#devaxconnect

aws DevAx connect

アプリの次はインフラのCI/CDも考えよう

コードによるインフラ管理・安全なCI/CDの実現

Masashi Tomooka
Prototyping Engineer
Amazon Web Services Japan G.K.
mtomooka@amazon.co.jp

自己紹介

友岡 雅志 Tomooka Masashi
技術統括本部 Prototyping Engineer

前職以前

- mBaaS開発 (Rails, Sinatra, Ruby, MySQL)
- ゲームクライアント開発 (Unity, C#)
- 航空宇宙系大学院 (C, C++, MATLAB, Python)

好きなAWSサービス

- AWS Cloud Development Kit

@tmokmss

本日のテーマ

インフラのCI/CD！

ここでいう「インフラ」とは

Infrastructure as Code (IaC) で記述可能なAWSリソース全般、と定義する

CloudFormation

Cloud Development Kit

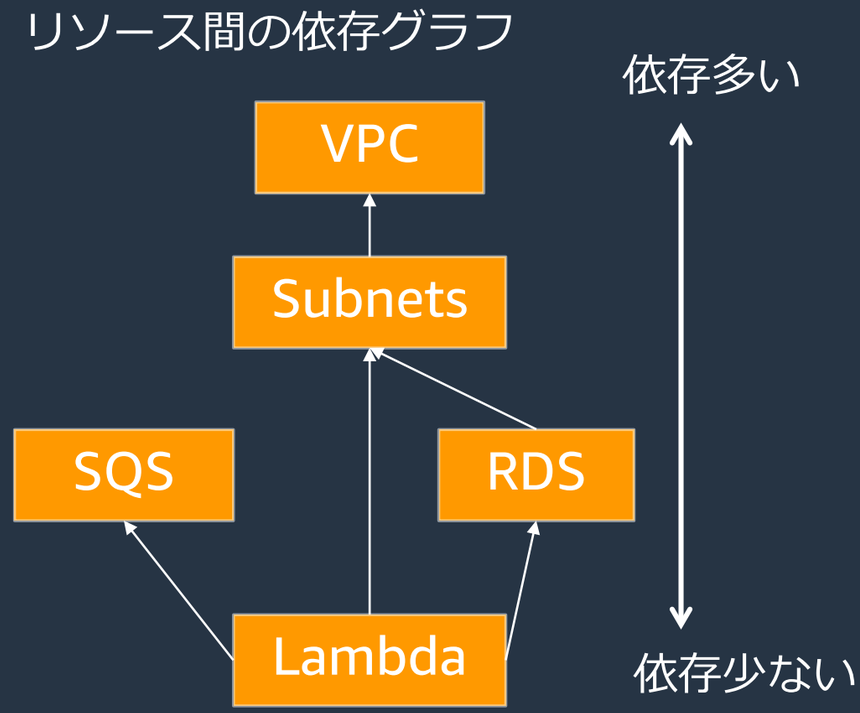
HashiCorp Terraform

参考: Infrastructure as Code 談議 2022

ステートフル vs ステートレス

- リソースの構成情報以外のデータを内包するかどうか
- ステートフル: RDSクラスタ, SQSキュー, S3バケット など
- ステートレス: VPC, ECSクラスタ, Lambda関数など

依存関係の多寡 (右図)



ここでアンケート

Q. インフラ、どのようにデプロイしてますか？

1. IaCを使ってインフラの大部分をCI/CDしている
2. IaCを使ってインフラの一部のみCI/CDしている
3. IaCを使って手動でデプロイしている
4. IaCは使わず、マネジメントコンソール等で構築･変更している

こんな課題を抱えてないですか

1. デプロイ手順書の運用・管理が煩雑

2. インフラを変更することへのハードルが高い

3. 強い権限を個別ユーザーに付与している

4. インフラの現状を把握できていない

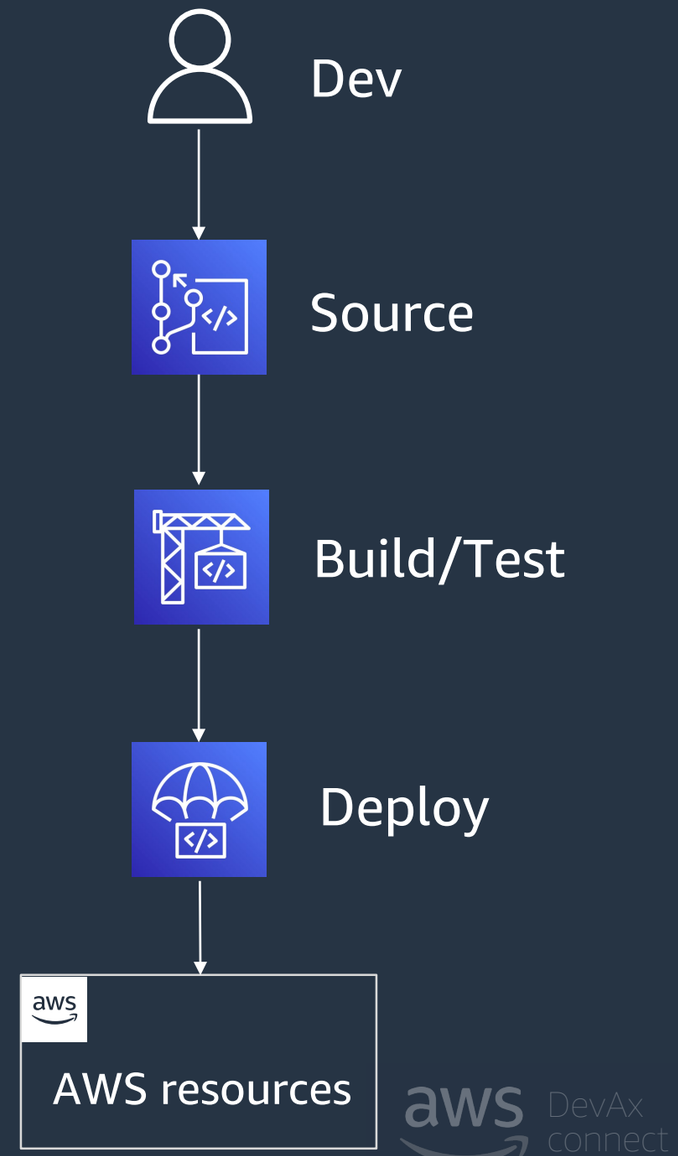
インフラのCI/CDが解決できること

1. デプロイ手順書の運用・管理が煩雑
   → 自動化により手順書は最小限に

2. インフラを変更することへのハードルが高い
   → デプロイが定形作業になり、より気軽なものに

3. 強い権限を個別ユーザーに付与している
   → パイプライン用のロールで管理

4. インフラの現状を把握できていない
   → IaC化による解決 (副産物)



