



Terraform の開始方法: AWS CDK および AWS CloudFormation エキスパート
向けガイド

AWS 規範ガイド



AWS 規範ガイド: Terraform の開始方法: AWS CDK および AWS CloudFormation エキスパート向けガイド

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスは、Amazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は、Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

序章	1
CloudFormation および Terraform の用語	2
リソース	4
プロバイダー	6
Terraform エイリアスの使用	8
モジュール	12
モジュールを呼び出す	13
ルートモジュール	13
状態とバックエンド	15
データソース	18
変数、ローカル値、出力	20
変数	20
ローカル値	22
出力値	23
関数、式、メタ引数	25
関数	25
表現	25
メタ引数	26
よくある質問	33
の代わりに Terraform をいつ使用すべきですか CloudFormationか？	33
AWS CDK の代わりに をいつ使用すべきですか CloudFormation？	33
Terraform 設定 AWS CDK を生成する のようなツールはありますか？	33
Terraform の詳細については、どうすればよいですか？	33
関連リソース	34
AWS ドキュメント	34
その他のリソース	34
付録: Terraform 属性アクセスの例	35
リソース	35
データソース	35
モジュール	35
変数	35
ローカル	36
ドキュメント履歴	37
用語集	38

#	38
A	39
B	42
C	44
D	47
E	51
F	53
G	54
H	55
I	56
L	58
M	59
O	63
P	66
Q	68
R	69
S	71
T	75
U	76
V	77
W	77
Z	78
.....	lxxx

Terraform の開始方法: AWS CDK および AWS CloudFormation エキスパート向けガイド

Steven Guggenheimer、Amazon Web Services (AWS)

2024 年 3 月 ([ドキュメント履歴](#))

クラウドリソースのプロビジョニングの経験が の領域のみにある場合 AWS、[AWS Cloud Development Kit \(AWS CDK\)](#) および 以外のコードとしてのインフラストラクチャ (IaC) ツールの使用経験が限られている可能性があります [AWS CloudFormation](#)。実際、Hashicorp Terraform などの同様のツールは、まったく使い慣れていない可能性があります。ただし、クラウドジャーニーを深く進めるほど、Terraform に遭遇するのは避けられないほどです。その中核となる概念に慣れることは、お客様にとって決定的なことです。

Terraform、AWS CDK、および は CloudFormation 同様の目標を達成し、多くのコアコンセプトを共有していますが、いくつかの違いがあります。Terraform に初めて近づいた場合は、これらの違いに対応できない場合があります。結局のところ、スタック AWS CDK と CloudFormation スタックはすべて 内をベースとしているため AWS アカウント、その方法では、維持するほとんどのリソースと直接的な関係があります。Terraform は、単一のクラウドプロバイダーの環境には基づいていません。これにより、さまざまなプロバイダーを柔軟にサポートできますが、リモートロケーションまでリソースを維持する必要があります。

このガイドは、Terraform の背後にある主要概念をわかりやすく説明し、今後の IaC チャレンジの処理に役立ちます。Terraform がプロバイダー、モジュール、ステートファイルなどの概念を使用してリソースをプロビジョニングする方法に焦点を当てています。また、Terraform の概念と、AWS CDK および が同様のオペレーション CloudFormation を実行する方法とも対照的です。

Note

は、開発者がプログラムによるコーディング言語を使用して CloudFormation スタックをデプロイする AWS CDK のに役立ちます。を実行すると `cdk synth`、コードは テンプレートに変換 CloudFormation されます。その時点から、プロセスは AWS CDK と の間で同じになります CloudFormation。簡潔にするために、このガイドでは通常、AWS IaC プロセスを CloudFormation 用語で参照しますが、比較は 同じです AWS CDK。

CloudFormation および Terraform の用語

Terraform と AWS CDK および を比較する場合 CloudFormation、IaC のコア概念の説明に使用される用語が一貫していないため、IaC のコア概念の照合が困難になる可能性があります。以下に、これらの用語と、このガイドでそれらの用語がどのように参照されるかを示します。

- **スタック** – スタックは CI/CD パイプラインにデプロイされ、単一のユニットとして追跡可能な IaC です。この用語は 一般的ですが CloudFormation、Terraform はこの用語を実際には使用しません。Terraform スタックは、すべての子モジュールを含むデプロイされたルートモジュールです。ただし、モジュール という用語との混同を避けるために、このガイドでは スタック という用語を使用して両方のツールの単一のデプロイを記述します。
- **状態** – 状態は、IaC デプロイスタック内で現在追跡されているすべてのリソースと現在の設定です。[Terraform の状態とバックエンドについて](#) セクションで説明したように、Terraform は よりも多くの 状態 という用語を使用します CloudFormation。これは、状態の維持は Terraform でより目立つが、状態の追跡と更新は とも同様に重要であるためです CloudFormation。
- **IaC ファイル** – IaC ファイルは、Infrastructure as Code (IaC) 言語を含む単一のファイルです。CloudFormation は、単一の CloudFormation ファイルをテンプレート と呼びます。ただし、Terraform の [テンプレート](#) と [テンプレートファイル](#) はまったく異なります。Terraform の CloudFormation テンプレートと同等の設定ファイル と呼ばれます。このガイドの混乱を最小限に抑えるために、ファイルまたは IaC ファイル という用語は、CloudFormation テンプレートと Terraform 設定ファイルの両方を参照するために使用されます。

次の表は、CloudFormation と Terraform で使用される用語を比較したものです。このテーブルの目的は、類似点を表示することです。これらは one-to-one 比較ではありません。各概念は、CloudFormation と Terraform の間で少なくともわずかに異なります。概念については、このガイドの関連セクションで詳しく説明します。

CloudFormation 用語	Terraform 用語	このガイドのセクション
CDK インターフェイス (IBucket など)	データソース	Terraform データソースについて
変更セット	計画	Terraform モジュールについて
条件関数	条件式	Terraform 関数、式、メタ引数について

CloudFormation 用語	Terraform 用語	このガイドのセクション
DependsOn 属性	depends_on メタ引数	Terraform 関数、式、メタ引数について
組み込み関数	関数	Terraform 関数、式、メタ引数について
モジュール	モジュール	Terraform モジュールについて
出力	出力値	Terraform 変数、ローカル値、出力について
パラメータ	変数	Terraform 変数、ローカル値、出力について
レジストリ	プロバイダー	Terraform プロバイダーについて
テンプレート	設定ファイル	すべて

Terraform リソースについて

AWS CloudFormation と Terraform の両方が存在する主な理由は、クラウドリソースの作成とメンテナンスです。しかし、クラウドリソースとは具体的に何でしょうか。リソースと CloudFormation Terraform リソースは同じものですか？ 答えは「はい」と「いいえ」です。このガイドでは、CloudFormation を使用して Amazon Simple Storage Service (Amazon S3) バケットを作成する例を示します。

次の CloudFormation コード例では、サンプル Amazon S3 バケットを作成します。

```
{
  "myS3Bucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": "my-s3-bucket",
      "BucketEncryption": {
        "ServerSideEncryptionConfiguration": [
          {
            "ServerSideEncryptionByDefault": {
              "SSEAlgorithm": "AES256"
            }
          }
        ]
      },
      "PublicAccessBlockConfiguration": {
        "BlockPublicAcls": true,
        "BlockPublicPolicy": true,
        "IgnorePublicAcls": true,
        "RestrictPublicBuckets": true
      },
      "VersioningConfiguration": {
        "Status": "Enabled"
      }
    }
  }
}
```

次の Terraform コード例では、同じ Amazon S3 バケットを作成します。

```
resource "aws_s3_bucket" "myS3Bucket" {
  bucket = "my-s3-bucket"
}
```



```
}

resource "aws_s3_bucket_server_side_encryption_configuration" "bucketencryption" {
  bucket = aws_s3_bucket.myS3Bucket.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

resource "aws_s3_bucket_public_access_block" "publicaccess" {
  bucket                = aws_s3_bucket.myS3Bucket.id
  block_public_acls     = true
  block_public_policy   = true
  ignore_public_acls   = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.myS3Bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

Terraform の場合、プロバイダーはリソースを定義し、デベロッパーはそれらのリソースを宣言して設定します。プロバイダーは、このガイドが次のセクションで説明する概念です。Terraform の例では、S3 バケットの設定のいくつかに対して完全に別々のリソースを作成します。設定用に個別のリソースを作成することは、Terraform AWS プロバイダーが AWS リソースを処理する方法に必ずしも典型的なものではありません。ただし、この例は重要な違いを示しています。CloudFormation リソースは [CloudFormation リソース仕様](#) によって厳密に定義されていますが、Terraform にはそのような要件はありません。Terraform では、リソースの概念は少し不明瞭です。

ツールは、単一のリソースとは何かを定義する正確なガードレールに関して異なる場合がありますが、一般的に、クラウドリソースとは、クラウドに存在し、作成、更新、または削除できる特定のエンティティです。したがって、関係するリソースの数に関係なく、前の 2 つの例ではどちらも、内でまったく同じ設定でまったく同じモノを作成します AWS アカウント。

Terraform プロバイダーについて

Terraform では、プロバイダーはクラウドプロバイダー、サードパーティーツール、その他の APIs。で Terraform AWSを使用するには、AWS リソースとやり取りする[AWS プロバイダー](#) を使用します。

[AWS CloudFormation レジストリ](#)を使用してデプロイスタックにサードパーティーの拡張機能を組み込むことがない場合、Terraform [プロバイダー](#)は慣れている可能性があります。CloudFormation はネイティブであるため AWS、リソースの AWS プロバイダーはデフォルトで既に存在します。一方、Terraform には単一のデフォルトプロバイダーがないため、特定のリソースのオリジンについては何も想定できません。つまり、Terraform 設定ファイルで最初に宣言する必要があるのは、リソースがどこに向かっていて、どのようにそこに到達するかということです。

この区別により、には存在しない Terraform の複雑さがさらに増します CloudFormation。ただし、その複雑さにより柔軟性が向上します。1 つの Terraform モジュール内で複数のプロバイダーを宣言でき、作成された基盤となるリソースは、同じデプロイレイヤーの一部として相互にやり取りできます。

これは、さまざまな方法で役立ちます。プロバイダーは必ずしも個別のクラウドプロバイダー用である必要はありません。プロバイダーは、クラウドリソースの任意のソースを表すことができます。例えば、Amazon Elastic Kubernetes Service (Amazon EKS) があるとします。Amazon EKS クラスターをプロビジョニングするときは、Helm チャートを使用してサードパーティーの拡張機能を管理し、Kubernetes 自体を使用してポッドリソースを管理できます。AWS、[Helm](#)、および [Kubernetes](#) にはそれぞれ独自の Terraform プロバイダーがあるため、これらのリソースをすべて同時にプロビジョニングして統合し、それらの間で値を渡すことができます。

Terraform の次のコード例では、AWS プロバイダーは Amazon EKS クラスターを作成し、結果の Kubernetes 設定情報が Helm プロバイダーと Kubernetes プロバイダーの両方に渡されます。

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = ">= 4.33.0"
    }

    helm = {
      source = "hashicorp/helm"
      version = "2.12.1"
    }
  }
}
```

```
kubernetes = {
  source = "hashicorp/kubernetes"
  version = "2.26.0"
}
}
required_version = ">= 1.2.0"
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_eks_cluster" "example_0" {
  name      = "example_0"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids               = var.subnet_ids
  }
}

locals {
  host      = aws_eks_cluster.example_0.endpoint
  certificate = base64decode(aws_eks_cluster.example_0.certificate_authority.data)
}

provider "helm" {
  kubernetes {
    host = local.host
    cluster_ca_certificate = local.certificate
    # exec allows for an authentication command to be run to obtain user
    # credentials rather than having them stored directly in the file
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_0.name]
      command     = "aws"
    }
  }
}

provider "kubernetes" {
```

```
host          = local.host
cluster_ca_certificate = local.certificate
exec {
  api_version = "client.authentication.k8s.io/v1beta1"
  args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_0.name]
  command     = "aws"
}
}
```

