AWS Black Belt Online Seminar

AWS CloudFormation

開発・テスト・デプロイ編

山川 達也

Solutions Architect
2023/12

AWS Black Belt Online Seminar とは

- 「サービス別」「ソリューション別」「業種別」などのテーマに分け、アマゾン ウェブ サービス ジャパン合同会社が提供するオンラインセミナーシリーズです
- AWS の技術担当者が、 AWS の各サービスやソリューションについてテーマごとに動画を公開します
- 以下の URL より、過去のセミナー含めた資料などをダウンロードすることができます
  - https://aws.amazon.com/jp/aws-jp-introduction/aws-jp-webinar-service-cut/
  - https://www.youtube.com/playlist?list=PLzWGOASvSx6FIwIC2X1nObr1KcMCBBlqY

ご感想は X (Twitter) へ！ハッシュタグは以下をご利用ください
#awsblackbelt

内容についての注意点

- 本資料では資料作成時点のサービス内容および価格についてご説明しています。AWS のサービスは常にアップデートを続けているため、最新の情報は AWS 公式ウェブサイト (https://aws.amazon.com/) にてご確認ください
- 資料作成には十分注意しておりますが、資料内の価格と AWS 公式ウェブサイト記載の価格に相違があった場合、AWS 公式ウェブサイトの価格を優先とさせていただきます
- 価格は税抜表記となっています。日本居住者のお客様には別途消費税をご請求させていただきます
- 技術的な内容に関しましては、有料の AWS サポート窓口へお問い合わせください
- 料金面でのお問い合わせに関しましては、カスタマーサポート窓口へお問い合わせください (マネジメントコンソールへのログインが必要です)

本セミナーの対象者

想定聴講者

- CloudFormation の開発をしようとされている方、すでに実施されている方

前提知識

- AWS の概要を理解していること
- CloudFormation の概要を理解していること
  (AWS Blackbelt CloudFormation #1#2 基礎編 視聴済み)
- インフラ構成管理ツールのご利用経験があると望ましい（必須ではない）

ゴール

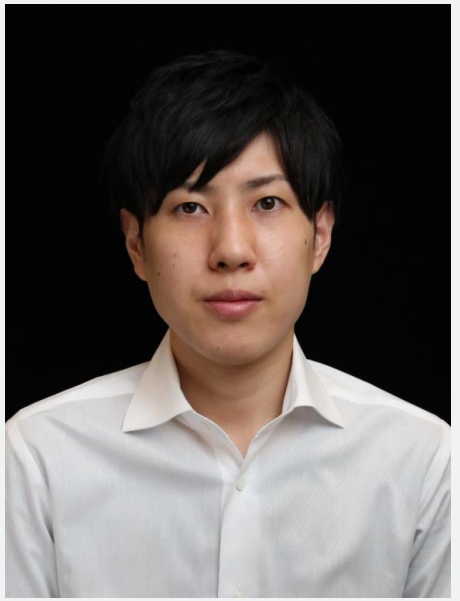
- CloudFormation の開発を進めていただく上での知識、理解を深めていただく

自己紹介

山川　達也

アマゾンウェブサービスジャパン
ソリューションアーキテクト

大手不動産業界のお客様を中心にご支援しています



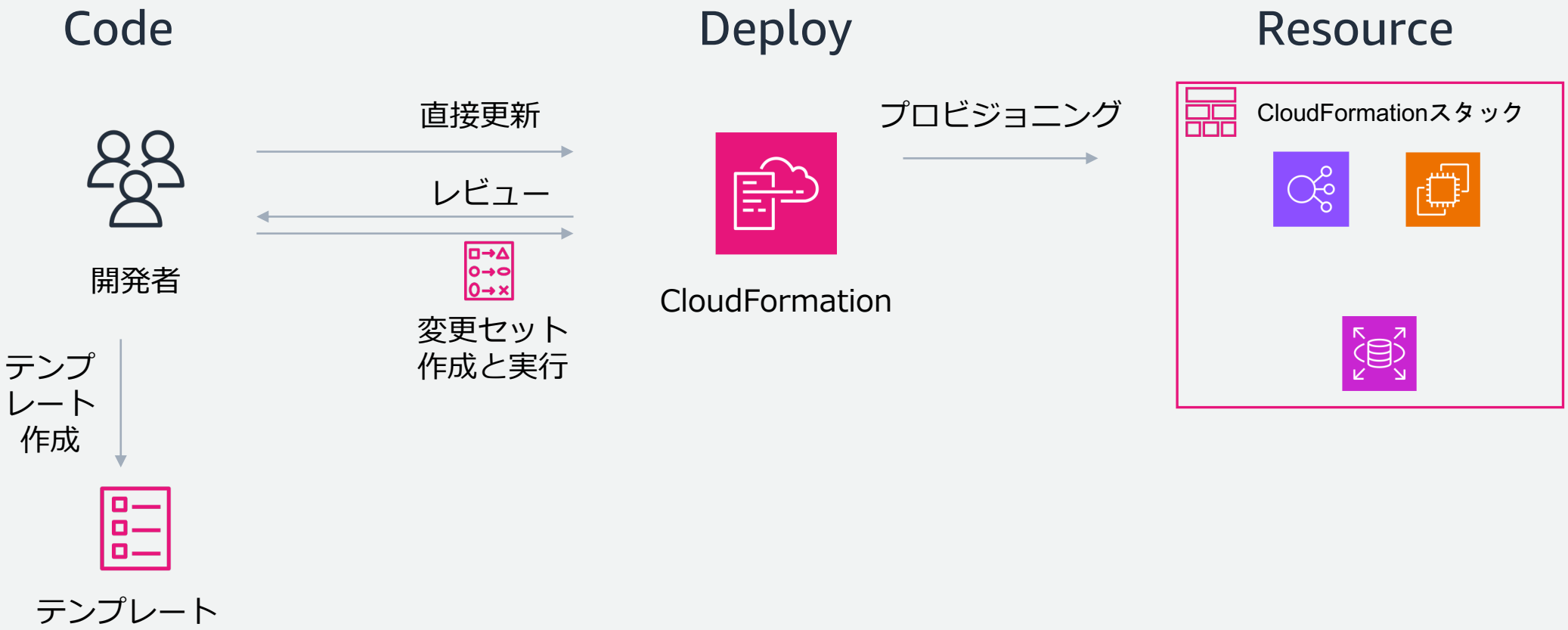
好きな AWS サービス
AWS CloudFormation

アジェンダ

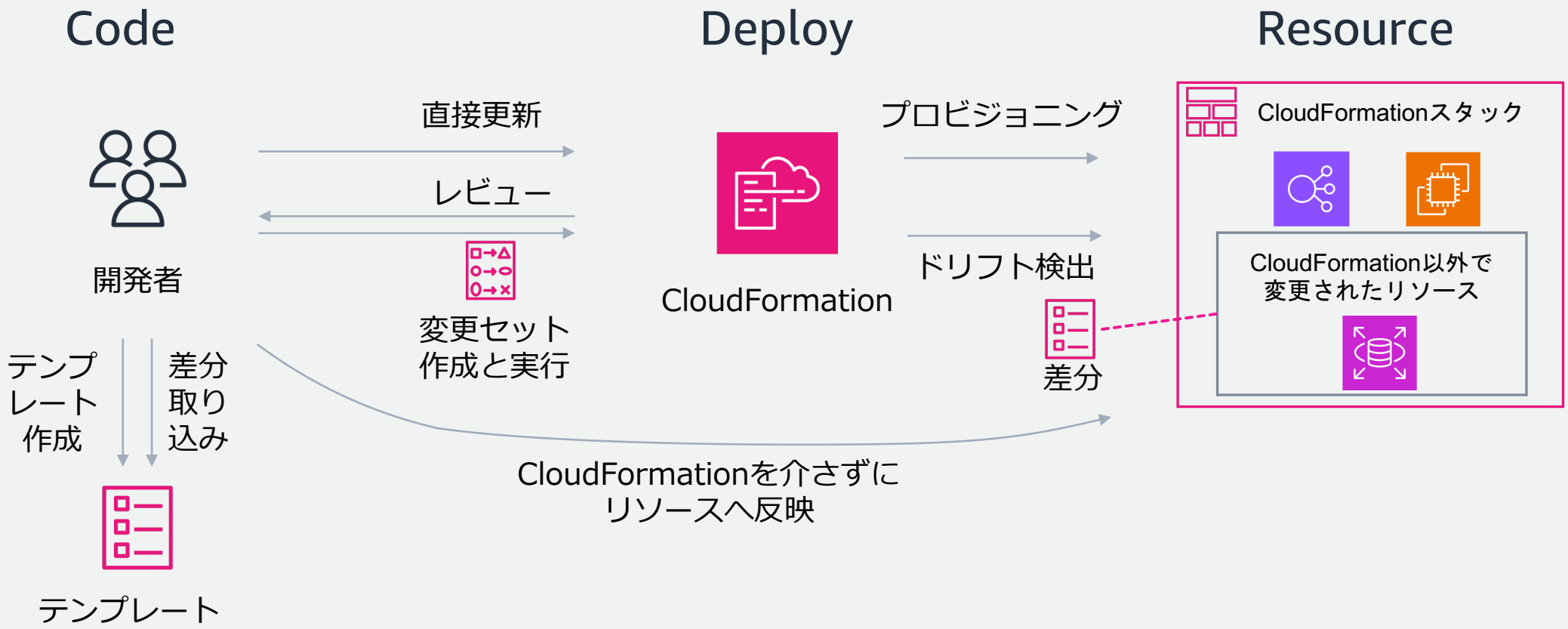
- はじめに
- CloudFormation スタックとコード設計の勘所
- 開発環境の整備
- テスト
- デプロイ

