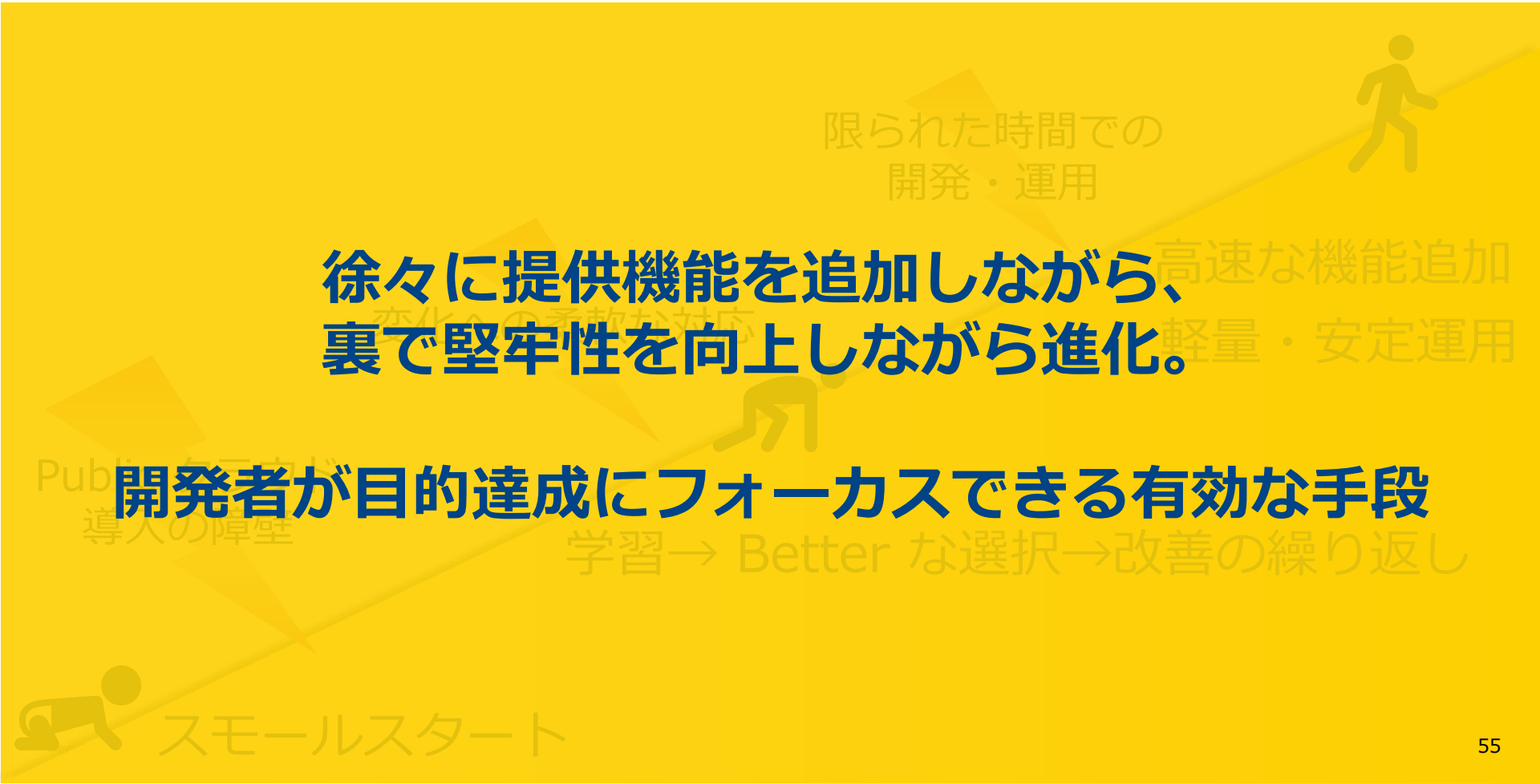
△運用メンバーの業務領域とのギャップ
- 心理的ハードル
- 障害が起きないので習熟が難しい
- たまに発生する GUI 作業の手順書化の手間



私達のサーバレスの歩み（イメージ）



私達のサーバレスの歩み（イメージ）



徐々に提供機能を追加しながら、
裏で堅牢性を向上しながら進化。

開発者が目的達成にフォーカスできる有効な手段

おわりに

- OCN の通信の一機能にサーバレスを導入したがどうだった？
  - 通常使いのクラウドであまり気にしないインフラレイヤーの要件（IPv6）
  - 単一クラウドで信頼性を確保しきれるか、力説できなかった
    - ✓ マルチクラウドで開始。
      安定運用出来ていることを確認できて、合理化へと舵を切れた

- サーバレスの使い所
  - 目的＝効能にフォーカスできる有効な思想・技術的アプローチ
  - 100% のものを自分たちでハンドルすることは既に不可能
    土台が競争力・価値とならないモノ、構成を問わないものは
    サーバレスも使い所（あるいはマネージドサービス）

Thank you.

Q&A 例

- サーバレスを採用したいか？採用できるか？
  （特に通信やその他プラットフォーム提供側で）

- 皆さんのサーバレス導入の苦労話、乗り越えたノウハウ、使い所

- その他
  - サーバレスに限らず、何らかのマネージドサービス、クラウド、他社プラットフォーム、等
  - サーバレスのディープな話