2つの IaC ツールに関しては、プロバイダーに関するトレードオフがあります。Terraform は、デプロイを駆動するエンジンである外部に配置されたプロバイダーパッケージに完全に依存します。は、すべての主要な AWS プロセスを CloudFormation 内部でサポートしています。では CloudFormation、サードパーティーの拡張機能を組み込む場合にのみ、サードパーティープロバイダーについて心配する必要があります。各アプローチには長所と短所があります。どちらが適しているかは、このガイドの範囲外ですが、両方のツールを評価する際には違いを覚えておくことが重要です。

Terraform エイリアスの使用

Terraform では、各プロバイダーにカスタム設定を渡すことができます。同じモジュール内で複数のプロバイダー設定を使用する場合はどうなりますか？ その場合は、[エイリアス](#)を使用する必要があります。エイリアスは、リソースレベルまたはモジュールレベルごとに使用するプロバイダーを選択するのに役立ちます。同じプロバイダーのインスタンスが複数ある場合は、エイリアスを使用してデフォルト以外のインスタンスを定義します。例えば、デフォルトのプロバイダーインスタンスは特定の `aws` リージョン、エイリアスを使用して代替リージョンを定義します。

次の Terraform の例は、エイリアスを使用して異なる `aws` でバケットをプロビジョニングする方法を示しています `aws` リージョン。プロバイダーのデフォルトリージョンは `us-west-2` ですが、`us-east-2` でリソースをプロビジョニングするために `east` エイリアスを使用できます `us-east-2`。

```
provider "aws" {
  region = "us-west-2"
}

provider "aws" {
  alias   = "east"
  region = "us-east-2"
}
```

```
resource "aws_s3_bucket" "myWestS3Bucket" {
  bucket = "my-west-s3-bucket"
}

resource "aws_s3_bucket" "myEastS3Bucket" {
  provider = aws.east
  bucket   = "my-east-s3-bucket"
}
```

前の例に示すように、メタ引数 `alias` とともに `provider` を使用する場合、特定のリソースに対して別のプロバイダー設定を指定できます。1つのスタック内の複数の AWS リージョンにリソースをプロビジョニングすることは、ほんの始まりです。エイリアスプロバイダーは、さまざまな点で便利に使い分けることができます。

例えば、一度に複数の Kubernetes クラスターをプロビジョニングするのが非常に一般的です。エイリアスを使用すると、追加の Helm プロバイダーと Kubernetes プロバイダーを設定して、Amazon EKS リソースごとにこれらのサードパーティツールを異なる方法で使用できるようになります。次の Terraform コード例は、エイリアスを使用してこのタスクを実行する方法を示しています。

```
resource "aws_eks_cluster" "example_0" {
  name      = "example_0"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[0]
  }
}

resource "aws_eks_cluster" "example_1" {
  name      = "example_1"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[1]
  }
}

locals {
  host      = aws_eks_cluster.example_0.endpoint
  certificate = base64decode(aws_eks_cluster.example_0.certificate_authority.data)
```

```
host1      = aws_eks_cluster.example_1.endpoint
certificate1 = base64decode(aws_eks_cluster.example_1.certificate_authority.data)
}

provider "helm" {
  kubernetes {
    host          = local.host
    cluster_ca_certificate = local.certificate
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_0.name]
      command     = "aws"
    }
  }
}

provider "helm" {
  alias = "helm1"
  kubernetes {
    host          = local.host1
    cluster_ca_certificate = local.certificate1
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_1.name]
      command     = "aws"
    }
  }
}

provider "kubernetes" {
  host          = local.host
  cluster_ca_certificate = local.certificate
  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_0.name]
    command     = "aws"
  }
}

provider "kubernetes" {
  alias = "kubernetes1"
}
```

```
host                = local.host1
cluster_ca_certificate = local.certificate1
exec {
  api_version = "client.authentication.k8s.io/v1beta1"
  args        = ["eks", "get-token", "--cluster-name",
aws_eks_cluster.example_1.name]
  command     = "aws"
}
}
```

Terraform モジュールについて

Infrastructure as Code (IaC) の領域では、モジュールは自己完結型のコードブロックであり、再利用のために分離され、一緒にパッケージ化されています。モジュールの概念は、Terraform 開発の不可避の側面です。詳細については、Terraform ドキュメントの「[モジュール](#)」を参照してください。はモジュール AWS CloudFormation もサポートしています。詳細については、AWS クラウド運用と移行ブログの「[モジュールの導入 AWS CloudFormation](#)」を参照してください。

Terraform と のモジュールの主な違いは、CloudFormation モジュールは特別なリソースタイプ () を使用してインポートされること CloudFormation で `AWS::CloudFormation::ModuleVersion`。Terraform では、すべての設定にルートモジュールと呼ばれるモジュールが 少なくとも 1 つあります。main.tf ファイル内の Terraform リソース、または Terraform 設定ファイル内のファイルは、ルートモジュールにあると見なされます。その後、ルートモジュールはスタック内に含めるために他のモジュールを呼び出すことができます。次の例は、オープンソースの [eks](#) モジュールを使用して Amazon Elastic Kubernetes Service (Amazon EKS) クラスタをプロビジョニングするルートモジュールを示しています。

```
terraform {
  required_providers {
    helm = {
      source = "hashicorp/helm"
      version = "2.12.1"
    }
  }
  required_version = ">= 1.2.0"
}

module "eks" {
  source = "terraform-aws-modules/eks/aws"
  version = "20.2.1"
  vpc_id = var.vpc_id
}

provider "helm" {
  kubernetes {
    host = module.eks.cluster_endpoint
    cluster_ca_certificate =
      base64decode(module.eks.cluster_certificate_authority_data)
  }
}
```


上記の設定ファイルにプロバイダーが含まれていないことに AWS 気付いたかもしれません。これは、モジュールが自己完結型で、独自のプロバイダーを含めることができるためです。Terraform プロバイダーはグローバルであるため、子モジュールのプロバイダーをルートモジュールで使用できません。ただし、これはすべてのモジュール値に当てはまるわけではありません。モジュール内の他の内部値は、デフォルトでそのモジュールのみにスコープされ、ルートモジュールでアクセスできるようにするには出力として宣言する必要があります。オープンソースモジュールを活用して、スタック内のリソース作成を簡素化できます。例えば、eks モジュールは EKS クラスターをプロビジョニングするだけでなく、完全に機能する Kubernetes 環境をプロビジョニングします。これを使用すると、eks モジュールの設定がニーズに合っている限り、数十行のコードを追加で書き込む必要がなくなります。

モジュールを呼び出す

Terraform のデプロイ中に実行する主な Terraform CLI コマンドの 2 つは、[Terraform init](#) と [Terraform apply](#) です。terraform init コマンドが実行するデフォルトのステップの 1 つは、すべての子モジュールを検索し、依存関係として .terraform/modules ディレクトリにインポートすることです。開発中、外部ソースモジュールを追加するたびに、apply コマンドを使用する前にを再初期化する必要があります。Terraform モジュール への参照が聞こえると、このディレクトリ内のパッケージが参照されます。厳密に言うと、コードで宣言するモジュールは呼び出し元のモジュールであるため、実際には、モジュールキーワードは実際のモジュールを呼び出し、依存関係として保存されます。

このようにして、呼び出し元のモジュールは、デプロイ時に置き換えられるモジュール全体のより簡潔な代表として機能します。このアイデアを活用するには、スタック内に独自のモジュールを作成して、任意の基準を使用してリソースを論理的に分離します。これを行う最終目標は、スタックの複雑さを軽減することであることに留意してください。モジュール間でデータを共有するには、モジュール内からそのデータを出力する必要があるため、モジュールに過度に依存すると、モノが過度に複雑になることがあります。

ルートモジュール

すべての Terraform 設定には少なくとも 1 つのモジュールがあるため、最も対処するモジュールのモジュールプロパティであるルートモジュールを調べるのに役立ちます。Terraform プロジェクトで作業するたびに、ルートモジュールは最上位ディレクトリ内のすべての .tf (または .tf.json) ファイルで構成されます。その最上位ディレクトリ terraform apply で を実行すると、Terraform はそこで見つかったすべての .tf ファイルの実行を試みます。サブディレクトリ内のファイルは、これらの最上位設定ファイルの 1 つで呼び出されない限り、無視されます。

これにより、コードの構造に柔軟性がもたらされます。また、複数のファイルが1つのプロセスに関与している可能性があるため、Terraform デプロイをファイルとしてではなくモジュールとして参照する方が正確である理由でもあります。Terraform がベストプラクティスとして推奨する [標準のモジュール構造](#) があります。ただし、最上位ディレクトリに .tf ファイルを配置すると、残りのファイルとともに実行されます。実際には、モジュール内のすべての最上位 .tf ファイルは、 の実行時にデプロイされます terraform apply。Terraform が最初に実行されるのはどのファイルですか？ この質問に対する回答は非常に重要です。

Terraform は、初期化後およびスタックデプロイ前に一連のステップを実行します。まず、既存の設定が分析され、次に [依存関係グラフ](#) が作成されます。依存関係グラフは、どのリソースをどの順序で呼び出すかを決定します。例えば、他のリソースで参照されるプロパティを含むリソースは、依存リソースの前に処理されます。同様に、 depends_on パラメータを使用して依存関係を明示的に宣言するリソースは、指定したリソースの後に処理されます。Terraform は、可能な限り並列処理を実装し、非依存リソースを同時に処理できます。 [terraform graph コマンド](#) を使用して、デプロイ前に [依存関係グラフ](#) を表示できます。