はじめに

開発者視点でのCloudFormation利用の流れ



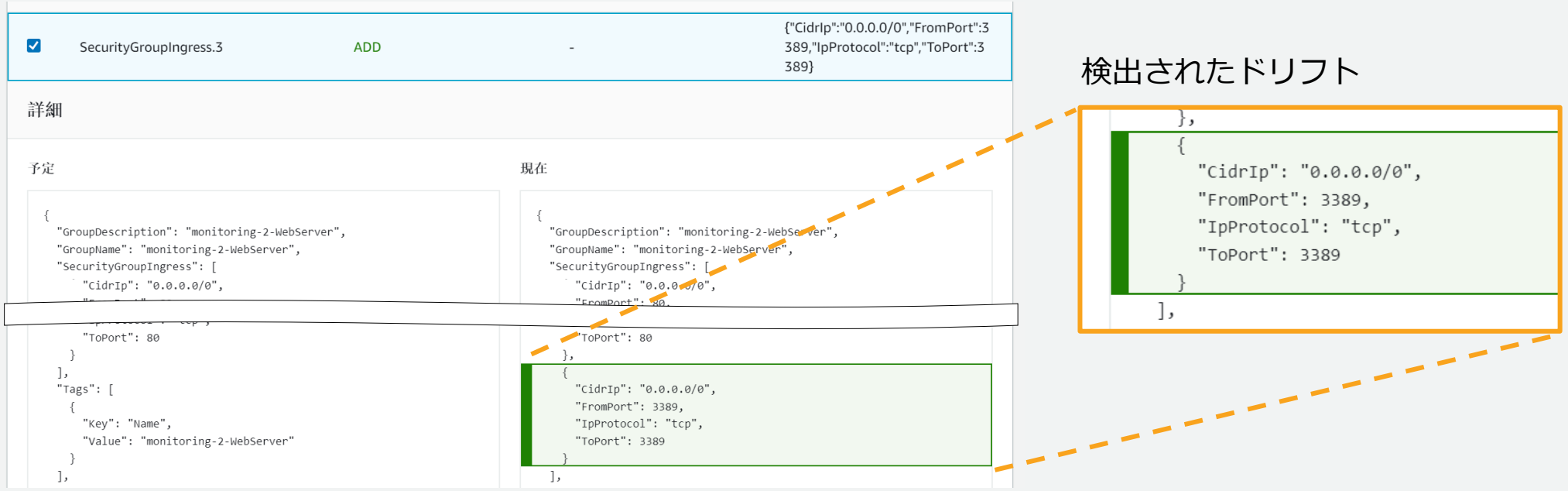
開発者視点でのCloudFormation利用の流れ



ドリフト検出

ドリフト検出機能を利用することで、テンプレート内のリソースプロパティがリソースの実際の設定と一致するかどうかを確認することが可能

ドリフトがある状態の際は、ドリフトを解消した後にスタックを更新することを推奨



CloudFormation スタックとコード設計の勘所

CloudFormation スタックの設計について

CloudFormation スタックの設計の際に、以下の観点を考慮することで、
開発や運用性の高いテンプレートの設計が可能

- CloudFormation におけるスタック分割
  - 組織やチームにあったスタック分割の進め方や方法論を整理する
- 環境ごとに再利用性可能なテンプレートの実現
  - 条件を定めて開発、検証、本番環境でテンプレートを再利用する
- 同一構成のシステム間で再利用しやすいテンプレートの作成
  - 環境依存のパラメーターをハードコーディングせずに
    再利用しやすいテンプレートを作成する

CloudFormation スタックの設計について

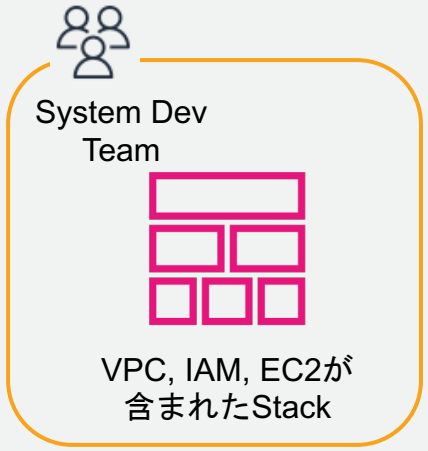
- CloudFormation 設計におけるスタック分割
- 環境ごとに再利用性可能なテンプレートの実現
- 同一構成のシステム間で再利用しやすいテンプレートの作成

CloudFormation 設計におけるスタック分割の悩み (1/2)

- 組織の体制、システム、環境、開発規模、CloudFormation のサービスクォータの上限、アカウントやリージョン跨ぎ等の理由により、スタックの分割方法について検討することがある
- スタックを分割するほど、各リソースの依存関係の管理が煩雑となるため、開発の生産性とのトレードオフ

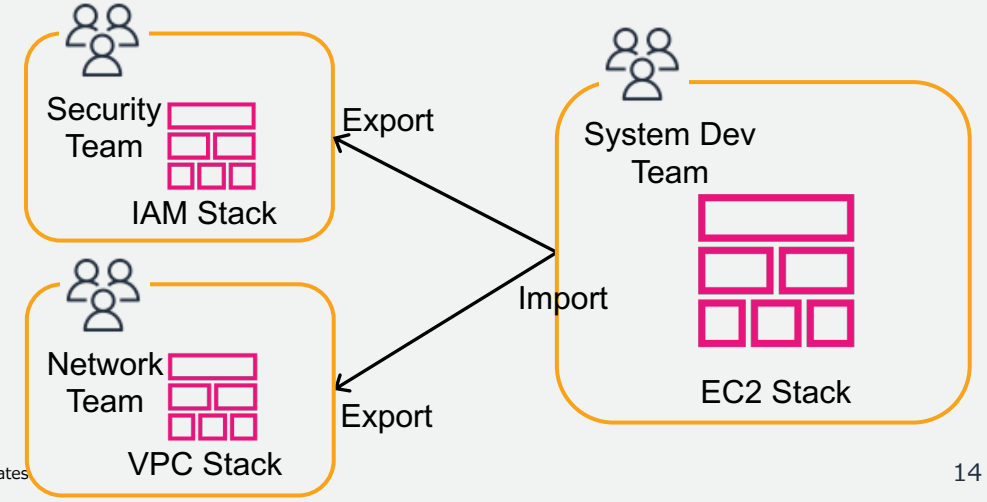
単一のスタックの場合

- CloudFormationが自動的に依存関係を解決



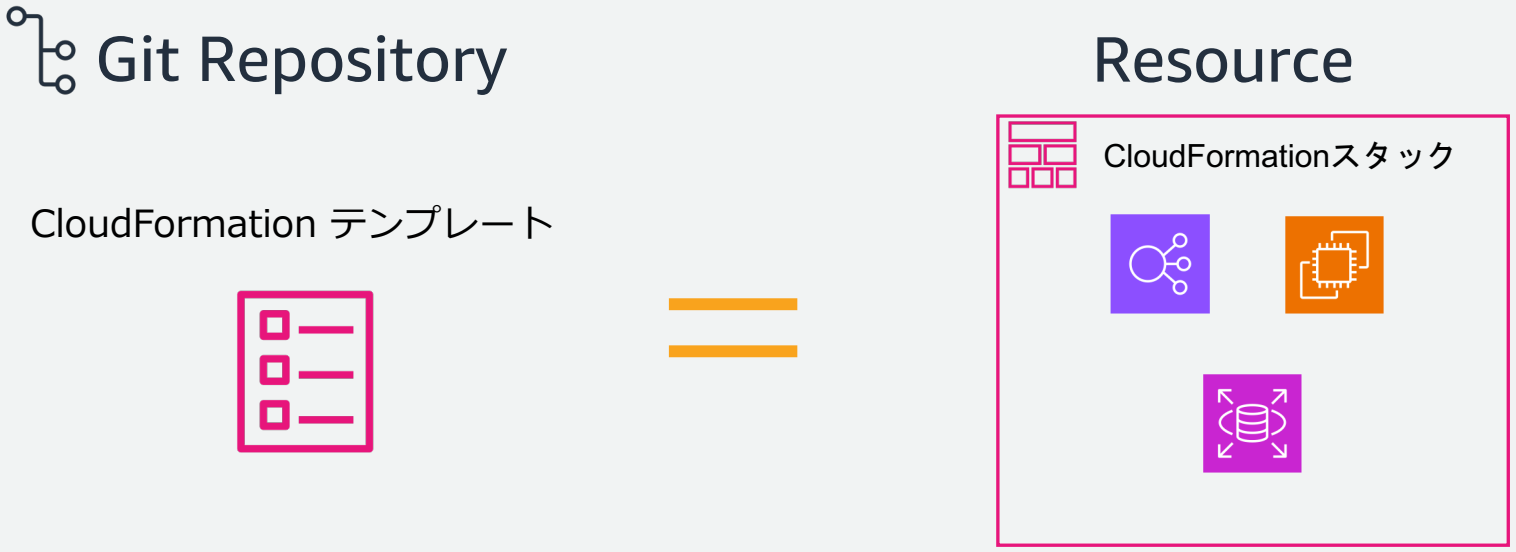
複数のスタックに分割した場合

- Exportされた値を別のスタックFn::ImportValueを用いてImportしなくてはならない
- 別スタックからImportされている値は変更することができない



CloudFormation 設計におけるスタック分割の悩み (2/2)

- Git リポジトリ内のすべての CloudFormation テンプレートを毎回デプロイすることで、Git リポジトリと実際の AWS リソースが同期された状態を維持できる
- 一部の CloudFormation テンプレートのみを選択的にデプロイすると、デプロイの一貫性と完全性が失われ、Git リポジトリを見ても実際のリソース構成がわからなくなるリスクがある
- さらに、スタックだけでなく Git リポジトリを複数に分割した場合、より広いスコープでデプロイ方法を検討する必要がある



