



JAWS-UG OKAYAMA 2025

laCの選定 ~AWS SAMのLambdaを ECSに乗り換えたときの備忘録~

KDDIアジャイル開発センター株式会社 あべたく



あべたく(阿部拓海) @east_takumi

出身/在住: 大分県国東市/福岡県福岡市

職種: Webバックエンド時々クラウド

コミュニティ活動

- JAWS-UG おおいた
- 福岡クラウドUG
- AWS Community Builder(Serverless)
- AWS Samurai 2023

今回の岡山でのミッション

推し(とったび)の巡った倉敷グルメを巡る



アジェンダ

1

ことの経緯

2

ECSへの移行で判明した制約と対応

3

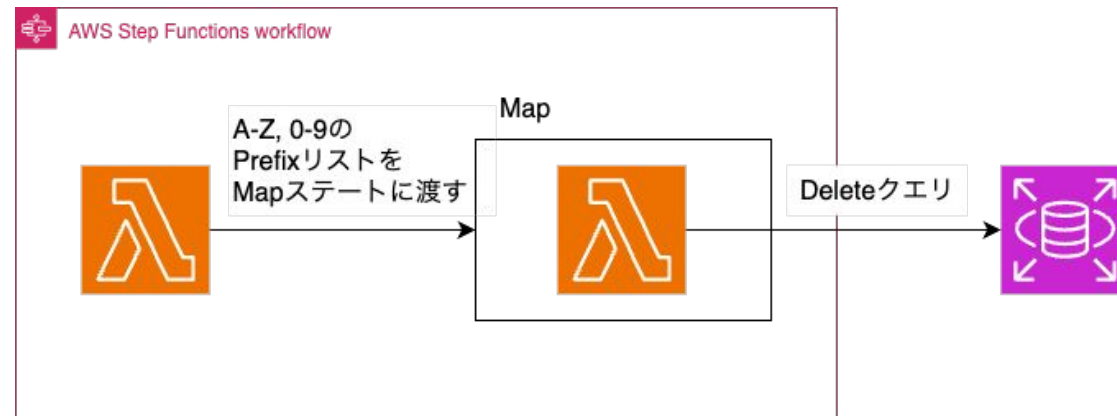
IaC選定について再考してみる

4

まとめ

そもそもの経緯/機能/アーキテクチャの(ざっくり)説明

- バッチ処理(Jenkins)をサーバレスで実装し直し
- RDSの対象となるレコードを削除するクエリをLambda生成→クエリ投下まで行う
 - LambdaとRDSの詳細な接続方式については省略
- リリース対象: Step FunctionsとLambdaのみ
 - RDSは既存リソース
- IaCとしてAWS SAMで定義



AWS SAM(Serverless Application Model)

AWS公式のサーバレス特化型IaCフレームワーク

CLIツール(sam build / sam deploy / sam local)で開発・テスト・デプロイをサポート

特徴

- Lambda や DynamoDB などサポートし、短く記述可能
- CloudFormation互換
 - SAMで足りない部分はCFnリソースを併記できる
- sam local でAPI GatewayやLambdaをローカルで模擬実行可能