依存関係グラフが作成されると、Terraform はデプロイ中に何を必要とするかを決定します。依存関係グラフを最新の状態ファイルと比較します。このプロセスの結果は計画 と呼ばれ、 [変更セット](#) と CloudFormation 非常によく似ています。現在の計画は、 [Terraform Plans](#) コマンドを使用して確認できます。

ベストプラクティスとして、標準モジュール構造にできるだけ近づくことをお勧めします。設定ファイルが長くなりすぎて効率的に管理できず、論理的な分離によって管理が簡素化される可能性がある場合は、コードを複数のファイルに分散できます。依存関係グラフと計画プロセスが、スタックをできるだけ効率的に実行するためにどのように機能するかに注意してください。

Terraform の状態とバックエンドについて

Infrastructure as Code (IaC) における最も重要な概念の 1 つは、状態 の概念です。IaC サービスは状態を維持します。これにより、デプロイするたびにリソースを再作成することなく、IaC ファイル内のリソースを宣言できます。IaC ファイルは、デプロイの最後にすべてのリソースの状態を文書化し、その状態を次のデプロイで宣言されたターゲット状態と比較できるようにします。したがって、現在の状態に という名前の Amazon Simple Storage Service (Amazon S3) バケットが含まれmy-s3-bucket、受信する変更にも同じバケットが含まれている場合、新しいプロセスでは、すべての新しいバケットを作成するのではなく、既存のバケットに変更が適用されます。

次の表は、一般的な IaC 状態プロセスの例を示しています。

現在の状態	ターゲットの状態	アクション
という名前の S3 バケットはありません my-s3-bucket	という名前の S3 バケット my-s3-bucket	という名前の S3 バケットを作成する my-s3-bucket
my-s3-bucket バケットのバージョニングが設定されていない	my-s3-bucket バケットのバージョニングが設定されていない	アクションなし
my-s3-bucket バケットのバージョニングが設定されていない	my-s3-bucket バケットのバージョニングが設定されている	バケットmy-s3-bucket のバージョニングを設定する
my-s3-bucket バケットのバージョニングが設定されている	という名前の S3 バケットはありません my-s3-bucket	削除の試行 my-s3-bucket

AWS CloudFormation と Terraform のトラック状態に関するさまざまな方法を理解するには、内で CloudFormation ホストされている と AWS クラウド Terraform が基本的にリモートであるという 2 つのツールの最初の基本的な違いを覚えておくことが重要です。この事実により、CloudFormation は内部的に状態を維持できます。CloudFormation コンソールに移動して特定のスタックのイベント履歴を表示できますが、CloudFormation サービス自体によって状態ルールが適用されます。

特定のリソースに対して CloudFormation で動作する 3 つのモードは Create、Update、および Delete です。現在のモードは、前回のデプロイで何が起きたかに基づいて決定され、それ以外の

方法では影響されません。決定されたモードに影響を与えるために CloudFormation リソースを手動で更新できますが、「このリソースについては、Create モードで動作する CloudFormation」というコマンドをに渡すことはできません。

Terraform はでホストされていないため AWS クラウド、状態を維持するプロセスはより設定可能である必要があります。このため、[Terraform の状態](#)は、自動的に生成された状態ファイル内で維持されます。Terraform デベロッパーは、よりもはるかに直接的に状態に対処する必要があります CloudFormation。覚えておくべき重要なことは、追跡状態が両方のツールで同じくらい重要であることです。

デフォルトでは、Terraform 状態ファイルは、Terraform スタックを実行するメインディレクトリの最上位にローカルに保存されます。ローカル開発環境から terraform apply コマンドを実行すると、Terraform が状態をリアルタイムで維持するために使用する terraform.tfstate ファイルを生成することがわかります。良い場合も悪い場合も、これにより Terraform の状態をよりもはるかに細かく制御できます CloudFormation。状態ファイルを直接更新しないでください、デプロイ間で状態を更新する Terraform CLI コマンドがいくつか実行できます。例えば、[Terraform のインポート](#)では、Terraform の外部で作成されたリソースをデプロイスタックに追加できます。逆に、[Terraform ステート rm を実行することで、状態からリソースを削除](#)できます。

Terraform が状態をどこかに保存する必要があるという事実は、バックエンドには適用されない別の概念につながります CloudFormation。[Terraform バックエンド](#)は、デプロイ後に Terraform スタックがステートファイルを保存する場所です。これは、新しいデプロイが開始されたときに状態ファイルを検索する場所でもあります。スタックをローカルで実行するときは、前述のように、Terraform 状態のコピーを最上位のローカルディレクトリに保持できます。これはローカルバックエンドと呼ばれます。

継続的インテグレーションおよび継続的デプロイ (CI/CD) 環境向けに開発する場合、ローカル状態ファイルは、バージョン管理の対象外となるように、通常 .gitignore ファイルに含まれます。次に、パイプライン内にローカル状態ファイルは存在しません。正しく動作するためには、パイプラインステージで正しい状態ファイルをどこかで見つける必要があります。そのため、Terraform 設定ファイルにはバックエンドブロックが含まれていることがよくあります。バックエンドブロックは、状態ファイルを見つけるために独自の最上位ディレクトリのどこかを探す必要があることを Terraform スタックに示します。

Terraform バックエンドは、[Amazon S3 バケット](#)、[API エンドポイント](#)、[リモート Terraform ワークスペース](#)など、ほぼどこにでも配置できます。<https://developer.hashicorp.com/terraform/language/settings/backends/remote> Amazon S3 バケットに保存されている Terraform バックエンドの例を次に示します。

```
terraform {
  backend "s3" {
    bucket = "my-s3-bucket"
    key    = "state-file-folder"
    region = "us-east-1"
  }
}
```

Terraform 設定ファイルに機密情報を保存しないように、バックエンドは部分的な設定もサポートしています。前の例では、バケットへのアクセスに必要な認証情報は設定に存在しません。認証情報は、環境変数から取得することも、などの他の方法を使用して取得することもできます AWS Secrets Manager。詳細については、[「AWS Secrets Manager と HashiCorp Terraform を使用した機密データの保護」](#)を参照してください。

一般的なバックエンドシナリオは、テスト目的でローカル環境で使用されるローカルバックエンドです。terraform.tfstate ファイルは .gitignore ファイルに含まれているため、リモートリポジトリにプッシュされません。その後、CI/CD パイプライン内の各環境は独自のバックエンドを維持します。このシナリオでは、複数のデベロッパーがこのリモート状態にアクセスできる可能性があるため、状態ファイルの整合性を保護する必要があります。複数のデプロイが実行されていて、状態が同時に更新されている場合、状態ファイルが破損している可能性があります。このため、ローカル以外のバックエンドでは、通常、状態ファイルはデプロイ中に[ロック](#)されます。

Terraform データソースについて

デプロイスタックが既存のリソースからのデータに依存するのは非常に一般的です。ほとんどの IaC ツールには、他のプロセスによって作成されたリソースをインポートする方法があります。これらのインポートされたリソースは通常読み取り専用であり ([IAM ロール](#) は注目すべき例外ですが)、スタック内のリソースに必要なデータにアクセスするために使用します。ではリソースのインポート AWS CloudFormation が許可されますが、この概念はを確認することでより適切に説明できます AWS Cloud Development Kit (AWS CDK)。

は、デベロッパーが既存のプログラミング言語を使用してテンプレートを生成する AWS CDK CloudFormation のに役立ちます。AWS CDK オペレーションの最終結果は、にインポートされたりソースです CloudFormation。ただし、で使用される構文 AWS CDK により、Terraform との比較が容易になります。を使用してリソースをインポートする例を次に示します AWS CDK。

```
const importedBucket: IBucket = Bucket.fromBucketAttributes(  
    scope,  
    "imported-bucket",  
    {  
        bucketName: "My_S3_Bucket"  
    }  
);
```

インポートされたリソースは通常、同じ種類の新しいリソースを作成するために使用するのと同じクラスの静的メソッドを呼び出すことによって作成されます。を呼び出すと新しいリソースが作成され、を呼び出すと既存のリソースが `Bucket.fromBucketAttributes(...インポートnew Bucket(...)` されます。バケットのプロパティのサブセットを関数に渡すと、は適切なバケットを見つける AWS CDK ことができます。ただし、もう 1 つの違いは、新しいバケットを作成すると、`Bucket` クラスのフルインスタンスが返され、その中にすべてのプロパティとメソッドが使用可能になることです。リソースをインポートすると `IBucket`、が持つ `Bucket` 必要があるプロパティのみを含むタイプである が返されます。外部スタックからリソースをインポートすることはできませんが、リソースで実行できる操作のオプションは限られています。

Terraform では、[データソース](#) を使用して同様の目標を達成します。定義されたほとんどの Terraform リソースには、付随するデータソースがあります。以下は、Terraform S3 バケットリソースとそれに続く対応するデータソースの例です。

```
# S3 Bucket resource:  
resource "aws_s3_bucket" "My_S3_Bucket" {
```

```
bucket = "My_S3_Bucket"
}

# S3 Bucket data source:
data "aws_s3_bucket" "My_S3_Bucket" {
  bucket = "My_S3_Bucket"
}
```

これら 2 つの項目の違いは、名前のプレフィックスのみです。データソースの [ドキュメント](#) に示すように、データソースに渡すことができるパラメータはリソースよりも少なくなります。これは、リソースがこれらのパラメータを使用して新しい S3 バケットのすべてのプロパティを宣言し、データソースには既存のリソースのデータを一意に識別してインポートするための十分な情報が必要なためです。

Terraform リソースとデータソースの構文の類似性は便利ですが、問題になる可能性もあります。初心者の Terraform 開発者が誤ってデータソースを構成で使用しているのは一般的です。Terraform データソースは常に読み取り専用です。これらは、読み取りアクション (別のリソースに ID 名を指定するなど) に対応するリソースの代わりに使用できます。ただし、書き込みアクションに使用することはできません。これにより、基盤となるリソースの一部が根本的に変更されます。このため、Terraform データソースは基盤となるリソースのクローンバージョンと考えることができます。

前の AWS CDK IBucket の例と同様に、データソースは読み取り専用シナリオに役立ちます。既存のリソースからデータを取得する必要があるが、スタック内でそのリソースを維持する必要がない場合は、データソースを使用します。この良い例は、アカウントのデフォルト VPC を使用する Amazon EC2 インスタンスを作成する場合です。その VPC は既に存在するため、必要なのはデータの取得だけです。次のコードサンプルは、データを使用してターゲット VPC を識別する方法を示しています。

```
data "aws_vpc" "default" {
  default = true
}

resource "aws_instance" "instance1" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
  subnet_id    = data.aws_vpc.default.main_route_table_id
}
```