なぜインフラはCI/CDしないのか？

インフラはアプリより変更頻度が低いので…

万が一意図しない変更が勝手に適用されたら被害が大きいので不安…
自動化すると何かとリスクが大きそうなので、手動で実行している

インフラは別チームが見ているので不可侵です

なぜインフラはCI/CDしないのか？

インフラはアプリより変更頻度が低いので…
→ 逆にCI/CDしないから変更頻度を高められないのかも？

万が一意図しない変更が勝手に適用されたら被害が大きいので不安…
自動化すると何かとリスクが大きそうなので、手動で実行している
→ この不安感を低減することが第一歩となりそう

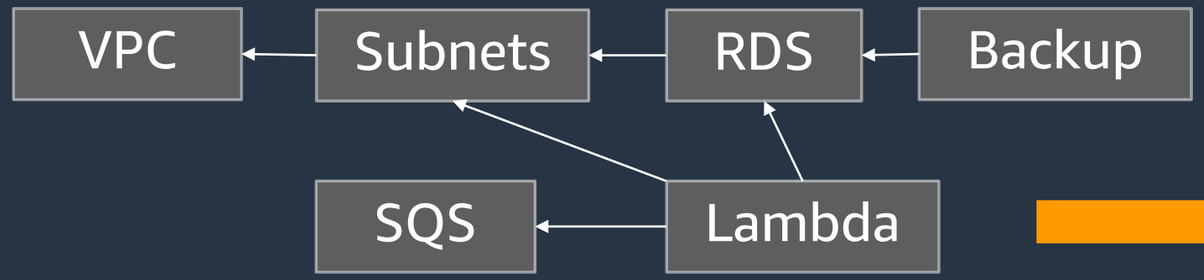
インフラは別チームが見ているので不可侵です
→ 組織的な要因も考えられますが、今日は技術的な側面に集中

安全なインフラCI/CDのために

CI/CD化はなんとなく不安…

その不安を払拭するためにできることを考える

インフラの変更が難しい理由



データの扱いを考慮する必要があるため

難度高

	ステートレス	ステートフル
そのリソースに対して依存するリソースが多い	VPC VPC サブネット ECS クラスタ	RDS クラスタ ElastiCache クラスタ
そのリソースに対して依存するリソースが少ない	ECS サービス/タスク定義 Lambda 関数 CloudFront distribution	S3 バケット SQS キュー Cognito ユーザープール

他のリソースにも変更が波及するため

難度高

例示はよくある傾向; 実際はワークロード次第です

安全にインフラをCI/CDするために

ゴール
- 意図しない変更が含まれないことを確実にする
- できる限りデプロイの影響を最小化する

そのための手段
1. 差分の確認
2. デプロイ前自動テストの整備
3. デプロイ後自動テストの整備
4. デプロイの影響を事前に検討

1. 差分の確認

IaCのソリューションでは、適用前に差分 (Diff) を確認する機能がある

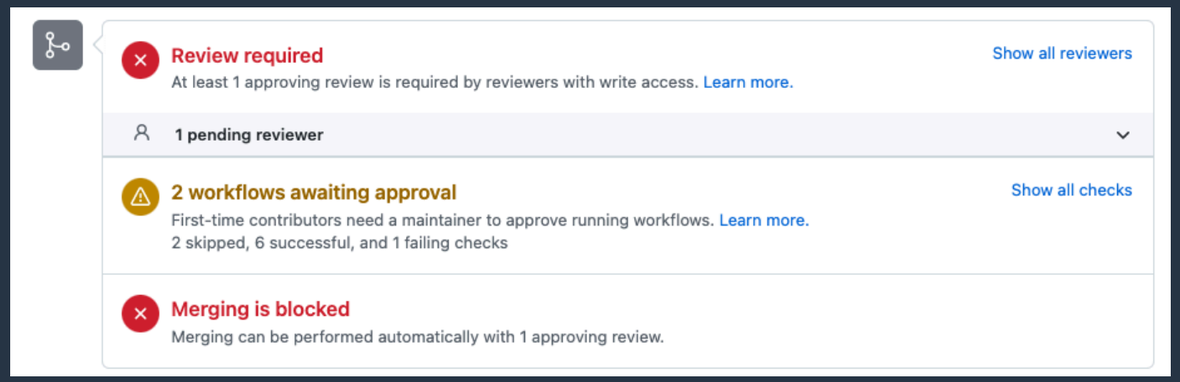
Diffの通りに変更が適用されることはIaC側が担保しているので
開発者はDiffが意図どおりであることだけを確認すれば良い