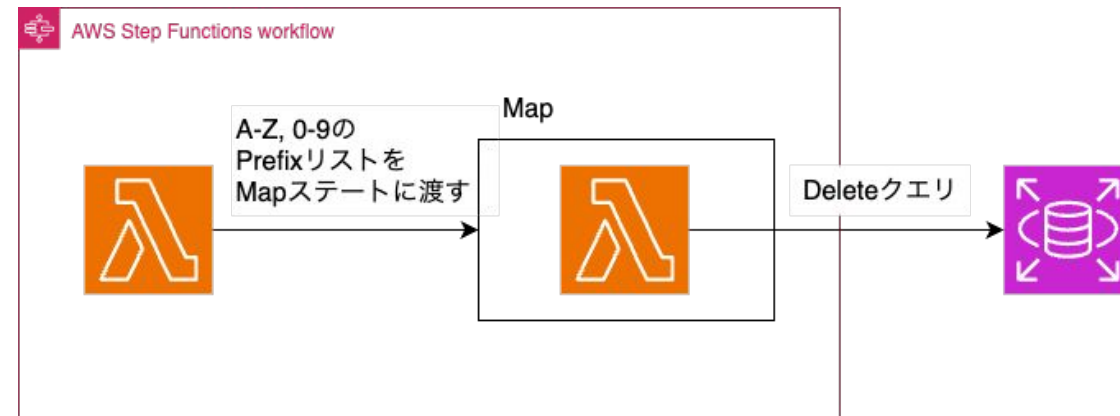


AWS SAM

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./src/
      Handler: app.lambda_handler
      Runtime: python3.9
      Events:
        ApiEvent:
          Type: Api
          Properties:
            Path: /hello
            Method: get
```

そもそもの経緯/機能/アーキテクチャの(ざっくり)説明

- バッチ処理(Jenkins)をサーバレスで実装し直し
- RDSの対象となるレコードを削除するクエリをLambda生成→クエリ投下まで行う
 - LambdaとRDSの詳細な接続方式については省略
- リリース対象: Step FunctionsとLambdaのみ
 - RDSは既存リソース
- IaCとしてAWS SAMで定義



→Map内のLambdaエラー発生！Timeout！？

そうか！

クエリで探索する範囲が広すぎるんだ！！！！

そもそもすぎて泣ける、、

しかしテーブル設計&仕様上、改善策は大規模(テーブル定義の見直し)になるため、その方策はすぐ取れない泣

そもそもの経緯/機能/アーキテクチャの(ざっくり)説明

- バッチ処理(Jenkins)をサーバレスで実装し直し

もうECS Fargate にしよう
(苦肉)

そもそもすぎて泣ける、、

しかしテーブル設計&仕様上、改善策は大規模(テーブル定義の見直し)になるため、その



ECS移行時に発生した壁

- IaCツールの制約
 - →AWS SAMがサポートするリソースにECSが入っていない
 - →ECSを使う場合はCFnで記述が必要→まあなんとかなる
- VPC設定←問題はここ
 - 新規でVPCなどを作らない場合
 - VPCの設定などはCFn Outputから取得は可能
 - →VPC設定などをTerraformで管理&Terraform OutputをCFn Outputに登録する仕組み等もなく、CFn Outputからの取得ができない
 - →VPC等の設定値を取得する仕組みが別途必要になる

Terraform

HashiCorp社が提供するマルチクラウド対応IaC

宣言的なHCL (HashiCorp Configuration Language) で記述

terraform plan → terraform apply のワークフローでリソースの差分を管理

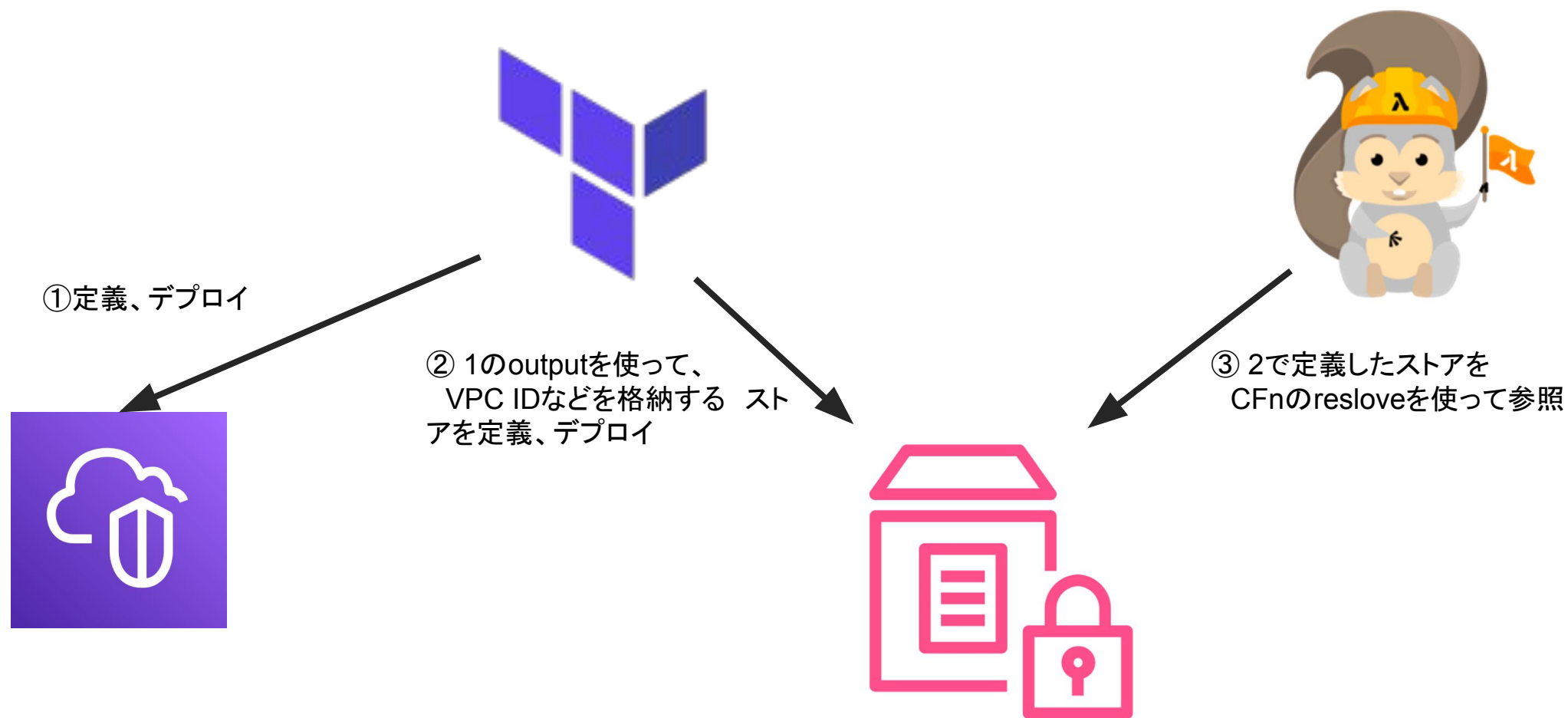
AWS, Azure, GCP, SaaSまで幅広く管理可能

tfstateでインフラの状態を記録・差分管理

Output機能でモジュール間のデータ共有やCI/CDで後続のデプロイに値を渡せたりする



対応: SSMパラメータストアからIDを取得する



対応: SSMパラメータストアからIDを取得する

CFnのreslove
(ダイナミックリファレンス)
を使って指定する