Terraform 変数、ローカル値、出力について

変数は、コードブロック内のプレースホルダーを許可することで、コードの柔軟性を高めます。変数は、コードが再利用されるたびに異なる値を表すことができます。Terraform は、モジュラスコープによって変数タイプを区別します。入力変数はモジュールに挿入できる外部値であり、出力値は外部で共有できる内部値であり、ローカル値は常に元の範囲内にとどまります。

変数

AWS CloudFormation は [パラメータ](#) を使用して、あるスタックのデプロイから次のスタックへの設定とリセットが可能なカスタム値を表します。同様に、Terraform は [入力変数](#)、または変数 を使用します。変数は Terraform 設定ファイル内の任意の場所で宣言でき、通常は必要なデータ型またはデフォルト値で宣言されます。次の 3 つの式はすべて有効な Terraform 変数宣言です。

```
variable "thing_i_made_up" {
  type = string
}

variable "random_number" {
  default = 5
}

variable "dogs" {
  type = list(object({
    name = string
    breed = string
  }))

  default = [
    {
      name = "Sparky",
      breed = "poodle"
    }
  ]
}
```

設定内で Sparky の品種にアクセスするには、変数を使用します `var.dogs[0].breed`。変数にデフォルトがなく、nullable として分類されていない場合は、変数の値をデプロイごとに設定する必要があります。それ以外の場合は、変数に新しい値を設定することはオプションです。ルートモジュー

ルでは、[コマンドライン](#)、[環境変数](#)、または `terraform.tfvars` ファイルで現在の`変数値`を設定できます。<https://developer.hashicorp.com/terraform/language/values/variables#variable-definitions-tfvars-files>次の例は、モジュールの最上位ディレクトリに保存されている `terraform.tfvars` ファイルに変数値を入力する方法を示しています。

```
# terraform.tfvars
dogs = [
  {
    name = "Sparky",
    breed = "poodle"
  },
  {
    name = "Fluffy",
    breed = "chihuahua"
  }
]

random_number = 7

thing_i_made_up = "Kabibble"
```

この例`dogs`の `terraform.tfvars` ファイルの の値は、変数宣言のデフォルト値を上書きします。子モジュール内で変数を宣言する場合は、次の例に示すように、モジュール宣言ブロック内で変数値を直接設定できます。

```
module "my_custom_module" {
  source      = "modulesource/custom"
  version     = "0.0.1"
  random_number = 8
}
```

変数を宣言するときに使用できる他の引数には、次のようなものがあります。

- `sensitive` – これを `true` に設定すると、変数値が Terraform プロセス出力で公開され `true` なくなります。
- `nullable` – これを `true` に設定すると `true`、変数に値を指定できなくなります。これは、デフォルトが設定されていない変数に便利です。
- `description` – スタックのメタデータに変数の説明を追加します。
- `validation` – 変数の検証ルールを設定します。

Terraform 変数の最も便利な側面の 1 つは、変数宣言内に 1 つ以上の検証オブジェクトを追加する機能です。検証オブジェクトを使用して、変数が渡す必要がある条件を追加したり、そうしないとデプロイが失敗したりすることができます。条件に違反するたびに表示するカスタムエラーメッセージを設定することもできます。

例えば、チームのメンバーが実行する Terraform 設定ファイルを設定します。スタックをデプロイする前に、チームメンバーは terraform.tfvars ファイルを作成して、重要な設定値を設定する必要があります。これらを思い出すには、次のような操作を行います。

```
variable "important_config_setting" {
  type = string

  validation {
    condition     = length(var.important_config_setting) > 0
    error_message = "Don't forget to create the terraform.tfvars file!"
  }

  validation {
    condition     = substr(var.important_config_setting, 0, 7) == "prefix-"
    error_message = "Remember that the value always needs to start with 'prefix-'"
  }
}
```

この例に示すように、1 つの変数内に複数の条件を設定できます。Terraform は、失敗した条件のエラーメッセージのみを表示します。これにより、変数値にあらゆる種類のルールを適用できます。変数値が原因でパイプラインに障害が発生した場合は、その理由が正確にわかります。

ローカル値

モジュール内にエイリアスを設定する値がある場合は、更新されないデフォルトの変数を宣言するのではなく、locals キーワードを使用します。名前が示すように、locals ブロックには、その特定のモジュールに内部的にスコープされた用語が含まれています。リソース名で使用する変数値にプレフィックスを追加するなど、文字列値を変換する場合は、ローカル値を使用することをお勧めします。次の例に示すように、1 つの locals ブロックでモジュールのすべてのローカル値を宣言できます。

```
locals {
  moduleName      = "My Module"
  localConfigId = concat("prefix-", var.important_config_setting)
```

```
}
```

値にアクセスすると、`locals`キーワードが のように単数になることに注意してください `local.LocalConfigId`。

出力値

Terraform 入力変数が CloudFormation パラメータに似ている場合、[Terraform 出力値](#) は [CloudFormation 出力](#) に似ていると言えるでしょう。どちらもデプロイスタック内から値を公開するために使用されます。ただし、Terraform モジュールはツールのファブリックに深く組み込まれているため、Terraform 出力値を使用して、モジュール内のすべてのモジュールが同じデプロイスタック内にある場合でも、モジュール内の値を親モジュールまたは他の子モジュールに公開します。2つのカスタムモジュールを構築し、最初のモジュールが2番目のモジュールのID値にアクセスする必要がある場合は、次のoutputブロックを2番目のモジュールに追加する必要があります。

```
output "module_id" {
  value = local.module_id
}
Then in the first module you could use it like this:
module "first_module" {
  source = "path/to/first/module"
}

resource "example_resource" "example_resource_name" {
  module_id = module.first_module.module_id
}
```

Terraform 出力値は同じスタック内で使用できるため、outputブロックの `sensitive` 属性を使用して、スタック出力に値が表示されないようにすることもできます。さらに、outputブロックは、変数が `precondition` ブロックを使用するのと同じ方法で `validation` ブロックを使用できます。これにより、変数が特定のルールのセットに従うようになります。これにより、デプロイを続ける前に、モジュール内のすべての値が期待どおりに存在していることを確認できます。

```
output "important_config_setting" {
  value = var.important_config_setting

  precondition {
    condition      = length(var.important_config_setting) > 0
    error_message = "You forgot to create the terraform.tfvars file again."
  }
}
```

```
}
```

Terraform 関数、式、メタ引数について

一般的なプログラミング言語ではなく宣言型設定ファイルを使用する IaC ツールの 1 つの批判は、カスタムプログラムロジックの実装が困難になることです。Terraform 設定では、この問題は関数、式、メタ引数を使用して対処されます。

関数

コードを使用してインフラストラクチャをプロビジョニングする大きな利点の 1 つは、一般的なワークフローを保存して繰り返し再利用し、多くの場合、毎回異なる引数を渡すことです。Terraform 関数は AWS CloudFormation [組み込み関数](#) に似ていますが、構文はプログラムによる言語での関数の呼び出し方法に似ています。このガイドの例では、[substr](#)、[concat](#)、[length](#)、[base64decode](#) など、いくつかの Terraform 関数に既に気付いているかもしれません。<https://developer.hashicorp.com/terraform/language/functions/length> 組み込み関数 CloudFormation と同様に、Terraform には、設定で使用できる一連の [組み込み関数](#) があります。例えば、特定のリソース属性がファイルに直接貼り付けるのが非効率な非常に大きな JSON オブジェクトを取り込む場合、そのオブジェクトを json ファイルに入れ、Terraform 関数を使用してそれにアクセスできます。次の例では、file 関数はファイルの内容を文字列形式で返し、それを オブジェクトタイプ jsondecode に変換します。

```
resource "example_resource" "example_resource_name" {
  json_object = jsondecode(file("/path/to/file.json"))
}
```

表現

Terraform では、[条件式](#) も許可されます。条件式は、より従来の [3 項演算子](#) 構文を使用する点を除いて、関数に似 CloudFormation condition ています。次の例では、2 つの式はまったく同じ結果を返します。2 番目の例は、Terraform が [スプラット式](#) を呼び出すものです。アスタリスクにより、Terraform はリストをループし、各項目の id プロパティのみを使用して新しいリストを作成します。

```
resource "example_resource" "example_resource_name" {
  boolean_value = var.value ? true : false
  numeric_value = var.value > 0 ? 1 : 0
  string_value  = var.value == "change_me" ? "New value" : var.value
  string_value_2 = var.value != "change_me" ? var.value : "New value"
}
```

```
}  
There are two ways to express for loops in a Terraform configuration:  
resource "example_resource" "example_resource_name" {  
  list_value    = [for object in var.ids : object.id]  
  list_value_2 = var.ids[*].id  
}
```

メタ引数

前のコード例では、`list_value` および `list_value_2` は引数と呼ばれます。これらのメタ引数の一部については、すでに理解しているかもしれませんが、Terraform にはいくつかのメタ引数もあり、引数と同じように動作しますが、いくつかの追加機能があります。

- [depends_on](#) メタ引数は CloudFormation [DependsOn 属性](#) と非常によく似ています。
- [プロバイダー](#) メタ引数を使用すると、複数のプロバイダー設定を一度に使用できます。
- [ライフサイクル](#) メタ引数を使用すると、[の削除](#) および [削除](#) ポリシーと同様に、リソース設定をカスタマイズできます CloudFormation。

他のメタ引数を使用すると、関数と式の機能をリソースに直接追加できます。例えば、[カウント](#) メタ引数は、複数の類似リソースを同時に作成するための便利なメカニズムです。次の例は、メタ引数を使用せずに 2 つの Amazon Elastic Container Service (Amazon EKS) count クラスターを作成する方法を示しています。

```
resource "aws_eks_cluster" "example_0" {  
  name      = "example_0"  
  role_arn = aws_iam_role.cluster_role.arn  
  vpc_config {  
    endpoint_private_access = true  
    endpoint_public_access  = true  
    subnet_ids               = var.subnet_ids[0]  
  }  
}  
  
resource "aws_eks_cluster" "example_1" {  
  name      = "example_1"  
  role_arn = aws_iam_role.cluster_role.arn  
  vpc_config {  
    endpoint_private_access = true  
    endpoint_public_access  = true  
    subnet_ids               = var.subnet_ids[1]  
  }  
}
```

```
}  
}
```