ライフサイクルと所有権によるスタック分割

共通のライフサイクル、所有権を持つリソースでグループ化し、スタック分割を検討する手法

ライフサイクルによる分割メリット

- デプロイの影響範囲を最小化できる

ライフサイクルによる分割デメリット

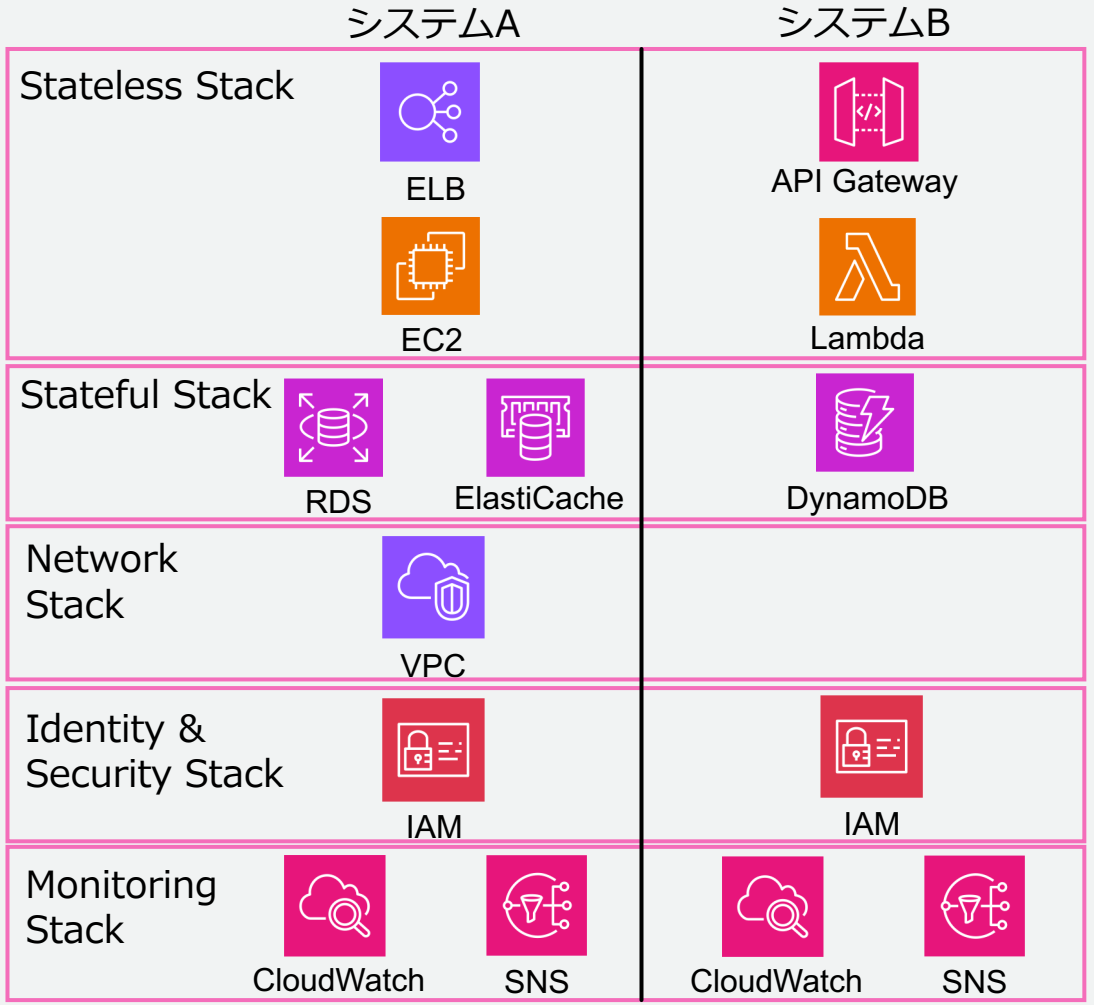
- 同一リソースにおいてもユースケースによってライフサイクルが異なる可能性があり、分類が難しい

所有権による分割メリット

- 開発者、チームによる責任範囲が明確になる

所有権による分割デメリット

- 開発におけるコミュニケーションコストが増える
- 所有者間で戦略やルールを浸透させるためのコストが増える



アプリケーション内の機能や役割によるスタック分割

クロススタック参照をできるだけ少なくすることを目標としてスタック分割を検討する手法

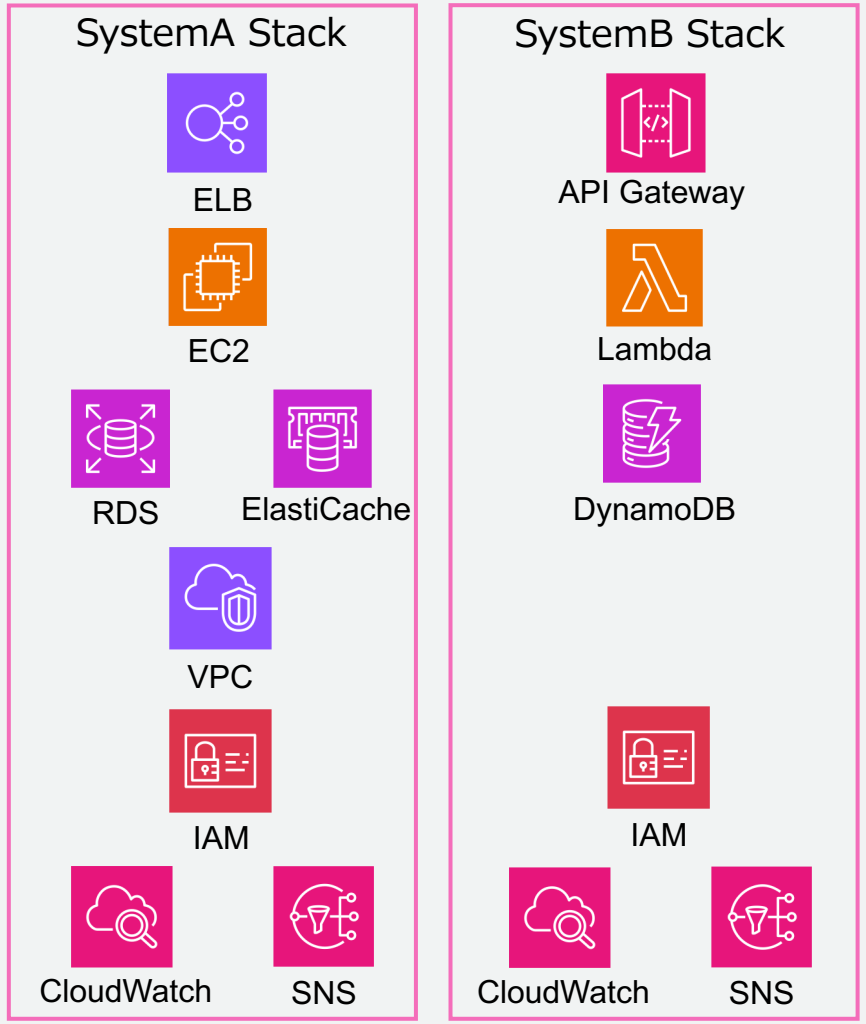
メリット

- システムに必要なリソースを1つのテンプレートに記載することで、クロススタック参照を少なくすることができ、開発・運用がしやすい
- 責任の範囲をシステムごとに設定できることから、分割の範囲を定めやすい
- 開発におけるコミュニケーションコストが減る

デメリット

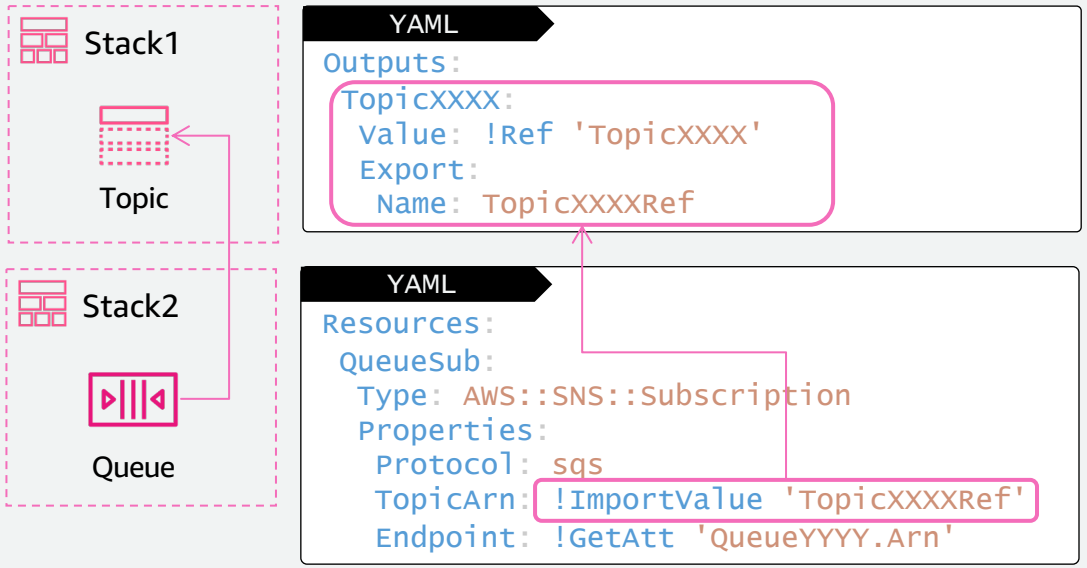
- 1つのテンプレートファイルが長くなり、可読性が低下する
- CloudFormation のリソース数やテンプレートのサイズの上限に抵触することがある