差分の確認例

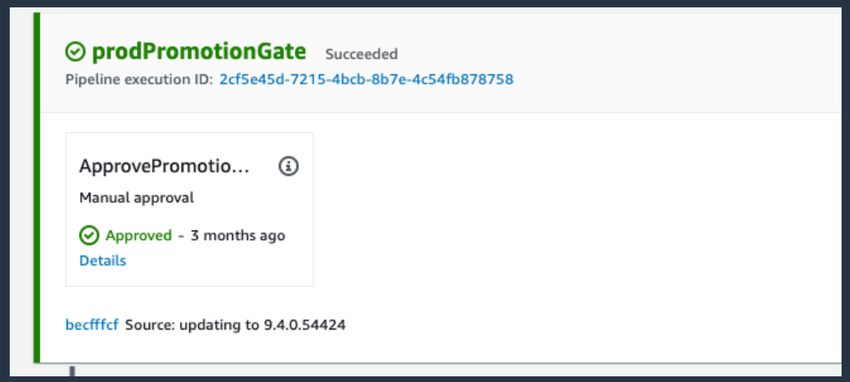
例えばCDKの場合:

```
→  serverless-full-stack-webapp-starter-kit git:(main) npx cdk diff
Stack ServerlessFullstackWebappStarterKitStack
Resources
[~] AWS::Cognito::UserPool Auth/UserPool AuthUserPool8115E87F
 └─ [~] Policies
     └─ [~] .PasswordPolicy:
         └─ [~] .MinimumLength:
             ├─ [-] 8
             └─ [+] 16
[~] AWS::DynamoDB::Table Database/Default DatabaseB269D8BB replace
 ├─ [~] AttributeDefinitions (may cause replacement)
 │   └─ @@ -1,6 +1,6 @@
 │       [ ] [
 │       [ ]   {
 │       [-]     "AttributeName": "PK",
 │       [+]     "AttributeName": "pk",
 │       [ ]     "AttributeType": "S"
```

Pull request単位で差分確認



CDパイプライン内で確認



2. デプロイ前自動テストの整備

そもそもデプロイするインフラに諸々の問題が含まれないかを検証
　予防的統制、Shift Leftなどとも

- 脆弱性の静的解析
  - 定義されるインフラに脆弱性が含まれていないか

(CDKなど、何らかのロジックでIaCテンプレートを合成するツールの場合)

- スナップショットテスト
  - 前のバージョンと出力を比較して、意図しない変更がないことを確認
  - 実環境へのアクセス不要。実装コストも低いので、採用しやすい手法

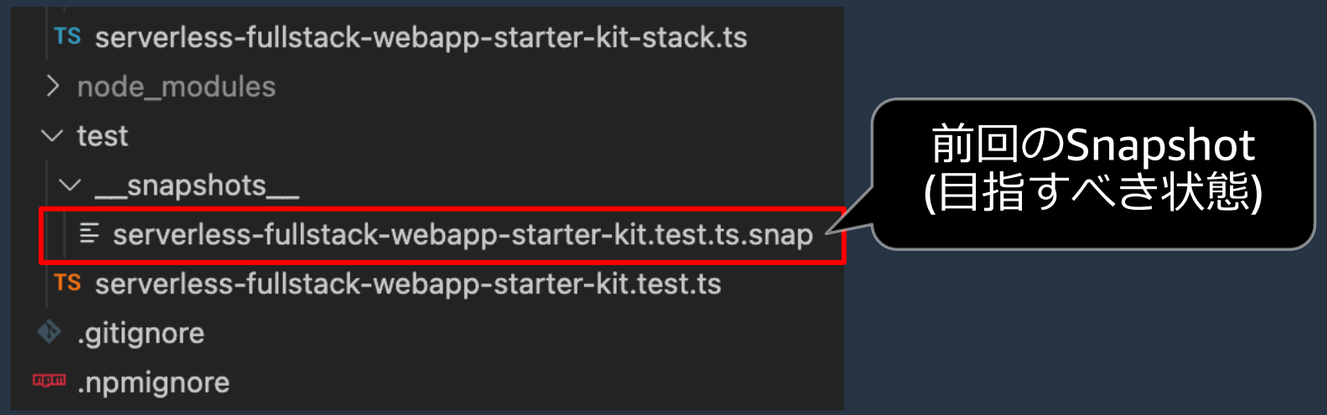
- 単体テスト / Validation test

スナップショットテストの例 (CDK)

コード

```
test("Snapshot test", () => {
  const app = new cdk.App();
  const stack = new ServerlessFullstackWebappStarterKitStack(app, "TestStack"
  const template = Template.fromStack(stack);
  expect(template).toMatchSnapshot();
});
```

リポジトリ内のスナップショット



実行結果

```
→  serverless-full-stack-webapp-starter-kit git:(main) ✗ npm test

> serverless-fullstack-webapp-starter-kit@0.1.0 test
> jest

 FAIL  test/serverless-fullstack-webapp-starter-kit.test.ts (9.794 s)
  × Snapshot test (468 ms)

  ● Snapshot test

    expect(received).toMatchSnapshot()

    Snapshot name: `Snapshot test 1`

    - Snapshot  - 3
    + Received  + 3

    @@ -349,11 +349,11 @@
              ],
              "EmailVerificationMessage": "The verification code to you
              "EmailVerificationSubject": "Verify your new account",
              "Policies": Object {
                "PasswordPolicy": Object {
    -             "MinimumLength": 8,
    +             "MinimumLength": 16,
                  "RequireNumbers": true,
                  "RequireSymbols": true,
```

静的解析の実装例 (cdk-nag)

簡単な設定で継続的に脆弱性の監視

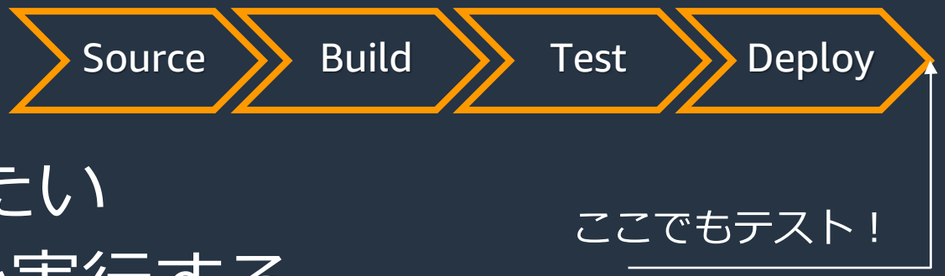
```
import { Aspects } from 'aws-cdk-lib';
import { AwsSolutionsChecks } from 'cdk-nag';
Aspects.of(app).add(new AwsSolutionsChecks());
```