次の例は、メタ引数を使用して count 2 つの Amazon EKS クラスターを作成する方法を示しています。

```
resource "aws_eks_cluster" "clusters" {  
  count    = 2  
  name     = "cluster_${count.index}"  
  role_arn = aws_iam_role.cluster_role.arn  
  vpc_config {  
    endpoint_private_access = true  
    endpoint_public_access  = true  
    subnet_ids               = var.subnet_ids[count.index]  
  }  
}
```

各にユニット名を付けるには、のリソースブロック内のリストインデックスにアクセスできません count.index。しかし、より複雑な類似リソースを複数作成する場合はどうなりますか？そこで [for_each](#) メタ引数が入ります。for_each メタ引数は、数値の代わりにリストまたはオブジェクトを渡すこと count を除いて、と非常によく似ています。Terraform は、リストまたはオブジェクトのメンバーごとに新しいリソースを作成します。これは、ループインデックスではなくリストの内容にアクセスできることを除いて count = length(list)、を設定する場合と似ています。

これは、項目のリストまたは単一のオブジェクトの両方で機能します。次の例では、ID id-1 として id-0 とを持つ 2 IDs つのリソースを作成します。

```
variable "ids" {  
  default = [  
    { id = "id-0" },  
    { id = "id-1" },  
  ]  
}  
  
resource "example_resource" "example_resource_name" {  
  # If your list fails, you might have to call "toset" on it to convert it to a set  
  for_each = toset(var.ids)  
  id       = each.value  
}
```

次の例では、Sparky の場合は poodle、Fluffy の場合は chihuahua の 2 つのリソースも作成します。

```

variable "dogs" {
  default = {
    poodle      = "Sparky"
    chihuahua   = "Fluffy"
  }
}

resource "example_resource" "example_resource_name" {
  for_each = var.dogs
  breed    = each.key
  name     = each.value
}

```

`count.index` を使用してカウント内のループインデックスにアクセスできるのと同様に、各オブジェクトを使用して `for_each` ループ内の各項目のキーと値にアクセスできます。`for_each` はリストとオブジェクトの両方を反復処理するため、各キーと値は少し混乱して追跡される可能性があります。次の表は、`for_each` メタ引数を使用するさまざまな方法と、反復ごとに値を参照する方法を示しています。

例	<code>for_each</code> タイプ	最初の反復	2 回目の反復
A	<pre>["poodle", "chihuahua"]</pre>	<pre>each.key = "poodle" each.value = null</pre>	<pre>each.key = "chihuahua" each.value = null</pre>
B	<pre>[{ type = "poodle", name = "Sparky" }, {</pre>	<pre>each.key = { type = "poodle", name = "Sparky" } each.value = null</pre>	<pre>each.key = { type = "chihuahua", name = "Fluffy" } each.value = null</pre>

例	for_each タイプ	最初の反復	2 回目の反復
	<pre>type = "chihuahua", name = "Fluffy" }]</pre>		
C	<pre>{ poodle = "Sparky", chihuahua = "Fluffy" }</pre>	<pre>each.key = "poodle" each.value = "Sparky"</pre>	<pre>each.key = "chihuahua" each.value = "Fluffy"</pre>

例	for_each タイプ	最初の反復	2 回目の反復
D	<pre>{ dogs = { poodle = "Sparky", chihuahua = "Fluffy" }, cats = { persian = "Felix", burmese = "Morris" } }</pre>	<pre>each.key = "dogs" each.value = { poodle = "Sparky", chihuahua = "Fluffy" }</pre>	<pre>each.key = "cats" each.value = { persian = "Felix", burmese = "Morris" }</pre>

例	for_each タイプ	最初の反復	2 回目の反復
E	<pre> { dogs = [{ type = "poodle", name = "Sparky" }, { type = "chihuahua", name = "Fluffy" }], cats = [{ type = "persian", name = "Felix" }, { type = "burmese", name = "Morris" }] } </pre>	<pre> each.key = "dogs" each.value = [{ type = "poodle", name = "Sparky" }, { type = "chihuahua", name = "Fluffy" }] </pre>	<pre> each.key = "cats" each.value = [{ type = "persian", name = "Felix" }, { type = "burmese", name = "Morris" }] </pre>

例	for_each タイプ	最初の反復	2 回目の反復
	<pre> }] } </pre>		

したがって、`var.animals` が行 E と等しい場合、次のコードを使用して動物ごとに 1 つのリソースを作成できます。

```

resource "example_resource" "example_resource_name" {
  for_each = var.animals
  type     = each.key
  breeds   = each.value[*].type
  names    = each.value[*].name
}

```

または、次のコードを使用して、動物ごとに 2 つのリソースを作成することもできます。

```

resource "example_resource" "example_resource_name" {
  for_each = var.animals.dogs
  type     = "dogs"
  breeds   = each.value.type
  names    = each.value.name
}

resource "example_resource" "example_resource_name" {
  for_each = var.animals.cats
  type     = "cats"
  breeds   = each.value.type
  names    = each.value.name
}

```

よくある質問

の代わりに Terraform をいつ使用すべきですか CloudFormationか？

一般に、ワークロードが主に をベースとしている場合 AWS、 は Terraform では一致しないネイティブサポートのレベル AWS CloudFormation を提供します。ただし、ワークロードにかなりのサードパーティプロセスが含まれている場合や、複数のクラウドプロバイダーに分散している場合、Terraform は考慮すべきツールです。

AWS CDK の代わりに をいつ使用すべきですか CloudFormation？

を使用する場合 AWS Cloud Development Kit (AWS CDK)、 も使用します CloudFormation。AWS CDK では、一般的なプログラミング言語を使用して CloudFormation テンプレートを生成できます。がサポートするプログラミング言語のいずれかの経験がある場合、AWS CDK [AWS CDK <https://docs.aws.amazon.com/cdk/v2/guide/languages.html>](https://docs.aws.amazon.com/cdk/v2/guide/languages.html)は CloudFormation テンプレートの生成に必要な時間を短縮できます。

Terraform 設定 AWS CDK を生成する のようなツールはありますか？

と比較すると AWS CDK、[CDK for Terraform \(CDKTF\)](#) は同じコンストラクティブライブラリを使用してリソースをプロビジョニングし、同じ [jsii](#) エンジンを使用して複数のプログラミング言語をサポートします。これを使用して、が CloudFormation テンプレートを生成するのと同じ方法で Terraform 設定 AWS CDK を生成できます。

Terraform の詳細については、どうすればよいですか？

Terraform の高度な概念の詳細については、「[Terraform ドキュメント](#)」を参照してください。また、すべての主要なプロバイダーとオープンソースモジュールのコンポーネントについても説明します。

関連リソース

AWS ドキュメント

- [「AWS CDK ドキュメント」](#)
- [「AWS CloudFormation ドキュメント」](#)
- [Terraform: Beyond the Basics with AWS](#) (AWS ブログ記事)

その他のリソース

- [CDK for Terraform のドキュメント](#)
- Terraformのドキュメント

付録: Terraform 属性アクセスの例

リソース

```
resource "aws_s3_bucket" "myS3Bucket" {  
    bucket = "my-s3-bucket"  
}  
  
bucketName = aws_s3_bucket.myS3Bucket.bucket
```

データソース

```
data "aws_s3_bucket" "myS3Bucket" {  
    bucket = "my-s3-bucket"  
}  
  
bucketName = data.aws_s3_bucket.myS3Bucket.bucket
```

モジュール

```
module "eks" {  
    source = "terraform-aws-modules/eks/aws"  
    version = "20.2.1"  
}  
  
vpc_id = module.eks.vpc_id
```

変数

```
variable "my_variable" = {  
    default = "dog"  
}  
  
animalType = var.my_variable
```

ローカル

```
locals {  
  type = "dog"  
}  
  
animalType = local.type
```


ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、[RSS フィード](#) をサブスクライブできます。

変更	説明	日付
初版発行	—	2024 年 3 月 29 日

AWS 規範的ガイドランスの用語集

以下は、AWS 規範的ガイドランスが提供する戦略、ガイド、パターンで一般的に使用される用語です。エントリを提案するには、用語集の最後のフィードバックの提供リンクを使用します。

数字

7 Rs

アプリケーションをクラウドに移行するための 7 つの一般的な移行戦略。これらの戦略は、ガートナーが 2011 年に特定した 5 Rs に基づいて構築され、以下で構成されています。

- リファクタリング/アーキテクチャの再設計 — クラウドネイティブ特徴を最大限に活用して、俊敏性、パフォーマンス、スケーラビリティを向上させ、アプリケーションを移動させ、アーキテクチャを変更します。これには、通常、オペレーティングシステムとデータベースの移植が含まれます。例: オンプレミスの Oracle データベースを Amazon Aurora PostgreSQL 互換エディションに移行します。
- リプラットフォーム (リフトアンドリシェイプ) — アプリケーションをクラウドに移行し、クラウド機能を活用するための最適化レベルを導入します。例: オンプレミスの Oracle データベースをの Oracle 用 Amazon Relational Database Service (Amazon RDS) に移行します AWS クラウド。
- 再購入 (ドロップアンドショップ) — 通常、従来のライセンスから SaaS モデルに移行して、別の製品に切り替えます。例: カスタマーリレーションシップ管理 (CRM) システムを Salesforce.com に移行します。
- リホスト (リフトアンドシフト) — クラウド機能を活用するための変更を加えずに、アプリケーションをクラウドに移行します。例: オンプレミスの Oracle データベースをの EC2 インスタンス上の Oracle に移行します AWS クラウド。
- 再配置 (ハイパーバイザーレベルのリフトアンドシフト) — 新しいハードウェアを購入したり、アプリケーションを書き換えたり、既存の運用を変更したりすることなく、インフラストラクチャをクラウドに移行できます。サーバーをオンプレミスプラットフォームから同じプラットフォームのクラウドサービスに移行します。例: Microsoft Hyper-Vアプリケーションをに移行します AWS。
- 保持 (再アクセス) — アプリケーションをお客様のソース環境で保持します。これには、主要なリファクタリングを必要とするアプリケーションや、お客様がその作業を後日まで延期したいアプリケーション、およびそれら移行するためのビジネス上の正当性がないため、お客様が保持するレガシーアプリケーションなどがあります。

- 使用停止 — お客様のソース環境で不要になったアプリケーションを停止または削除します。

A

ABAC

[「属性ベースのアクセスコントロール」](#)を参照してください。

抽象化されたサービス

[「マネージドサービス」](#)を参照してください。

ACID

[「原子性、一貫性、分離性、耐久性」](#)を参照してください。

アクティブ - アクティブ移行

(双方向レプリケーションツールまたは二重書き込み操作を使用して) ソースデータベースとターゲットデータベースを同期させ、移行中に両方のデータベースが接続アプリケーションからのトランザクションを処理するデータベース移行方法。この方法では、1 回限りのカットオーバーの必要がなく、管理された小規模なバッチで移行できます。アクティブ/[パッシブ移行](#)よりも柔軟ですが、より多くの作業が必要です。