*CloudFormation のサービスクォータの上限を超えるケースにおいては、ネストされたスタックを利用した分割の検討ができる



CloudFormation における強い参照と弱い参照

強い参照

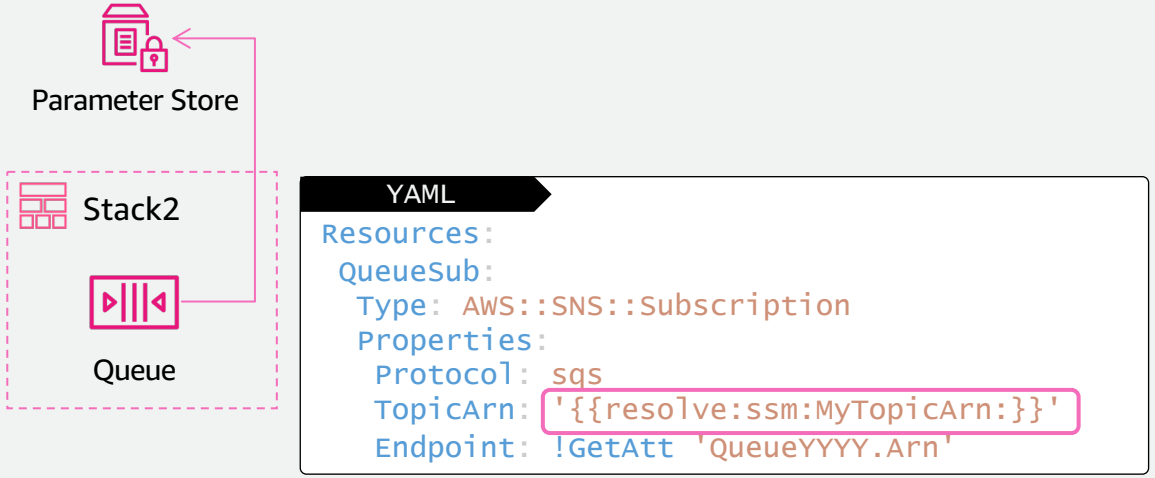


Export した値を `!ImportValue` で参照すると CloudFormation は参照しているスタックを記録 * し被参照リソースが誤って削除さないようにガードする

* `$ aws cloudformation list-imports` で確認可能

→ 安全なため推奨されるがデプロイ順序の考慮が必要

弱い参照



動的参照は CloudFormation がスタックのデプロイ中に解決するが、参照しているスタックは記録しない

CloudFormation の Parameter による値渡しも同様

→ 被参照リソースを制約なく削除可能。ただし、壊れても気づきづらい

CloudFormation スタックの設計について

- CloudFormation 設計におけるスタック分割
- 環境ごとに再利用性可能なテンプレートの実現
- 同一構成のシステム間で再利用しやすいテンプレートの作成

環境ごとに再利用可能なテンプレートの作成

- 各システムにおいて、本番と同様の構成を開発や検証環境に複製し、本番と差分の少ない環境でテストを実施することが求められる
- Parameters セクションの入力値に応じて、Mappings や Conditions により設定をコントロールすることで、テンプレートを各環境で再利用可能にすることが可能

Mappings を利用した再利用可能なテンプレートの作成

Parameters セクションで環境情報を入力し、Mappings セクションで各環境に応じた値とパラメーターをマッピングさせることで、環境ごとに設定を変更することが可能

```yaml
Parameters:
  EnvType:
    Description: 'Specify the Environment type of the stack.'
    Type: String
    Default: Test
    AllowedValues:
      - Test
      - Prod
    ConstraintDescription: 'Specify either Test or Prod.'

Mappings:
  EnvToInstanceType:
    Test:
      InstanceType: t3.micro
    Prod:
      InstanceType: m6i.large
```

Parametersで指定したEnvTypeとインスタンスタイプの値をマッピング

```yaml
Resources:
  WebServerInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiID
      InstanceType: !FindInMap
        - EnvToInstanceType
        - !Ref EnvType
        - InstanceType
```

マッピングした値は !FindInMap関数で取得 EnvType に応じてインスタンスを構築

Conditions を利用した再利用可能なテンプレートの作成

Parameters セクションで環境情報を入力し、Conditions セクションで条件分岐させることで環境ごとに設定を変更することが可能

```yaml
Parameters:
  LatestAmiId:
    Type: AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>
    Default: /aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2

  EnvType:
    Description: Specify the Environment type of the stack.
    Type: String
    AllowedValues:
      - test
      - prod
    Default: test
    ConstraintDescription: Specify either test or prod.

Conditions:
  IsProduction: !Equals
    - !Ref EnvType
    - prod
```

Parameters で指定した EnvType が prod であるかどうかを判断

```yaml
Resources:
  EC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref LatestAmiId
      InstanceType: !If [IsProduction, m6i.large, t3.micro]
```

prod であれば m6i.large インスタンスを構築

CloudFormation スタックの設計について

- CloudFormation 設計におけるスタック分割
- 環境ごとに再利用性可能なテンプレートの実現
- 同一構成のシステム間で再利用しやすいテンプレートの実現

同一構成のシステム間で再利用しやすいテンプレートの作成

- CloudFormation では、Amazon Resource Name (ARN) をテンプレートで指定したり、クロススタック参照のためにアカウント・リージョンごとに一意な値をエクスポートする場面がある

- これらの値をハードコーディングせず、擬似パラメータを利用して記述することでアカウントやリージョンを環境ごとに書き換えることが不要となり、同一構成のシステムを構築するためのテンプレートにおいて、再利用が容易となる

- エクスポート値は、${AWS::StackName} を含めることで一意な値を生成可能

擬似パラメータの例:
変更前: !Sub ‘arn:aws:ssm:us-east-1:111122223333:parameter/MyParameter’
変更後: !Sub ‘arn:aws:ssm:${AWS::Region}:${AWS::AccountId}:parameter/MyParameter’

開発環境の整備

CloudFormation 専用の Linter cfn-lint の導入

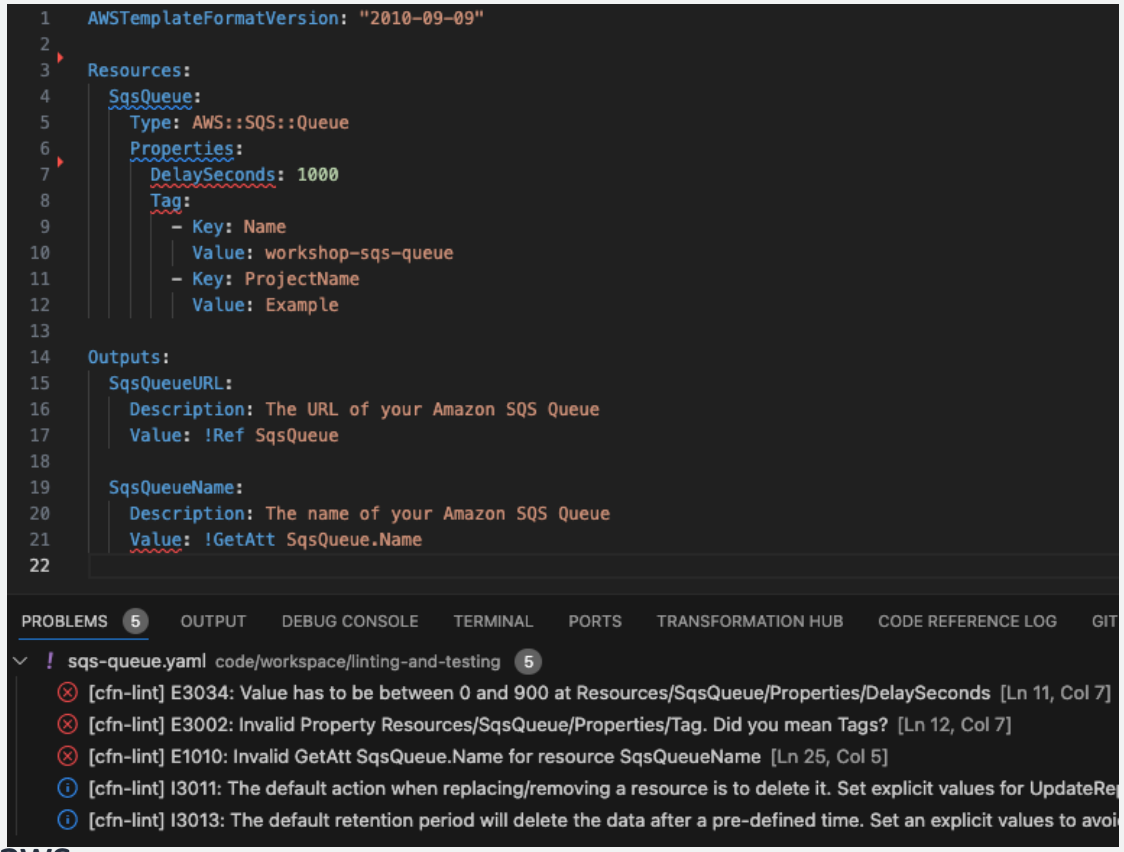
- CloudFormation では、cfn-lint という Linter が GitHub でパブリック公開
- Pip、Homebrew (macOS)、Docker、各IDE (Visual Studio Code, IntelliJ IDEA, Sublime, Emacs, Vim etc.) 用の Plugin など、各個人環境に合わせて導入が可能

cfn-lint https://github.com/aws-cloudformation/cfn-lint

cfn-lint の利用方法

Visual Studio Code からの実行例

検出された箇所に下線が引かれ、
PROBLEMS パネルに自動的に問題を出力



コマンドによる実行例

コマンドラインからの実行

```
$ cfn-lint template.yaml
```

複数のファイルを指定

```
$ cfn-lint template1.yaml template2.yaml
```

ワイルドカードを利用し複数ファイルを対象

```
$ cfn-lint path/*.yaml
$ cfn-lint path/**/*.yaml
```

cfn-lint https://github.com/aws-cloudformation/cfn-lint

CloudFormation Rain の導入

CloudFormation Rain は、テンプレートのデプロイ、削除、生成、スタックの操作等を容易に実行するコマンドラインツール

- rain deploy コマンドにより、スタックへのデプロイを実行。削除は rain rm コマンドで実行。加えられる変更の概要を表示しつつ、スタックのデプロイ中に更新ステータスを表示。
- rain fmt コマンドにより、CloudFormation を標準フォーマットにフォーマットしたり、JSON/YAML 形式へ変更
- rain build コマンドにより、リソースのテンプレートを生成可能 (次ページにて生成 AI 連携について記載)
- その他、StackSets の操作、デプロイ失敗の予測、ログの表示等、様々な機能をサポート