Rule ID	Resource ID	Compliance	Rule Level	Rule Info
AwsSolutions-S10	ServerlessFullstackWebapp	Compliant	Error	The S3 Bucket or bucket policy does not require requests to use SSL.
AwsSolutions-S1	ServerlessFullstackWebapp	Non-Compliant	Error	The S3 Bucket has server access logs disabled.
AwsSolutions-S2	ServerlessFullstackWebapp	Compliant	Error	The S3 Bucket does not have public access restricted and blocked.
AwsSolutions-S3	ServerlessFullstackWebapp	Compliant	Error	The S3 Bucket does not default encryption enabled.
AwsSolutions-S5	ServerlessFullstackWebapp	Compliant	Error	The S3 static website bucket either has an open world bucket policy or does not use a Origin Access Identity (OAI) in the bucket policy for limited getObject and/or putObject permissions.
AwsSolutions-S10	ServerlessFullstackWebapp	Compliant	Error	The S3 Bucket or bucket policy does not require requests to use SSL.
AwsSolutions-IAM5	ServerlessFullstackWebapp	Compliant	Error	The IAM entity contains wildcard permissions and does not have a cdk-nag rule suppre evidence for those permission.
AwsSolutions-IAM4	ServerlessFullstackWebapp	Non-Compliant	Error	The IAM user, role, or group uses AWS managed policies.
AwsSolutions-IAM5	ServerlessFullstackWebapp	Compliant	Error	The IAM entity contains wildcard permissions and does not have a cdk-nag rule suppre evidence for those permission.
AwsSolutions-SQS2	ServerlessFullstackWebapp	Compliant	Error	The SQS Queue does not have server-side encryption enabled.
AwsSolutions-SQS3	ServerlessFullstackWebapp	Non-Compliant	Error	The SQS queue does not have a dead-letter queue (DLQ) enabled or have a cdk-nag ru suppression indicating it is a DLQ.
AwsSolutions-SQS4	ServerlessFullstackWebapp	Non-Compliant	Error	The SQS queue does not require requests to use SSL.

CloudFormation: cfn-nag, cfn-guard など
Terraform: Checkov, tfsec など

参考: AWS Cloud Development Kit と cdk-nag で
アプリケーションのセキュリティとコンプライアンスを管理する

3. デプロイ後自動テストの整備



インフラが正常にデプロイされたことを確認したい
→ デプロイ後、AWSリソースに対する検証を実行する

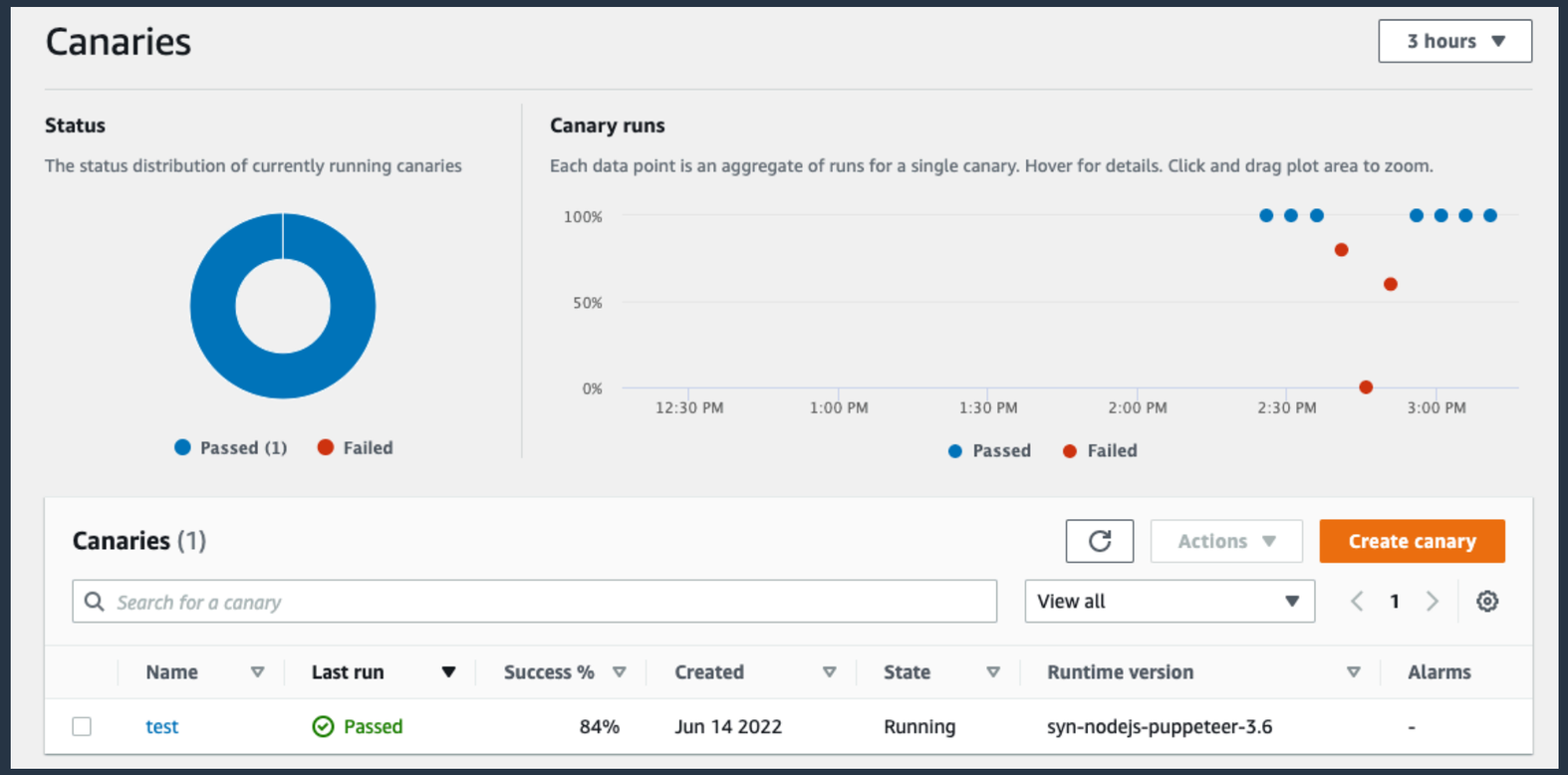
前段のステップで見逃しうることを確認できると良い　例えば:

- APIエンドポイントの呼び出し
- 負荷試験
- ネットワーク疎通 (NACL, Security group設定など)
- 脆弱性の発見
- etc…

確認したいことに合わせて適切にツールを選択する

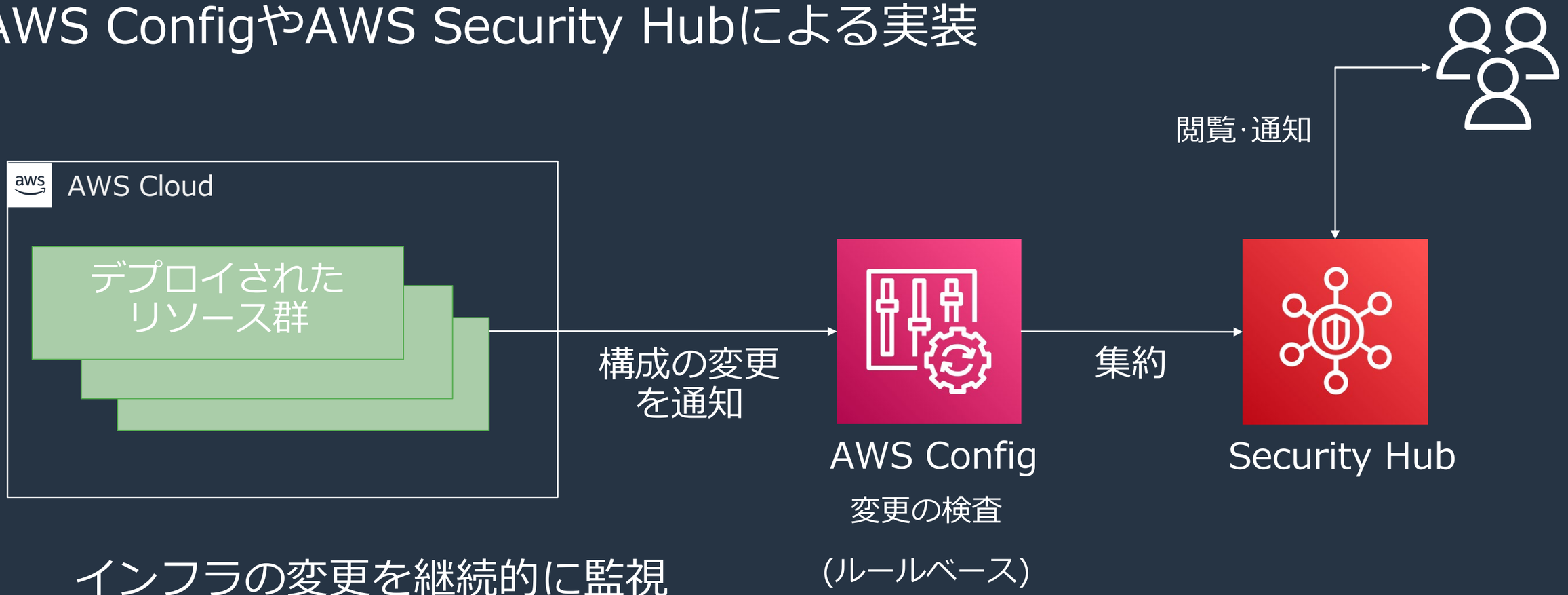
例: APIエンドポイントの呼び出し

CloudWatch SyntheticsのCanaryでWeb APIエンドポイントを監視



例: 脆弱性の発見

AWS ConfigやAWS Security Hubによる実装



インフラの変更を継続的に監視

安全にインフラをCI/CDするために

ゴール

- 意図しない変更が含まれないことを確実にする
- できる限りデプロイの影響を最小化する

手段

1. 差分の確認
2. デプロイ前自動テストの整備
3. デプロイ後自動テストの整備
4. デプロイの影響を事前に検討

4. デプロイの影響を事前に検討

デプロイ前に必ず確認するチェックリストを作る

- ✓ デプロイによるサービスの中断は発生するか
- ✓ デプロイする変更はロールバック可能か

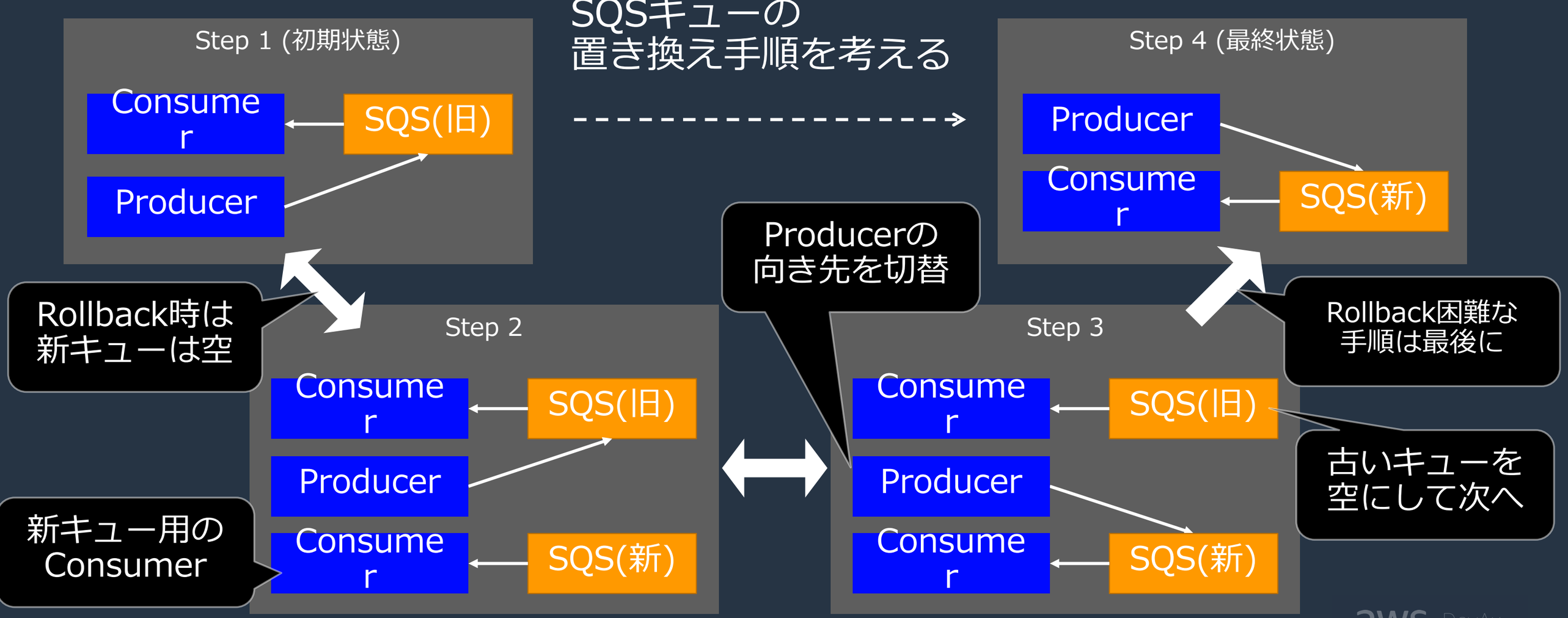
例えば、CloudFormationによる変更は3種類に大別される

1. 中断を伴わない更新 (update with no interruption)
2. 一時的な中断を伴う更新 (update with some interruption)
3. 置き換え (replacement)

```
            └ [+] 16
[~] AWS::DynamoDB::Table Database/Default DatabaseB269D8BB replace
 ├ [~] AttributeDefinitions (may cause replacement)
 │  └ @@ -1,6 +1,6 @@
 │     [ ] [
 │     [ ]   {
 │     [-]     "AttributeName": "PK",
 │     [+]     "AttributeName": "pk",