アクティブ - パッシブ移行

ソースデータベースとターゲットデータベースを同期させながら、データがターゲットデータベースにレプリケートされている間、接続しているアプリケーションからのトランザクションをソースデータベースのみで処理するデータベース移行の方法。移行中、ターゲットデータベースはトランザクションを受け付けません。

集計関数

行のグループを操作し、グループの単一の戻り値を計算する SQL 関数。集計関数の例としては、SUMや などがあありますMAX。

AI

[「人工知能」](#)を参照してください。

AIOps

[「人工知能オペレーション」](#)を参照してください。

匿名化

データセット内の個人情報を完全に削除するプロセス。匿名化は個人のプライバシー保護に役立ちます。匿名化されたデータは、もはや個人データとは見なされません。

アンチパターン

繰り返し起こる問題に対して頻繁に用いられる解決策で、その解決策が逆効果であったり、効果がなかったり、代替案よりも効果が低かったりするもの。

アプリケーションコントロール

マルウェアからシステムを保護するために、承認されたアプリケーションのみを使用できるようにするセキュリティアプローチ。

アプリケーションポートフォリオ

アプリケーションの構築と維持にかかるコスト、およびそのビジネス価値を含む、組織が使用する各アプリケーションに関する詳細情報の集まり。この情報は、[ポートフォリオの検出と分析プロセス](#) の需要要素であり、移行、モダナイズ、最適化するアプリケーションを特定し、優先順位を付けるのに役立ちます。

人工知能 (AI)

コンピューティングテクノロジーを使用し、学習、問題の解決、パターンの認識など、通常は人間に関連づけられる認知機能の実行に特化したコンピュータサイエンスの分野。詳細については、「[人工知能 \(AI\) とは何ですか?](#)」を参照してください。

AI オペレーション (AIOps)

機械学習技術を使用して運用上の問題を解決し、運用上のインシデントと人の介入を減らし、サービス品質を向上させるプロセス。AWS 移行戦略での AIOps の使用方法については、[オペレーション統合ガイド](#) を参照してください。

非対称暗号化

暗号化用のパブリックキーと復号用のプライベートキーから成る 1 組のキーを使用した、暗号化のアルゴリズム。パブリックキーは復号には使用されないため共有しても問題ありませんが、プライベートキーの利用は厳しく制限する必要があります。

原子性、一貫性、分離性、耐久性 (ACID)

エラー、停電、その他の問題が発生した場合でも、データベースのデータ有効性と運用上の信頼性を保証する一連のソフトウェアプロパティ。

属性ベースのアクセス制御 (ABAC)

部署、役職、チーム名など、ユーザーの属性に基づいてアクセス許可をきめ細かく設定する方法。詳細については、AWS Identity and Access Management (IAM) ドキュメントの「[の ABAC AWS](#)」を参照してください。

信頼できるデータソース

最も信頼性のある情報源とされるデータのプライマリーバージョンを保存する場所。匿名化、編集、仮名化など、データを処理または変更する目的で、信頼できるデータソースから他の場所にデータをコピーすることができます。

アベイラビリティゾーン

他のアベイラビリティゾーンの障害から AWS リージョン 隔離され、同じリージョン内の他のアベイラビリティゾーンへの低コストで低レイテンシーのネットワーク接続を提供する 内の別の場所。

AWS クラウド導入フレームワーク (AWS CAF)

組織がクラウドに正常に移行 AWS するための効率的で効果的な計画を立てるのに役立つ、のガイドラインとベストプラクティスのフレームワークです。AWS CAF は、ビジネス、人材、ガバナンス、プラットフォーム、セキュリティ、運用という 6 つの重点分野にガイドランスを編成します。ビジネス、人材、ガバナンスの観点では、ビジネススキルとプロセスに重点を置き、プラットフォーム、セキュリティ、オペレーションの視点は技術的なスキルとプロセスに焦点を当てています。例えば、人材の観点では、人事 (HR)、人材派遣機能、および人材管理を扱うステークホルダーを対象としています。この観点から、AWS CAF は、組織がクラウド導入を成功させるための準備に役立つ、人材開発、トレーニング、コミュニケーションに関するガイドランスを提供します。詳細については、[AWS CAF ウェブサイト](#) と [AWS CAF のホワイトペーパー](#) を参照してください。

AWS ワークロード認定フレームワーク (AWS WQF)

データベース移行ワークロードを評価し、移行戦略を推奨し、作業見積もりを提供するツール。AWS WQF は AWS Schema Conversion Tool (AWS SCT) に含まれています。データベーススキーマとコードオブジェクト、アプリケーションコード、依存関係、およびパフォーマンス特性を分析し、評価レポートを提供します。

B

不正なボット

個人や組織に混乱や損害を与えることを目的とした[ボット](#)。

BCP

[「事業継続計画」](#)を参照してください。

動作グラフ

リソースの動作とインタラクションを経時的に示した、一元的なインタラクティブビュー。Amazon Detective の動作グラフを使用すると、失敗したログオンの試行、不審な API 呼び出し、その他同様のアクションを調べることができます。詳細については、Detective ドキュメントの[Data in a behavior graph](#)を参照してください。

ビッグエンディアンシステム

最上位バイトを最初に格納するシステム。[エンディアンネス](#)も参照してください。

二項分類

バイナリ結果 (2 つの可能なクラスのうちの一つ) を予測するプロセス。例えば、お客様の機械学習モデルで「この E メールはスパムですか、それともスパムではありませんか」などの問題を予測する必要があるかもしれません。または「この製品は書籍ですか、車ですか」などの問題を予測する必要があるかもしれません。

ブルームフィルター

要素がセットのメンバーであるかどうかをテストするために使用される、確率的でメモリ効率の高いデータ構造。

ブルー/グリーンデプロイ

2 つの異なる同一の環境を作成するデプロイ戦略。現在のアプリケーションバージョンは 1 つの環境 (青) で実行し、新しいアプリケーションバージョンは他の環境 (緑) で実行します。この戦略は、最小限の影響で迅速にロールバックするのに役立ちます。

ボット

インターネット経由で自動タスクを実行し、人間のアクティビティやインタラクションをシミュレートするソフトウェアアプリケーション。インターネット上の情報のインデックスを作成するウェブクローラーなど、一部のボットは有用または有益です。悪質なボットと呼ばれる他のボット

トの中には、個人や組織に混乱を与えたり、損害を与えたりすることを意図しているものがあります。

ポットネット

[マルウェア](#)に感染し、[ポット](#)のヘルダーまたはポットオペレーターと呼ばれる、単一関係者の管理下にあるポットのネットワーク。ポットは、ポットとその影響をスケールするための最もよく知られているメカニズムです。

ブランチ

コードリポジトリに含まれる領域。リポジトリに最初に作成するブランチは、メインブランチといます。既存のブランチから新しいブランチを作成し、その新しいブランチで機能を開発したり、バグを修正したりできます。機能を構築するために作成するブランチは、通常、機能ブランチと呼ばれます。機能をリリースする準備ができたなら、機能ブランチをメインブランチに統合します。詳細については、[「ブランチについて」](#) (GitHub ドキュメント) を参照してください。

ブレイクグラスアクセス

例外的な状況や承認されたプロセスを通じて、ユーザーが通常アクセス許可を持たない AWS アカウント にすばやくアクセスできるようにします。詳細については、Well-Architected [ガイド](#) の「[ブレイクグラス手順の実装](#)」インジケータ AWS を参照してください。

ブラウフィールド戦略

環境の既存インフラストラクチャ。システムアーキテクチャにブラウフィールド戦略を導入する場合、現在のシステムとインフラストラクチャの制約に基づいてアーキテクチャを設計します。既存のインフラストラクチャを拡張している場合は、ブラウフィールド戦略と[グリーンフィールド](#)戦略を融合させることもできます。

バッファキャッシュ

アクセス頻度が最も高いデータが保存されるメモリ領域。

ビジネス能力

価値を生み出すためにビジネスが行うこと (営業、カスタマーサービス、マーケティングなど)。マイクロサービスのアーキテクチャと開発の決定は、ビジネス能力によって推進できます。詳細については、ホワイトペーパー [AWSでのコンテナ化されたマイクロサービスの実行](#) の [ビジネス機能を中心に組織化](#) セクションを参照してください。

ビジネス継続性計画 (BCP)

大規模移行など、中断を伴うイベントが運用に与える潜在的な影響に対処し、ビジネスを迅速に再開できるようにする計画。

C

CAF

[AWS 「クラウド導入フレームワーク」を参照してください。](#)

Canary デプロイ

エンドユーザーへのバージョンの低速かつ増分的なリリース。確信できたら、新しいバージョンをデプロイし、現在のバージョン全体を置き換えます。

CCoE

[「Cloud Center of Excellence」を参照してください。](#)

CDC

[「データキャプチャの変更」を参照してください。](#)

変更データキャプチャ (CDC)

データソース (データベーステーブルなど) の変更を追跡し、その変更に関するメタデータを記録するプロセス。CDC は、ターゲットシステムでの変更を監査またはレプリケートして同期を維持するなど、さまざまな目的に使用できます。

カオスエンジニアリング

障害や破壊的なイベントを意図的に導入して、システムの耐障害性をテストします。[AWS Fault Injection Service \(AWS FIS \)](#) を使用して、AWS ワークロードに負荷をかけ、その応答を評価する実験を実行できます。

CI/CD

[「継続的インテグレーションと継続的デリバリー」を参照してください。](#)

分類

予測を生成するのに役立つ分類プロセス。分類問題の機械学習モデルは、離散値を予測します。離散値は、常に互いに区別されます。例えば、モデルがイメージ内に車があるかどうかを評価する必要がある場合があります。

クライアント側の暗号化

ターゲットがデータ AWS のサービスを受信する前に、ローカルでデータを暗号化します。

Cloud Center of Excellence (CCoE)

クラウドのベストプラクティスの作成、リソースの移動、移行のタイムラインの確立、大規模変革を通じて組織をリードするなど、組織全体のクラウド導入の取り組みを推進する学際的なチーム。詳細については、AWS クラウド エンタープライズ戦略ブログの[CCoE の投稿](#)を参照してください。

クラウドコンピューティング

リモートデータストレージと IoT デバイス管理に通常使用されるクラウドテクノロジー。クラウドコンピューティングは、一般的に[エッジコンピューティング](#)テクノロジーに接続されています。