rain deploy コマンド実行後の出力

```
Deploying template 'resources.yaml' as stack 'resources' in us-east-1.
Stack resources: CREATE_COMPLETE
Successfully deployed resources
```

rain build AWS::EC2::VPC > vpc.yaml コマンド実行後の vpc.yaml の一部

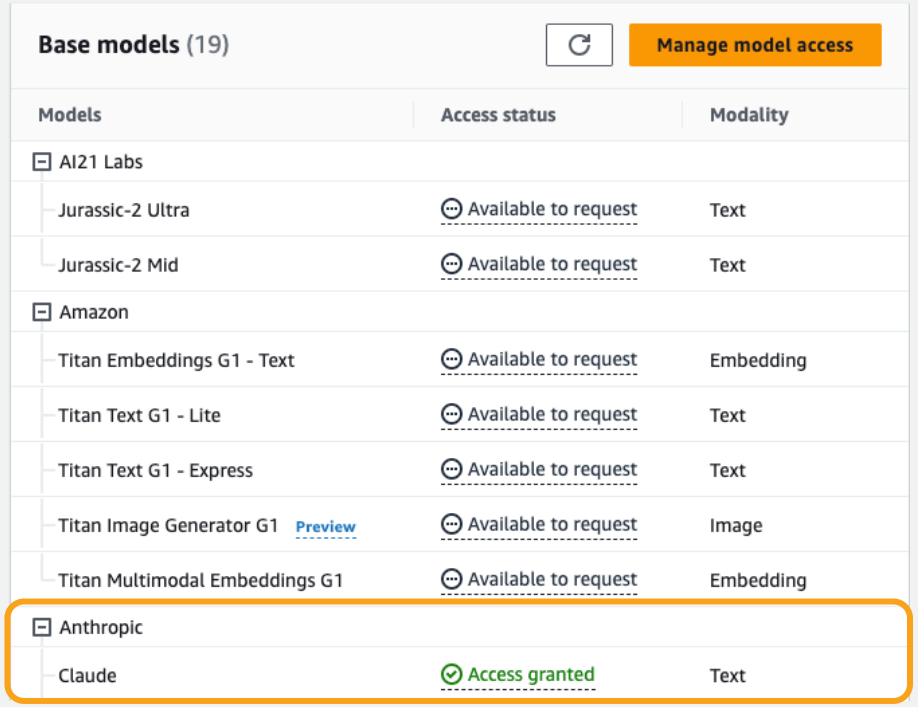
```yaml
AWSTemplateFormatVersion: "2010-09-09"

Description: Template generated by rain

Resources:
  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: CHANGEME # Optional
      EnableDnsHostnames: false # Optional
      EnableDnsSupport: false # Optional
      InstanceTenancy: CHANGEME # Optional
      Ipv4IpamPoolId: CHANGEME # Optional
      Ipv4NetmaskLength: 0 # Optional
      Tags:
        - Key: CHANGEME
          Value: CHANGEME
```

rain https://github.com/aws-cloudformation/rain

Rain と Amazon Bedrock の連携

Rainは、Amazon Bedrock と Claude2 モデルを使用した生成AIによるテンプレート生成をサポート。Claude は日本語に対応。

事前に Bedrock のAnthropic Claude モデルを有効化



作成したい構成を記載の上 rain build コマンドを実行

```
$ rain build -p "1つのVPCに2つのサブネット" > vpc_genai.yaml
```

コマンド実行後の vpc_genai.yaml の一部

```yaml
AWSTemplateFormatVersion: 2010-09-09
Description: A sample template

Resources:

  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsHostnames: true
      Tags:
        - Key: Name
          Value: !Ref AWS::StackName

  PublicSubnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref VPC
      CidrBlock: 10.0.1.0/24
      AvailabilityZone: !Select [ 0, !GetAZs '' ]
```

cfn-lint https://github.com/aws-cloudformation/cfn-lint

Amazon CodeWhisperer の活用

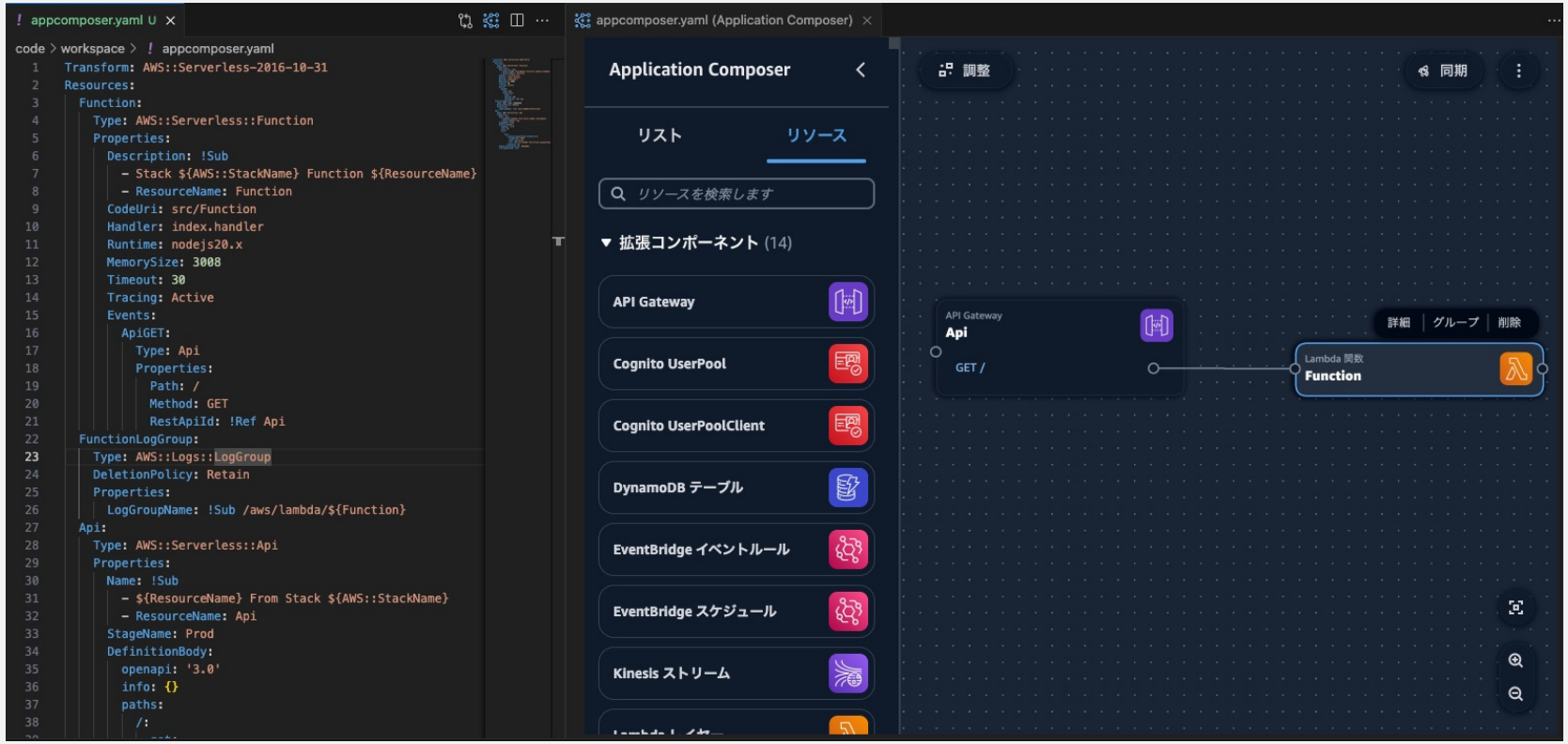
- CodeWhisperer を利用すると、コメント行に記載した構成を実現するためのコードサジェストやコードの補完が可能
- CloudFormation の YAML / JSON のセキュリティスキャンに対応
- エディタに AWS Toolkit をインストールし、AWS Builder ID でサインインすることで利用を開始

Amazon CodeWhisperer にて AI を活用した新しいコード修正、IaC サポート、および Visual Studio との統合提供を開始
https://aws.amazon.com/jp/blogs/news/amazon-codewhisperer-offers-new-ai-powered-code-remediation-iac-support-and-integration-with-visual-studio/

AWS Application Composer の活用

Application Composer は、Visual Studio CodeやAWS コンソールから任意のCloudFormationリソースをビジュアルキャンバスにドラッグアンドドロップし、CloudFormationテンプレートを自動生成するサービス

- VS Code の拡張機能 AWS Toolkit for Visual Studio Code として提供
- 14の拡張コンポーネントに加え、1,000を超える標準リソースをサポート
- 同期 (Sync) ボタンをワンクリックで CloudFormation スタックとしてデプロイ
- 生成 AI と連携し、リソース設定のためのコードの提案が可能
- 既存のコードの可視化に利用可能



IDE extension for AWS Application Composer enhances visual modern applications development with AI-generated IaC
https://aws.amazon.com/jp/blogs/aws/ide-extension-for-aws-application-composer-enhances-visual-modern-applications-development-with-ai-generated-iac/

テスト

CloudFormation の継続的テストについて

- CloudFormation のテストツールや AWS サービスを用いて、テストの効率化を目指す
- 目的や開発プロセスに応じて、下記のテストツールの CI/CD 上への組み込みを検討

- cfn-lint
- Guard
- Hooks
- AWS Control Tower プロアクティブコントロール
- AWS Configルール プロアクティブなコンプライアンス
- TaskCat