```

2と3はデプロイ時の挙動を特に確認する必要がある

ロールバック可能なデプロイ例



安全にインフラをCI/CDするために

ゴール

- 意図しない変更が含まれないことを確実にする
- できる限りデプロイの影響を最小化する

手段

1. 差分の確認
2. デプロイ前自動テストの整備
3. デプロイ後自動テストの整備
4. デプロイの影響を事前に検討

ここまでのまとめ

いくつかのプラクティスを実践すればインフラも安全にデプロイ可能
実はアプリのCI/CDと多くのポイントで共通

インフラ特有の難しさが顕在化するとき

- ステートフルなリソース
- 依存の多いリソース

どのように第一歩を踏み出すか？

千里の道も一歩から

インフラのCI/CD、どう始めるか？

Two-way door decision

後戻りできる施策なら悩まずに実行できる

うまくいかなければ戻せば良い
悩むくらいならとりあえずやってみよう

現実的なみちしるべ

1. CI/CD化対象のシステムを決める
2. 対象のインフラをできるだけIaC化する
3. CI/CDパイプラインを構築する
4. まずは差分の表示までを自動化　手動承認のあとにデプロイを実行
5. 運用がこなれてきたら、手動承認をなくしさらなる自動化へ

1. CI/CD対象システムの見定め

評価ポイント

- ✓ まずは小さなシステム・非商用のシステムから
- ✓ 変更頻度が高いシステム
- ✓ 自チームがオーナーシップを持てるシステム
- ✓ ステートフル or 依存の多いリソースは省いても良いかもしれない

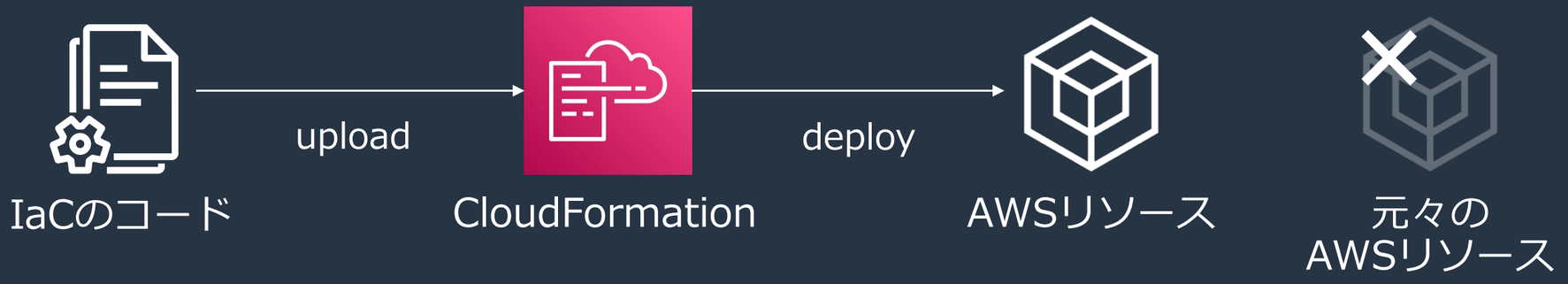
CI/CDは目的ではなく手段！導入のコスパを考えよう

現実的なみちしるべ

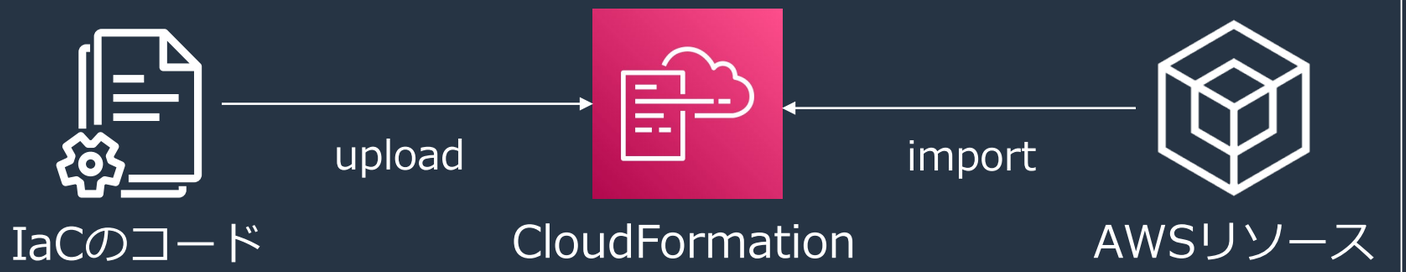
1. CI/CD化対象のシステムを決める
2. 対象のインフラをできるだけIaC化する
3. CI/CDパイプラインを構築する
4. まずは差分の表示までを自動化　手動承認のあとにデプロイを実行
5. 運用がこなれてきたら、手動承認をなくしさらなる自動化へ