クラウド運用モデル

IT 組織において、1 つ以上のクラウド環境を構築、成熟、最適化するために使用される運用モデル。詳細については、[「クラウド運用モデルの構築」](#)を参照してください。

導入のクラウドステージ

組織が移行するときに通常実行する 4 つのフェーズ AWS クラウド :

- プロジェクト — 概念実証と学習を目的として、クラウド関連のプロジェクトをいくつか実行する
- 基礎固め — お客様のクラウドの導入を拡大するための基礎的な投資 (ランディングゾーンの作成、CCoE の定義、運用モデルの確立など)
- 移行 — 個々のアプリケーションの移行
- 再発明 — 製品とサービスの最適化、クラウドでのイノベーション

これらのステージは、AWS クラウド エンタープライズ戦略ブログのブログ記事[「クラウドファーストへのジャーニー」](#)と[「導入のステージ」](#)で Stephen Orban によって定義されました。移行戦略とどのように関連しているかについては、AWS [「移行準備ガイド」](#)を参照してください。

CMDB

[「設定管理データベース」](#)を参照してください。

コードリポジトリ

ソースコードやその他の資産 (ドキュメント、サンプル、スクリプトなど) が保存され、バージョン管理プロセスを通じて更新される場所。一般的なクラウドリポジトリには、GitHub またはが含まれます AWS CodeCommit。コードの各バージョンはブランチと呼ばれます。マイクロサー

ビスの構造では、各リポジトリは 1 つの機能専用です。1 つの CI/CD パイプラインで複数のリポジトリを使用できます。

コールドキャッシュ

空である、または、かなり空きがある、もしくは、古いデータや無関係なデータが含まれているバッファキャッシュ。データベースインスタンスはメインメモリまたはディスクから読み取る必要があります。バッファキャッシュから読み取るよりも時間がかかるため、パフォーマンスに影響します。

コールドデータ

めったにアクセスされず、通常は過去のデータです。この種類のデータをクエリする場合、通常は低速なクエリでも問題ありません。このデータを低パフォーマンスで安価なストレージ階層またはクラスに移動すると、コストを削減することができます。

コンピュータビジョン (CV)

機械学習を使用してデジタルイメージやビデオなどのビジュアル形式から情報を分析および抽出する [AI](#) の分野。例えば、はオンプレミスカメラネットワークに CV を追加するデバイス AWS Panorama を提供し、Amazon SageMaker は CV の画像処理アルゴリズムを提供します。

設定ドリフト

ワークロードの場合、設定は想定した状態から変化します。これにより、ワークロードが非準拠になる可能性があり、通常は段階的かつ意図的ではありません。

構成管理データベース (CMDB)

データベースとその IT 環境 (ハードウェアとソフトウェアの両方のコンポーネントとその設定を含む) に関する情報を保存、管理するリポジトリ。通常、CMDB のデータは、移行のポートフォリオの検出と分析の段階で使用します。

コンフォーマンスパック

コンプライアンスチェックとセキュリティチェックをカスタマイズするためにアセンブルできる AWS Config ルールと修復アクションのコレクション。YAML テンプレートを使用して、コンフォーマンスパックを AWS アカウント およびリージョンの単一のエンティティとしてデプロイすることも、組織全体にデプロイすることもできます。詳細については、AWS Config ドキュメントの「[コンフォーマンスパック](#)」を参照してください。

継続的インテグレーションと継続的デリバリー (CI/CD)

ソフトウェアリリースプロセスのソース、ビルド、テスト、ステージング、本番の各ステージを自動化するプロセス。CI/CD は一般的にパイプラインと呼ばれます。プロセスの自動化、生産性

の向上、コード品質の向上、配信の加速化を可能にします。詳細については、「[継続的デリバリーの利点](#)」を参照してください。CD は継続的デプロイ (Continuous Deployment) の略語でもあります。詳細については「[継続的デリバリーと継続的なデプロイ](#)」を参照してください。

CV

[「コンピュータビジョン」](#)を参照してください。

D

保管中のデータ

ストレージ内にあるデータなど、常に自社のネットワーク内にあるデータ。

データ分類

ネットワーク内のデータを重要度と機密性に基づいて識別、分類するプロセス。データに適した保護および保持のコントロールを判断する際に役立つため、あらゆるサイバーセキュリティのリスク管理戦略において重要な要素です。データ分類は、AWS Well-Architected フレームワークのセキュリティの柱のコンポーネントです。詳細については、[データ分類](#)を参照してください。

データドリフト

実稼働データと ML モデルのトレーニングに使用されたデータとの間に有意な差異が生じたり、入力データが時間の経過と共に有意に変化したりすることです。データドリフトは、ML モデル予測の全体的な品質、精度、公平性を低下させる可能性があります。

転送中のデータ

ネットワーク内 (ネットワークリソース間など) を活発に移動するデータ。

データメッシュ

一元化された管理とガバナンスにより、分散型の分散型データ所有権を提供するアーキテクチャフレームワーク。

データ最小化

厳密に必要なデータのみを収集し、処理するという原則。データ最小化を実践 AWS クラウドすることで、プライバシーリスク、コスト、分析のカーボンフットプリントを削減できます。

データ境界

AWS 環境内の一連の予防ガードレール。信頼できる ID のみが、期待されるネットワークから信頼できるリソースにアクセスしていることを確認できます。詳細については、[「でのデータ境界の構築 AWS」](#)を参照してください。

データの前処理

raw データをお客様の機械学習モデルで簡単に解析できる形式に変換すること。データの前処理とは、特定の列または行を削除して、欠落している、矛盾している、または重複する値に対処することを意味します。

データ出所

データの生成、送信、保存の方法など、データのライフサイクル全体を通じてデータの出所と履歴を追跡するプロセス。

データ件名

データを収集、処理している個人。

データウェアハウス

分析などのビジネスインテリジェンスをサポートするデータ管理システム。データウェアハウスには通常、大量の履歴データが含まれており、クエリや分析によく使用されます。

データベース定義言語 (DDL)

データベース内のテーブルやオブジェクトの構造を作成または変更するためのステートメントまたはコマンド。

データベース操作言語 (DML)

データベース内の情報を変更 (挿入、更新、削除) するためのステートメントまたはコマンド。

DDL

[「データベース定義言語」](#)を参照してください。

ディープアンサンブル

予測のために複数の深層学習モデルを組み合わせる。ディープアンサンブルを使用して、より正確な予測を取得したり、予測の不確実性を推定したりできます。

ディープラーニング

人工ニューラルネットワークの複数層を使用して、入力データと対象のターゲット変数の間のマッピングを識別する機械学習サブフィールド。

defense-in-depth

一連のセキュリティメカニズムとコントロールをコンピュータネットワーク全体に層状に重ねて、ネットワークとその内部にあるデータの機密性、整合性、可用性を保護する情報セキュリティの手法。この戦略をに採用するときは AWS、AWS Organizations 構造の異なるレイヤーに複数のコントロールを追加して、リソースの安全性を確保します。例えば、defense-in-depth アプローチでは、多要素認証、ネットワークセグメンテーション、暗号化を組み合わせることができます。

委任管理者

では AWS Organizations、互換性のあるサービスが AWS メンバーアカウントを登録して組織のアカウントを管理し、そのサービスのアクセス許可を管理できます。このアカウントを、そのサービスの委任管理者と呼びます。詳細、および互換性のあるサービスの一覧は、AWS Organizations ドキュメントの[AWS Organizationsで利用できるサービス](#)を参照してください。

デプロイメント

アプリケーション、新機能、コードの修正をターゲットの環境で利用できるようにするプロセス。デプロイでは、コードベースに変更を施した後、アプリケーションの環境でそのコードベースを構築して実行します。

開発環境

[「環境」](#)を参照してください。

検出管理

イベントが発生したときに、検出、ログ記録、警告を行うように設計されたセキュリティコントロール。これらのコントロールは副次的な防衛手段であり、実行中の予防的コントロールをすり抜けたセキュリティイベントをユーザーに警告します。詳細については、Implementing security controls on AWSの[Detective controls](#)を参照してください。

開発バリューストリームマッピング (DVSM)

ソフトウェア開発ライフサイクルのスピードと品質に悪影響を及ぼす制約を特定し、優先順位を付けるために使用されるプロセス。DVSM は、もともとリーンマニユファクチャリング・プラクティスのために設計されたバリューストリームマッピング・プロセスを拡張したものです。ソフトウェア開発プロセスを通じて価値を創造し、動かすために必要なステップとチームに焦点を当てています。

デジタルツイン

建物、工場、産業機器、生産ラインなど、現実世界のシステムを仮想的に表現したものです。デジタルツインは、予知保全、リモートモニタリング、生産最適化をサポートします。

ディメンションテーブル

[スタースキーマ](#) では、ファクトテーブル内の量的データに関するデータ属性を含む小さなテーブル。ディメンションテーブル属性は通常、テキストフィールドまたはテキストのように動作する離散数値です。これらの属性は、クエリの制約、フィルタリング、結果セットのラベル付けに一般的に使用されます。

ディザスタ

ワークロードまたはシステムが、導入されている主要な場所でのビジネス目標の達成を妨げるイベント。これらのイベントは、自然災害、技術的障害、または意図しない設定ミスやマルウェア攻撃などの人間の行動の結果である場合があります。

ディザスタリカバリ (DR)

[災害によるダウンタイムとデータ損失を最小限に抑えるために使用する戦略とプロセス](#)。詳細については、AWS Well-Architected [フレームワークの「でのワークロードのディザスタリカバリ」](#) [AWS: クラウドでのリカバリ](#) を参照してください。

DML

[「データベース操作言語」](#) を参照してください。

ドメイン駆動型設計

各コンポーネントが提供している変化を続けるドメイン、またはコアビジネス目標にコンポーネントを接続して、複雑なソフトウェアシステムを開発するアプローチ。この概念は、エリック・エヴァンスの著書、Domain-Driven Design: Tackling Complexity in the Heart of Software (ドメイン駆動設計: ソフトウェアの中心における複雑さへの取り組み) で紹介されています (ボストン: Addison-Wesley Professional, 2003)。strangler fig パターンでドメイン駆動型設計を使用する方法の詳細については、[コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET \(ASMX\) ウェブサービスを段階的にモダナイズ](#) を参照してください。

DR

[「ディザスタリカバリ」](#) を参照してください。

ドリフト検出

ベースライン設定からの偏差の追跡。例えば、AWS CloudFormation を使用して [システムリソースのドリフトを検出したり](#)、を使用して AWS Control Tower ガバナンス要件への準拠に影響を与える可能性のある [ランディングゾーンの変更を検出したり](#) できます。