*Guard、Hooks、AWS Control Tower、AWS Configルールを用いると、組織の共通ルールを定めた上で開発全体に統制をかけることも可能

AWS CloudFormation Guard の導入

- セキュリティ、ガバナンス、ポリシーコンプライアンスの要件を設定し、AWS 環境を中央集権的に管理する際に CloudFormation Guard が利用可能
- Guard を用いて Policy as Code を実現すると、開発者のローカル環境やデリバリーパイプラインの継続的インテグレーション (CI) フェーズでルールに即しているかどうかの検証が可能

AWS CloudFormation Guard https://github.com/aws-cloudformation/cloudformation-guard

AWS CloudFormation Guard の利用方法

- CloudFormation のテンプレートに対し、値を確認するための Rule を記述した .guard ファイルを用意
- cfn-guard validate コマンドでテンプレートとルールを指定し、テストを実行

S3 Bucket 作成のテンプレート

```yaml
AWSTemplateFormatVersion: "2010-09-09"

Resources:
  SampleBucket:
    Type: AWS::S3::Bucket

    Properties:
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
              SSEAlgorithm: AES256
      VersioningConfiguration:
        Status: Enabled
```

S3 Bucket リソースタイプのチェックを行う Guard Rule
SSEAlgorithm と Versioning の値をチェック

```
AWS::S3::Bucket {
    Properties {
        BucketEncryption.ServerSideEncryptionConfiguration[*] {
            ServerSideEncryptionByDefault.SSEAlgorithm == 'AES256'
            <<BucketEncryption not configured with the AES256 algorithm>>
        }
        VersioningConfiguration.Status == 'Enabled'
        <<BucketEncryption not configured with versioning enabled>>
    }
}
```

cfn-gurad validateコマンド実行後の出力

```
example_bucket.yaml Status = PASS
PASS rules
example_bucket.guard/default    PASS
```


AWS CloudFormation Guard https://github.com/aws-cloudformation/cloudformation-guard

Guard ルール記述時のポイント

ルール記述の際には、以下の構文を利用
構文: <query> <operator> [query|value literal] [custom message]

	説明	必須
query	ドット (.) で区切られた式で階層化し、評価対象を指定	必須
operator	queryの状態を確認するための単項演算子または二項演算子 二項演算子:== (等しい) | != (等しくない) | > (より大きい) >= (以上) | < (より小さい) | <= (以下) | IN (リスト内) 単項演算子: exists | empty | is_string | is_list | is_struct | not (!)	必須
query|value literal	クエリや条件で利用する値リテラルを記述 string, integer(64), float(64), bool, char, regexといった値の他に、 範囲を示す r[<下限>, <上限>] といった記述も可能	二項演算子を使用する場合は必須
custom message	関する情報を提供する文字列。メッセージはvalidateコマンドと testコマンドの詳細出力に表示され、ルール評価の理解やデバッグに役立つ	任意

*複数のルールを記述した場合、暗黙的にAND条件で判断される。構文の後ろにORを指定することが可能
構文: <query> <operator> [query|value literal] [custom message] [or|OR]

Writing Guard rules https://docs.aws.amazon.com/ja_jp/cfn-guard/latest/ug/writing-rules.html

Guard ルールの記述例

単項演算子を利用したときの利用例

- S3Bucket に BucketEncryption プロパティが定義されているかどうかをチェック
- BucketEncryption プロパティが定義されていれば評価は PASS

```
Resources.S3Bucket.Properties.BucketEncryption exists
```

二項演算子を利用したときの利用例

- VolumeType プロパティに指定された値が io2、gp3 であるかどうかをチェック
- NewVolume にいずれかが指定されていれば、評価は PASS

```
Resources.NewVolume.Properties.NewVolume.VolumeType IN ['io2','gp3’]
```

custom messageの利用例

- エラー時に、メッセージとして「Allowed Volume Types are io2 and gp3」を表示

```
Resources.NewVolume.Properties.VolumeType IN [ 'io2','gp3’ ]
<<Allowed Volume Types are io2 and gp3>>
```


AWS CloudFormation Guard https://github.com/aws-cloudformation/cloudformation-guard

モジュール性と再利用性を考慮した Guard ルール記述

モジュール化された Guard ルールを作成すると、再利用性が向上するだけでなく、データを検証したときに失敗したルールを特定したり、ルールをトラブルシューティングするのが簡単

- 条件を定義する when を利用し、S3 バケットが存在する場合にのみルールを実行するように記述
- 1つのロジックを2つのルール (Named-rule block) に分解して記述

```
AWS::S3::Bucket {
    Properties {
        BucketEncryption.ServerSideEncryptionConfiguration[*] {
            ServerSideEncryptionByDefault.SSEAlgorithm == 'AES256'
            <<BucketEncryption not configured with the AES256 algorithm>>
        }
        VersioningConfiguration.Status == 'Enabled'
        <<BucketEncryption not configured with versioning enabled>>
    }
}
```

```
let my_buckets = Resources.*[ Type == 'AWS::S3::Bucket' ]

rule validate_bucket_sse when %my_buckets !empty {
    %my_buckets.Properties {
        BucketEncryption.ServerSideEncryptionConfiguration[*] {
            ServerSideEncryptionByDefault.SSEAlgorithm == 'AES256'
            <<BucketEncryption not configured with the AES256 algorithm>>
        }
    }
}

rule validate_bucket_versioning when %my_buckets !empty {
    %my_buckets.Properties {
        VersioningConfiguration.Status == 'Enabled'
        <<BucketEncryption not configured versioning enabled>>
    }
}
```


Composing named-rule blocks https://docs.aws.amazon.com/ja_jp/cfn-guard/latest/ug/named-rule-block-composition.html

Guard を利用したユニットテスト

Guard ではルールのテストを作成し、ルールが結果通りに機能することを検証可能

テストケースと期待する結果を
記載したyamlファイル

```yaml
- input:
    Resources:
      MyExampleBucket:
        Type: AWS::S3::Bucket
        Properties:
          BucketEncryption:
            ServerSideEncryptionConfiguration:
              - ServerSideEncryptionByDefault:
                  SSEAlgorithm: AES256
  expectations:
    rules:
      validate_bucket_sse_example: PASS

- input:
    Resources:
      MyExampleBucket:
        Type: AWS::S3::Bucket
        Properties:
          VersioningConfiguration:
            Status: Suspended
  expectations:
    rules:
      validate_bucket_versioning_example: FAIL
```

Guard ルールを記載したファイル

```
let my_buckets = Resources.*[ Type == 'AWS::S3::Bucket' ]


rule validate_bucket_sse_example when %my_buckets !empty {
    %my_buckets.Properties {
        BucketEncryption.ServerSideEncryptionConfiguration[*] {
            ServerSideEncryptionByDefault.SSEAlgorithm == 'AES256'
            <<BucketEncryption not configured with the AES256 algorithm>>
        }
    }
}


rule validate_bucket_versioning_example when %my_buckets !empty {
    %my_buckets.Properties {
        VersioningConfiguration.Status == 'Enabled'
        <<BucketEncryption not configured versioning enabled>>
    }
}
```

cfn-guard testコマンド実行後の出力

```
Test Case #1
  No Test expectation was set for Rule validate_bucket_versioning_example
  PASS Rules:
    validate_bucket_sse_example: Expected = PASS

Test Case #2
  No Test expectation was set for Rule validate_bucket_sse_example
  PASS Rules:
    validate_bucket_versioning_example: Expected = FAIL
```

AWS CloudFormation Hooks の導入

- CloudFormation によるスタックやリソース作成/更新/削除の前に処理を実行するために、CloudFormation Hooks の利用が可能
- カスタムロジックを呼び出してアクションを自動化したり、リソース設定を検査することが可能
- CloudFormation Public Registry に公開されているサンプル Hooks を利用したり、CloudFormation CLIを使用してHooksを作成し、CloudFormation Private Registry に公開

*Hooksの利用方法の詳細は、「AWS Blackbelt CloudFormationレジストリ編」を参照

AWS Cloudformation Sample Hooks
https://github.com/aws-cloudformation/aws-cloudformation-samples/tree/main/hooks

Guard と Hooks の違い

Guard

- CloudFormation テンプレートが特定のルールやポリシーに従っているかを検証
- CloudFormation テンプレートがデプロイされる前に、そのテンプレートが定義されたルールセットに適合しているかどうかをチェック。主にビルドステージの静的分析のフェーズで実施

Hooks

- CloudFormation のデプロイプロセス中に特定のリソースに対してカスタム検証ルールを適用し、リソースがデプロイされる前に、特定の要件やポリシーに従っているかを検証
- スタックの作成、更新、削除のプロセス中にトリガーされ、Lambda 関数を使用してリソース設定を検証し、カスタムロジックによるアクションの自動化を実施

AWS Control Tower や AWS Config ルールによるガードレールや事前テストの実現

AWS Control Tower プロアクティブコントロール

- すべての CloudFormation デプロイメントに自動的に適用され、準拠していないリソースが プロビジョニングされる前にブロック
- AWS CloudFormation Hooks によりプロビジョニングのブロックを実装

AWS Config ルール プロアクティブなコンプライアンス

- AWSリソースをプロビジョニングする前に、事前定義した AWS Config ルールに準拠しているかどうかを事前に確認する機能
- CloudFormation Hooks を使用し、実際のデプロイが行われる前に設定の事前確認が可能