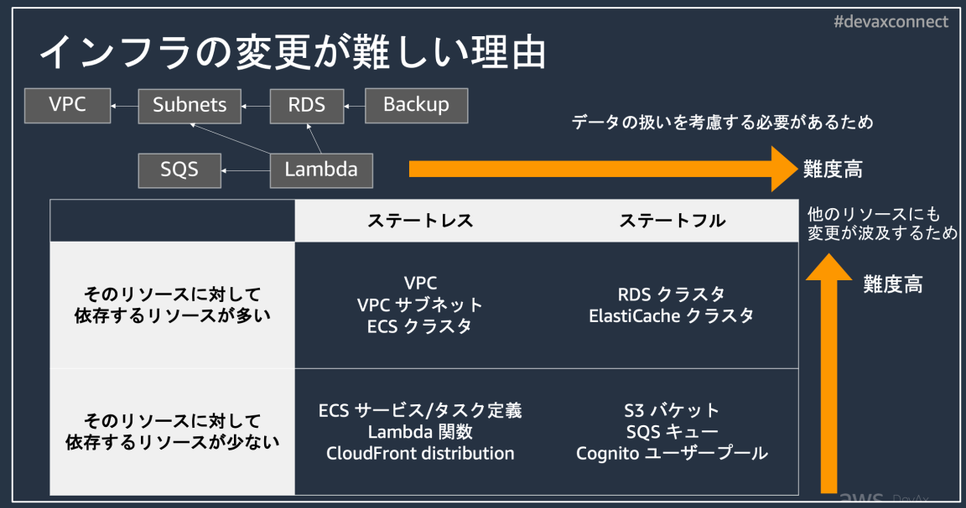
2. 既存インフラのIaC化 - 推奨方法

- ステートレスかつ依存の少ないリソース → IaCで作り直す



- その他のリソース管理下にインポートする



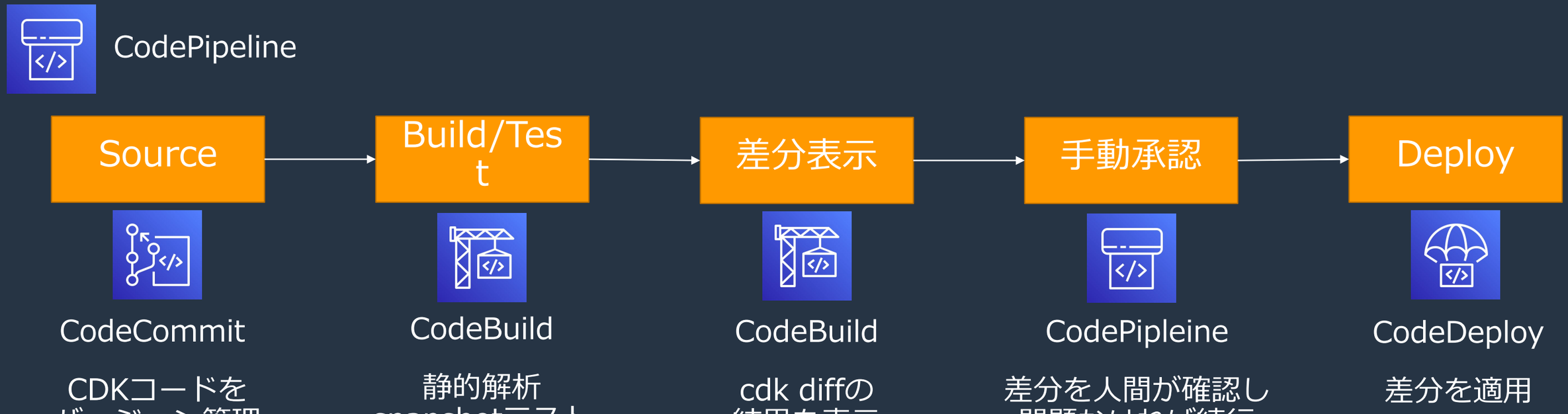


現実的なみちしるべ

1. CI/CD化対象のシステムを決める
2. 対象のインフラをできるだけIaC化する
3. CI/CDパイプラインを構築する
4. まずは差分の表示までを自動化　手動承認のあとにデプロイを実行
5. 運用がこなれてきたら、手動承認をなくしさらなる自動化へ