```
# ECS Service
ECSService:
  Type: AWS::ECS::Service
  Properties:
    ServiceName: !Sub "${ProjectName}-service"
    Cluster: !Ref ECSCluster
    TaskDefinition: !Ref ECSTaskDefinition
    DesiredCount: 1
    LaunchType: FARGATE
    NetworkConfiguration:
      AwsvpcConfiguration:
        SecurityGroups:
          - !Ref ECSSecurityGroup
        Subnets: !Split
          - !Sub '{{resolve:ssm:/${ProjectName}/vpc/private_subnet_ids}}'
    AssignPublicIp: DISABLED
```

それぞれの特徴比較

	AWS SAM	CloudFormation	Terraform
用途	サーバーレスアプリ (Lambda, API Gateway, Step Functions など) の構築・運用	AWSリソース全般をIaCで管理	AWS含むマルチクラウド、SaaSのIaC管理
範囲	サーバレス中心 (ECS, VPCは弱い)	全AWSリソース (最も網羅的)	AWS, GCP, Azure, Datadog, GitHub など幅広い
長所	- Lambdaなどを少ないコードで定義できる - <code>sam local</code> でローカルテスト可能 - CFnに変換してデプロイされるため AWS公式サポート範囲	- 新サービス対応が最速 - スタックで依存関係管理可能	- 再利用性が高い - OutputやRemote Stateを活用してチーム開発・マイクロサービス連携が柔軟
短所	・SAM記法はサーバレスのみ ・CFn依存なので複雑な構成は書きづらい	- 大規模だと管理が大変 - テンプレートの分割や共通化がしづらい	- <code>tfstate</code> の運用・管理が難しい - CI/CDや大規模運用を意識すると複雑さが増しやすい

選定する際に確認するポイント

1. サービス要件

- アーキテクチャがどのくらい大きいのか？
- 使用予定のAWSサービスがツールでサポートされているか？

1. チーム特性

- チームの既存知識、学習コスト、メンバーの入れ替わり具合
- 上記を考慮した開発スピード/保守性のバランス

1. 運用性

- デバッグのしやすさなど
- CI/CDパイプラインとの統合しやすさ

「プロジェクトに最適なツール」を選ぶことが重要

そして時に構成を見直すことをチームとしてする

- AWS SAMが得意なものとサポートされていない範囲
 - サーバレスを手軽にデプロイするのに使いやすい！
- 複数のIaCツールを使う際はリソース共有に注意が必要
 - SSMパラメータなどを使っていこう
- それぞれのプロジェクトにあったIaCツールを選定しよう
 - 見直す機会を定期的に作ることで、より最適化に近づける



Be a Change Leader.

アジャイルに力を与え

共に成長し続ける社会を創る