TaskCat によるインテグレーションテスト

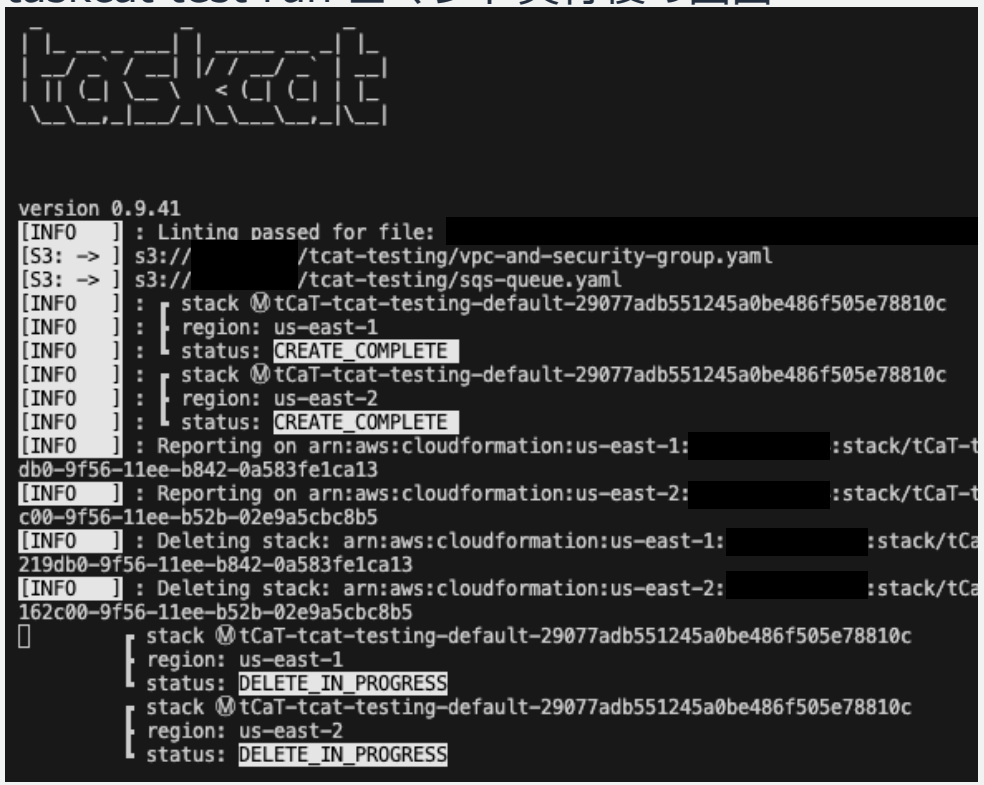
- 指定した AWS リージョンに CloudFormation テンプレートを実行し、それらが期待通りに機能するかどうかを検証するインテグレーションテストのために TaskCat が利用可能
- 指定した AWS リージョンに対し、テンプレートからスタックを作成し、成功/失敗のレポートを出力した後にスタックを削除
- デプロイ先のリージョンを複数指定することによる並列実行も可能

テストの実行内容を記載した
.taskcat.yml ファイル

```yaml
project:
  name: tcat-testing
  package_lambda: false

  regions:
  - us-east-1
  - us-east-2
tests:
  default:
    template: ./vpc-and-security-group.yaml

general:
  s3_bucket: tcat-test
  parameters:
    VpcIpv4Cidr: 172.16.0.0/16
```

テストを実行したい
リージョンを指定

taskcat test run コマンド実行後の画面



TaskCat https://github.com/aws-ia/taskcat

デプロイ

CloudFormation のデプロイ方法について

CloudFormation では、以下の方法でデプロイを検討

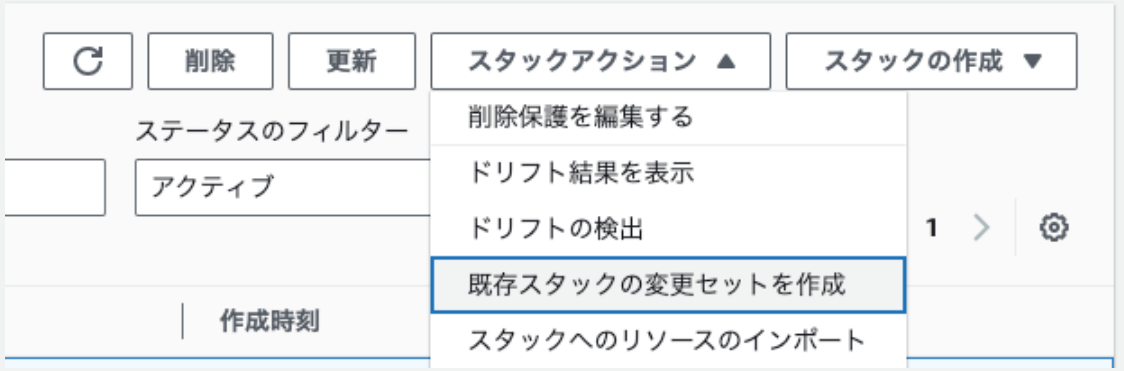
- マネジメントコンソールや CLI から手動でテンプレートをデプロイ
  - CI/CD の整備をするほどの頻度や規模でない・ヒューマンエラーによるリスクが少ない等、自動化の恩恵を受けづらい時に利用
- CI/CD パイプラインからデプロイ
  - テストも含めた自動化を実現したい時に利用
- Git からの同期
  - Git ベースでコード管理しており、開発用途で CD を簡易に自動化したい時に利用

CloudFormation のデプロイ方法

- マネジメントコンソールや CLI から手動でテンプレートをデプロイ
- CI/CD パイプラインからデプロイ
- Git からの同期

マネジメントコンソールや CLI から手動でテンプレートをデプロイ

- S3 経由やマネジメントコンソール上からテンプレートファイルの直接アップロードによりスタックの作成や更新が可能
- CLI で更新する場合、deploy コマンドを利用することで変更セット作成後にスタックの作成や更新が可能
- 変更セットを利用し、デプロイの前にリソースの更新や削除を把握
- deploy コマンドを利用し、変更セットを確認してから変更を反映したい場合は、--no-execute-changeset フラグを利用



CloudFormation のデプロイ方法

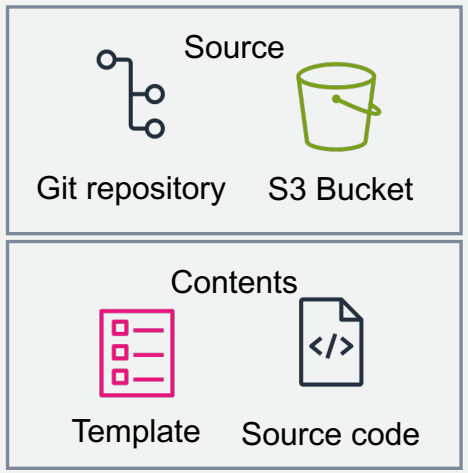
- マネジメントコンソールや CLI から手動でテンプレートをデプロイ
- CI/CD パイプラインからデプロイ
- Git からの同期

CI/CD パイプラインからデプロイ

CI/CDパイプライン経由で、テストやアプリケーションのビルド、デプロイが可能

CFn スタックをデプロイするパイプラインの例

1 Source → 2 Build → 3 テストスタックデプロイ → 4 本番スタックデプロイ



パラメータ (設定) ファイル作成
Lambda ソースコードのビルド
コンテナイメージのビルド
cfn-lint、Guard等のテストツール実行

テストスタック作成
↓
Approve
↓
テストスタック削除

Hooks等のテストツール実行

変更セット作成
↓
Approve
↓
本番スタックデプロイ

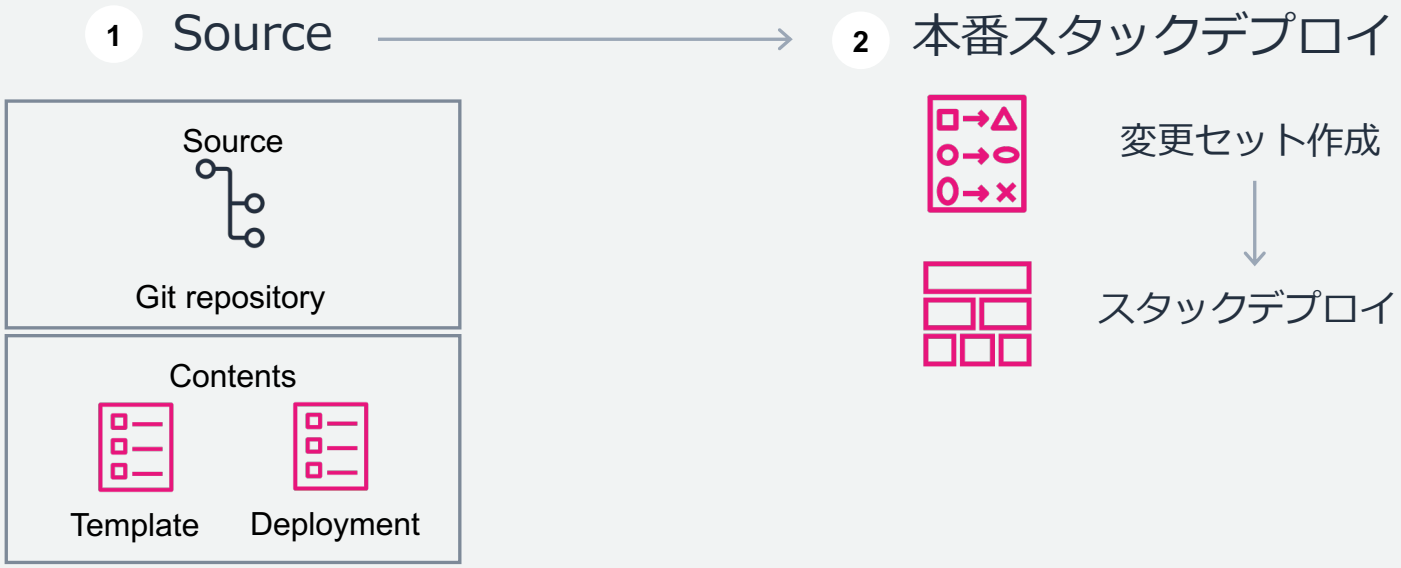
CodePipelineのCloudFormationアクションを利用すると、任意のコードは実行できないことから、よりセキュアにスタックデプロイが可能
通常は、スタックデプロイをするコンピュートリソースにAdmin相当の権限を与えることが必須

https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/continuous-delivery-codepipeline-basic-walkthrough.html