第一歩のインフラCI/CDパイプライン

例えばCDKの場合

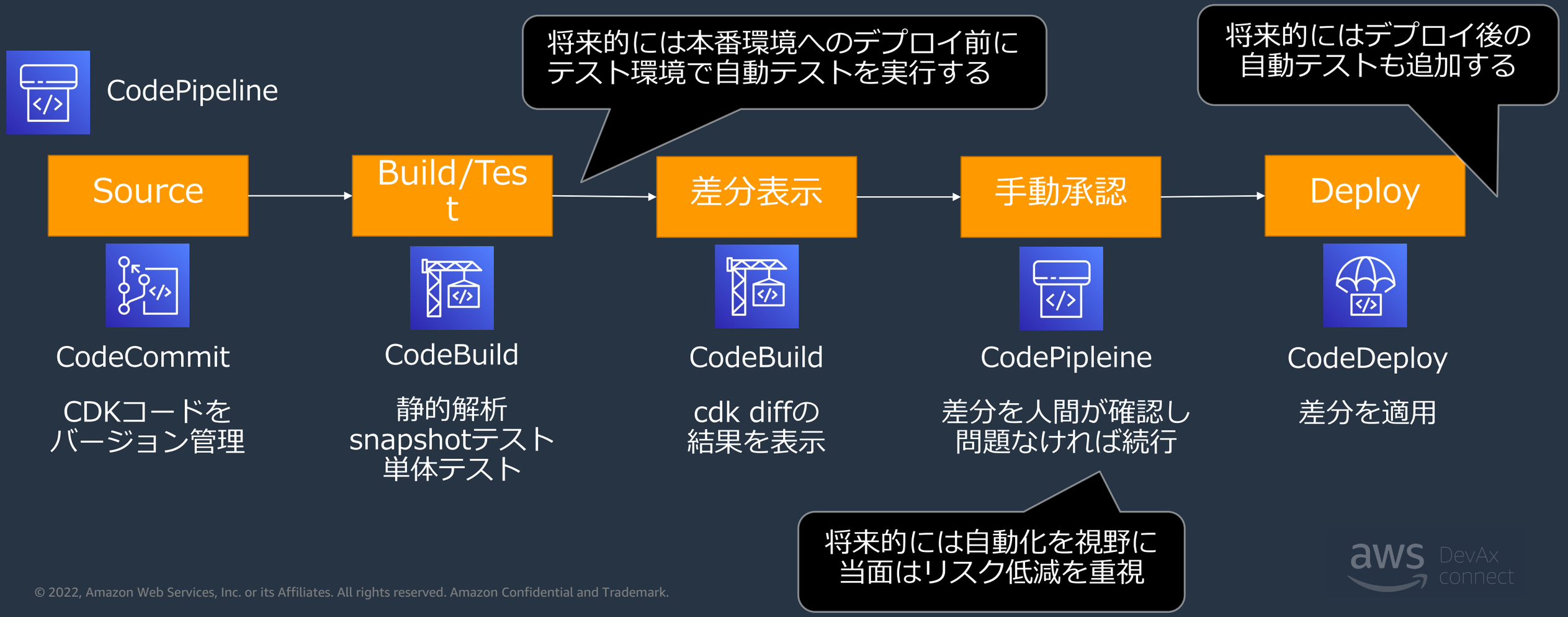


現実的なみちしるべ

1. CI/CD化対象のシステムを決める
2. 対象のインフラをできるだけIaC化する
3. CI/CDパイプラインを構築する
4. まずは差分の表示までを自動化　手動承認のあとにデプロイを実行
5. 運用がこなれてきたら、手動承認をなくしさらなる自動化へ

第一歩のインフラCI/CDパイプライン

例えばCDKの場合



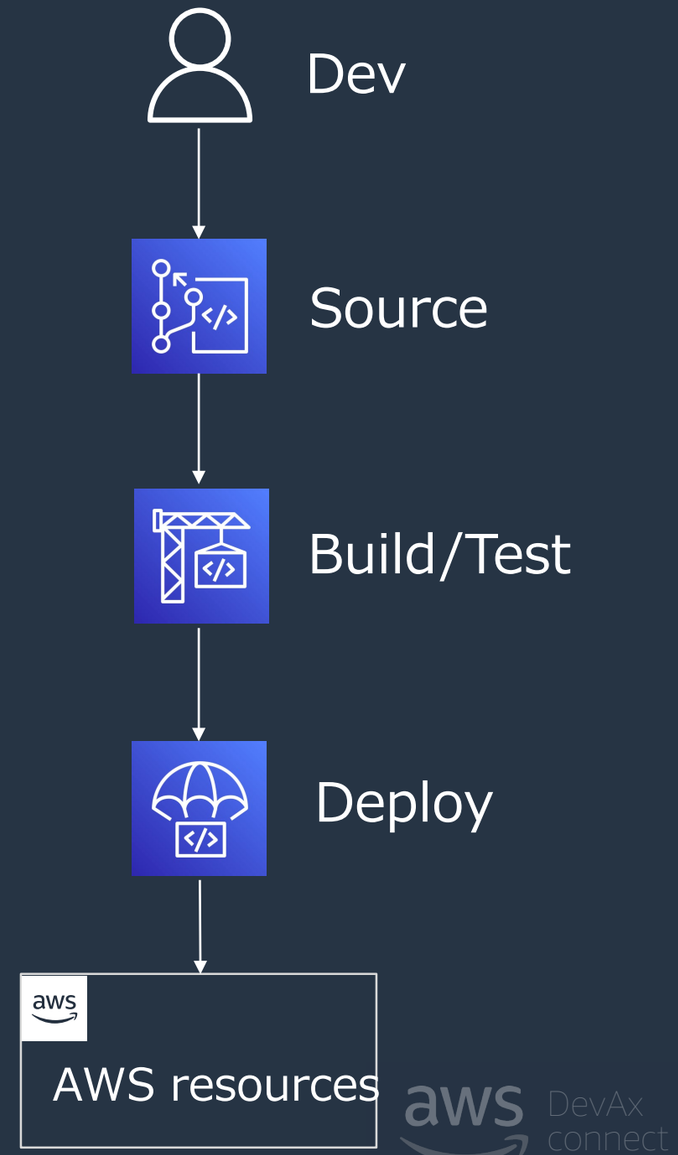
インフラのCI/CDが解決できること (再掲)

1. インフラを変更することへのハードルが高い
   → デプロイが定形作業になり、より気軽なものに

2. デプロイ手順書の運用・管理が煩雑
   → 自動化により手順書は最小限に

3. 強い権限を個別ユーザーに付与している
   → パイプライン用のロールを一元管理

4. インフラの現状を把握できていない
   → IaC化による解決 (副産物)



まとめ

- CI/CDは安全にイノベーションを加速できる　インフラも例外ではない
- いくつかのポイントを押さえれば、インフラも安全にCI/CDできる
- CI/CDはあくまでも手段の一つ　現状に最適な選択を採る事が大事
- 低リスクで始める方法があるので、一度検討してみては

Thank you!

Masashi Tomooka
Prototyping Engineer
Amazon Web Services Japan G.K.
mtomooka@amazon.co.jp @tmokmss

aws DevAx connect