CloudFormation のデプロイ方法

- マネジメントコンソールや CLI から手動でテンプレートをデプロイ
- CI/CD パイプラインからデプロイ
- Git からの同期

Git からの同期



- Git と連携させて CloudFormation テンプレートをデプロイする際に利用
- CloudFormation テンプレートファイルと、スタックを設定するパラメータを含むスタックデプロイメントファイルの2つに変更がないかを監視し、Git リポジトリ上で更新があればスタックの作成、更新を自動的に実施
- デプロイ前に変更セットを作成
- GitHub、GitHub Enterprise、GitLab、Bitbucket のリポジトリをサポート (2023年12月現在)

Git からの同期設定 (1/4)

デベロッパー用ツールの設定より、事前に、利用している Git リポジトリへの接続を作成

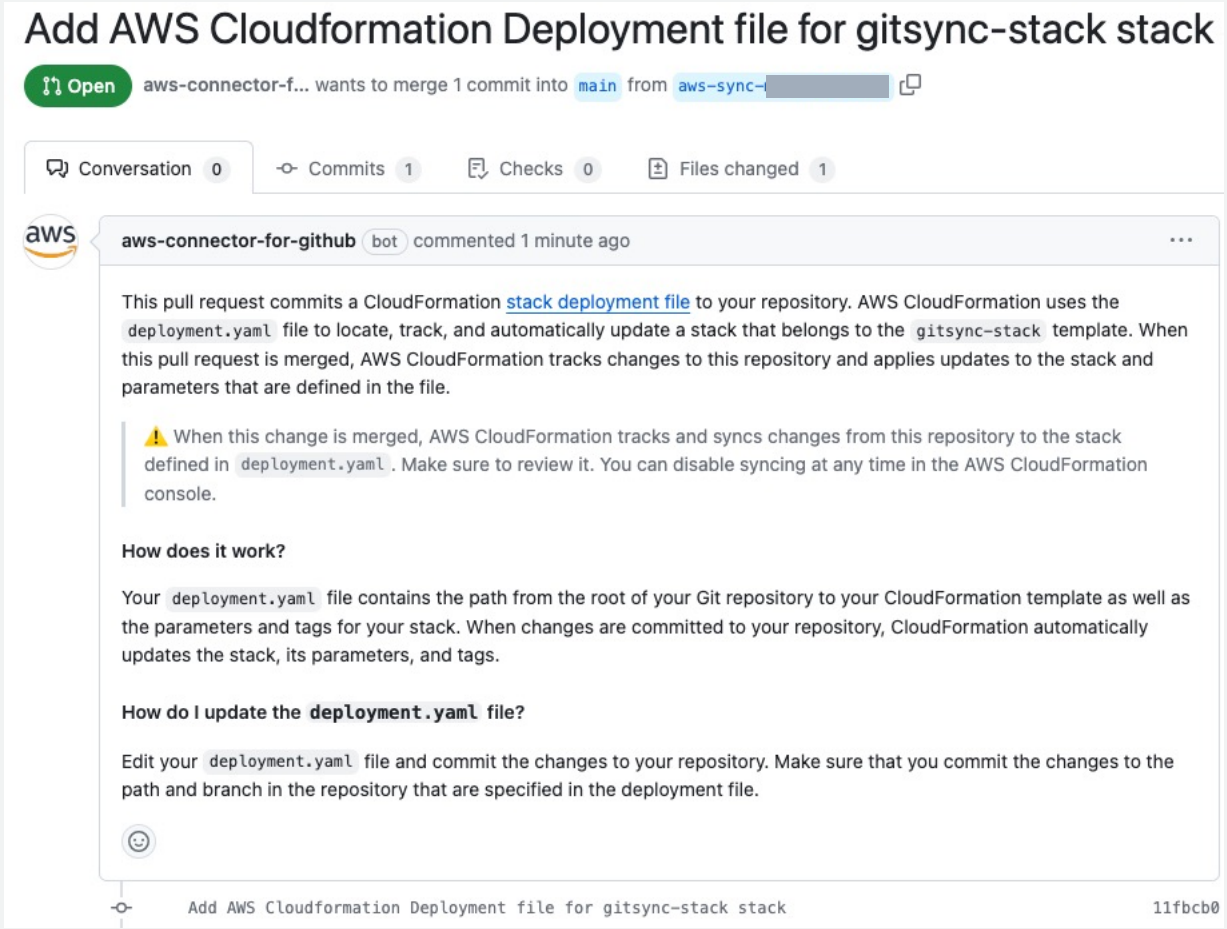
Git からの同期設定 (2/4)

スタックの作成画面->テンプレートの指定より「Gitから同期」を選択し、画面の指示に従って値を設定

Git からの同期設定 (3/4)

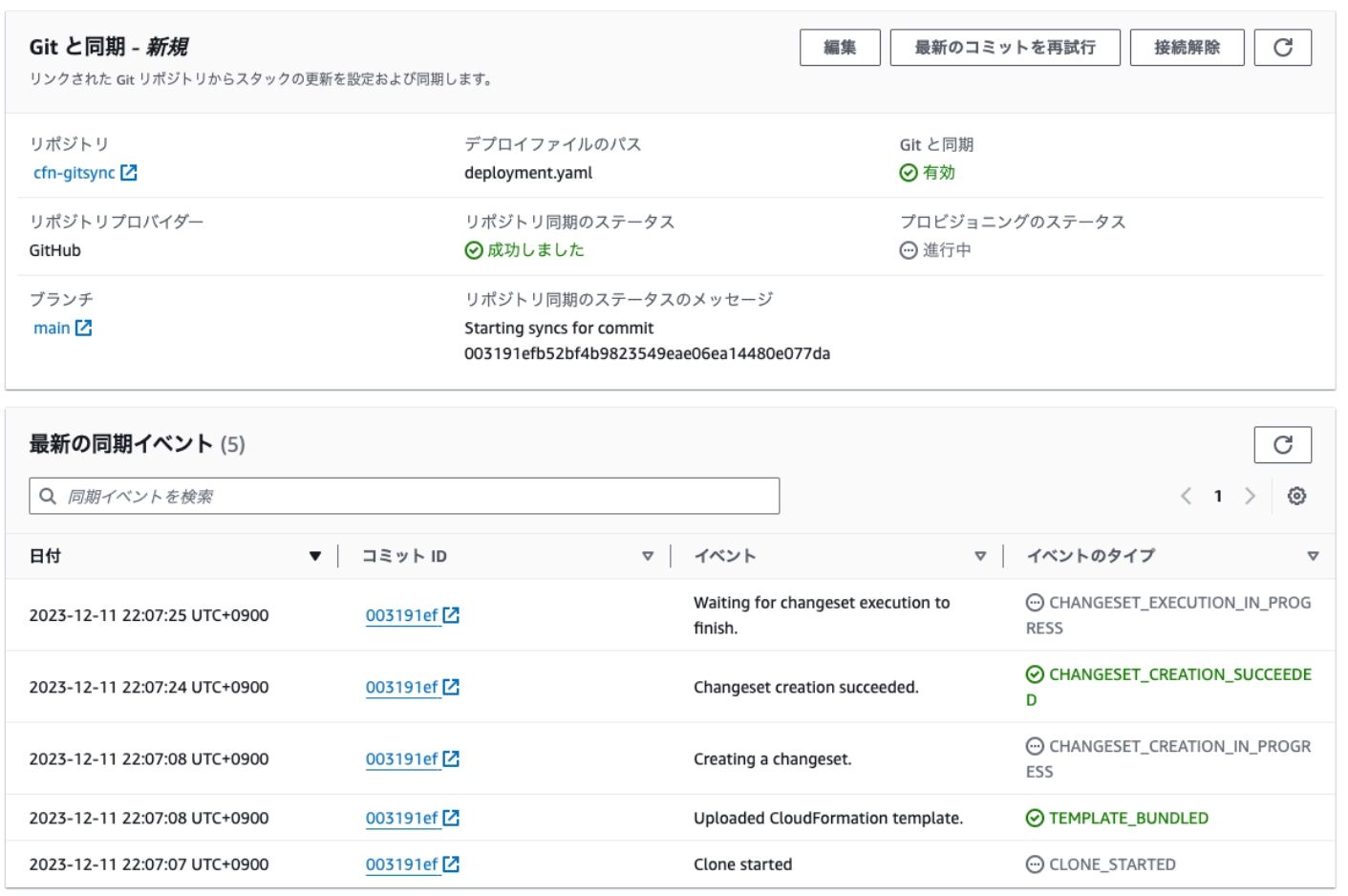
Git と同期ステータスが有効となると、
コードリポジトリに対してPull Request を発行

AWS が発行した Pull Request を
ユーザー側で Merge



Git からの同期設定 (4/4)

指定した Git リポジトリのブランチに更新をかけると、変更セットを作成し、スタックを更新



まとめ

- CloudFormation スタックとコード設計の勘所
  - CloudFormation 設計におけるスタック分割
  - 環境ごとに再利用性可能なテンプレートの実現
  - 同一構成のシステム間で再利用しやすいテンプレートの作成

- 開発環境の整備
  - テンプレート開発支援ツールの導入
    cfn-lint / Rain / CodeWhisperer / Application Composer

- テスト
  - テストツール導入
    cfn-lint / Guard / Hooks / TaskCat
    AWS Control Tower プロアクティブコントロール
    AWS Config ルール プロアクティブなコンプライアンス

- デプロイ
  - マネジメントコンソールや CLI から手動でテンプレートをデプロイ
  - CI/CD パイプラインからデプロイ
  - Git からの同期

Thank you!