DVSM

[「開発値ストリームマッピング」](#) を参照してください。

E

EDA

[「探索的データ分析」](#)を参照してください。

エッジコンピューティング

IoT ネットワークのエッジにあるスマートデバイスの計算能力を高めるテクノロジー。[クラウドコンピューティング](#)と比較すると、エッジコンピューティングは通信レイテンシーを短縮し、応答時間を短縮できます。

暗号化

人間が読み取り可能なプレーンテキストデータを暗号文に変換するコンピューティングプロセス。

暗号化キー

暗号化アルゴリズムが生成した、ランダム化されたビットからなる暗号文字列。キーの長さは決まっておらず、各キーは予測できないように、一意になるように設計されています。

エンディアン

コンピュータメモリにバイトが格納される順序。ビッグエンディアンシステムでは、最上位バイトが最初に格納されます。リトルエンディアンシステムでは、最下位バイトが最初に格納されます。

エンドポイント

[「サービスエンドポイント」](#)を参照してください。

エンドポイントサービス

仮想プライベートクラウド (VPC) 内でホストして、他のユーザーと共有できるサービス。を使用してエンドポイントサービスを作成し AWS PrivateLink、他の AWS アカウント または AWS Identity and Access Management (IAM) プリンシパルにアクセス許可を付与できます。これらのアカウントまたはプリンシパルは、インターフェイス VPC エンドポイントを作成することで、エンドポイントサービスにプライベートに接続できます。詳細については、Amazon Virtual Private Cloud (Amazon VPC) ドキュメントの「[エンドポイントサービスを作成する](#)」を参照してください。

エンタープライズリソースプランニング (ERP)

エンタープライズの主要なビジネスプロセス (アカウンティング、[MES](#)、プロジェクト管理など) を自動化および管理するシステム。

エンベロープ暗号化

暗号化キーを、別の暗号化キーを使用して暗号化するプロセス。詳細については、AWS Key Management Service (AWS KMS) [ドキュメントの「エンベロープ暗号化」](#)を参照してください。

環境

実行中のアプリケーションのインスタンス。クラウドコンピューティングにおける一般的な環境の種類は以下のとおりです。

- 開発環境 — アプリケーションのメンテナンスを担当するコアチームのみが使用できる、実行中のアプリケーションのインスタンス。開発環境は、上位の環境に昇格させる変更をテストするときに使用します。このタイプの環境は、テスト環境と呼ばれることもあります。
- 下位環境 — 初期ビルドやテストに使用される環境など、アプリケーションのすべての開発環境。
- 本番環境 — エンドユーザーがアクセスできる、実行中のアプリケーションのインスタンス。CI/CD パイプラインでは、本番環境が最後のデプロイ環境になります。
- 上位環境 — コア開発チーム以外のユーザーがアクセスできるすべての環境。これには、本番環境、本番前環境、ユーザー承認テスト環境などが含まれます。

エピック

アジャイル方法論で、お客様の作業の整理と優先順位付けに役立つ機能カテゴリ。エピックでは、要件と実装タスクの概要についてハイレベルな説明を提供します。例えば、AWS CAF セキュリティエピックには、ID とアクセスの管理、検出コントロール、インフラストラクチャセキュリティ、データ保護、インシデント対応が含まれます。AWS 移行戦略のエピックの詳細については、[プログラム実装ガイド](#)を参照してください。

ERP

[「エンタープライズリソース計画」](#)を参照してください。

探索的データ分析 (EDA)

データセットを分析してその主な特性を理解するプロセス。お客様は、データを収集または集計してから、パターンの検出、異常の検出、および前提条件のチェックのための初期調査を実行します。EDA は、統計の概要を計算し、データの可視化を作成することによって実行されます。

F

ファクトテーブル

[スタースキーマ](#) の中央テーブル。事業運営に関する定量的データを保存します。通常、ファクトテーブルには、メジャーを含む列とディメンションテーブルへの外部キーを含む列の 2 種類の列が含まれます。

フェイルファスト

頻繁で段階的なテストを使用して開発ライフサイクルを短縮する哲学。これはアジャイルアプローチの重要な部分です。

障害分離境界

では AWS クラウド、障害の影響を制限し AWS リージョン、ワークロードの耐障害性を向上させるアベイラビリティゾーン、コントロールプレーン、データプレーンなどの境界です。詳細については、[AWS 「障害分離境界」](#) を参照してください。

機能ブランチ

[「ブランチ」](#) を参照してください。

特徴量

お客様が予測に使用する入力データ。例えば、製造コンテキストでは、特徴量は製造ラインから定期的にキャプチャされるイメージの可能性もあります。

特徴量重要度

モデルの予測に対する特徴量の重要性。これは通常、Shapley Additive Deskonations (SHAP) や積分勾配など、さまざまな手法で計算できる数値スコアで表されます。詳細については、[「を使用した機械学習モデルの解釈可能性 : AWS」](#) を参照してください。

機能変換

追加のソースによるデータのエンリッチ化、値のスケーリング、単一のデータフィールドからの複数の情報セットの抽出など、機械学習プロセスのデータを最適化すること。これにより、機械学習モデルはデータの恩恵を受けることができます。例えば、「2021-05-27 00:15:37」の日付を「2021 年」、「5 月」、「木」、「15」に分解すると、学習アルゴリズムがさまざまなデータコンポーネントに関連する微妙に異なるパターンを学習するのに役立ちます。

FGAC

[「きめ細かなアクセスコントロール」](#) を参照してください。

きめ細かなアクセス制御 (FGAC)

複数の条件を使用してアクセス要求を許可または拒否すること。

フラッシュカット移行

段階的なアプローチを使用するのではなく、[変更データキャプチャ](#)による継続的なデータレプリケーションを使用して、可能な限り短時間でデータを移行するデータベース移行方法。目的はダウンタイムを最小限に抑えることです。

G

ジオブロッキング

[「地理的制限」](#)を参照してください。

地理的制限 (ジオブロッキング)

Amazon では CloudFront、特定の国のユーザーがコンテンツディストリビューションにアクセスできないようにするオプションです。アクセスを許可する国と禁止する国は、許可リストまたは禁止リストを使って指定します。詳細については、CloudFront ドキュメントの[「コンテンツの地理的ディストリビューションの制限」](#)を参照してください。

Gitflow ワークフロー

下位環境と上位環境が、ソースコードリポジトリでそれぞれ異なるブランチを使用する方法。Gitflow ワークフローはレガシーと見なされ、[トランクベースのワークフロー](#)はモダンで推奨されるアプローチです。

グリーンフィールド戦略

新しい環境に既存のインフラストラクチャが存在しないこと。システムアーキテクチャにグリーンフィールド戦略を導入する場合、既存のインフラストラクチャ (別名[ブラウンフィールド](#)) との互換性の制約を受けることなく、あらゆる新しいテクノロジーを選択できます。既存のインフラストラクチャを拡張している場合は、ブラウンフィールド戦略とグリーンフィールド戦略を融合させることもできます。

ガードレール

組織単位 (OU) 全般のリソース、ポリシー、コンプライアンスを管理するのに役立つ概略的なルール。予防ガードレールは、コンプライアンス基準に一致するようにポリシーを実施します。これらは、サービスコントロールポリシーと IAM アクセス許可の境界を使用して実装

されます。検出ガードレールは、ポリシー違反やコンプライアンス上の問題を検出し、修復のためのアラートを発信します。これらは、AWS Config、Amazon AWS Security Hub、GuardDuty、Amazon Inspector AWS Trusted Advisor、およびカスタム AWS Lambda チェックを使用して実装されます。

H

HA

[「高可用性」](#)を参照してください。

異種混在データベースの移行

別のデータベースエンジンを使用するターゲットデータベースへお客様の出典データベースの移行 (例えば、Oracle から Amazon Aurora)。異種間移行は通常、アーキテクチャの再設計作業の一部であり、スキーマの変換は複雑なタスクになる可能性があります。[AWS は、スキーマの変換に役立つ AWS SCTを提供します。](#)

ハイアベイラビリティ (HA)

課題や災害が発生した場合に、介入なしにワークロードを継続的に運用できること。HA システムは、自動的にフェイルオーバーし、一貫して高品質のパフォーマンスを提供し、パフォーマンスへの影響を最小限に抑えながらさまざまな負荷や障害を処理するように設計されています。

ヒストリアンのモダナイゼーション

製造業のニーズによりよく応えるために、オペレーションテクノロジー (OT) システムをモダナイズし、アップグレードするためのアプローチ。ヒストリアンは、工場内のさまざまなソースからデータを収集して保存するために使用されるデータベースの一種です。

同種データベースの移行

お客様の出典データベースを、同じデータベースエンジンを共有するターゲットデータベース (Microsoft SQL Server から Amazon RDS for SQL Server など) に移行する。同種間移行は、通常、リホストまたはリプラットフォーム化の作業の一部です。ネイティブデータベースユーティリティを使用して、スキーマを移行できます。

ホットデータ

リアルタイムデータや最近の翻訳データなど、頻繁にアクセスされるデータ。通常、このデータには高速なクエリ応答を提供する高性能なストレージ階層またはクラスが必要です。

ホットフィックス

本番環境の重大な問題を修正するために緊急で配布されるプログラム。緊急性のため、通常、修正は一般的な DevOps リリースワークフローの外で行われます。

ハイパーケア期間

カットオーバー直後、移行したアプリケーションを移行チームがクラウドで管理、監視して問題に対処する期間。通常、この期間は 1~4 日です。ハイパーケア期間が終了すると、アプリケーションに対する責任は一般的に移行チームからクラウドオペレーションチームに移ります。

I

IaC

[「Infrastructure as Code」](#) を参照してください。

ID ベースのポリシー

AWS クラウド 環境内のアクセス許可を定義する 1 つ以上の IAM プリンシパルにアタッチされたポリシー。

アイドル状態のアプリケーション

90 日間の平均的な CPU およびメモリ使用率が 5~20% のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するか、オンプレミスに保持するのが一般的です。

IIoT

[「産業モノのインターネット」](#) を参照してください。

イミュータブルインフラストラクチャ

既存のインフラストラクチャを更新、パッチ適用、または変更する代わりに、本番ワークロード用の新しいインフラストラクチャをデプロイするモデル。イミュータブルなインフラストラクチャは、[本質的にミュータブルなインフラストラクチャ](#) よりも一貫性、信頼性、予測性が高くなります。詳細については、AWS Well-Architected フレームワークの[「変更不可能なインフラストラクチャを使用したデプロイ」](#) のベストプラクティスを参照してください。

インバウンド (受信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーションの外部からネットワーク接続を受け入れ、検査し、ルーティングする VPC。[AWS Security Reference Architecture](#) では、アプリ

ケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

増分移行

アプリケーションを 1 回ですべてカットオーバーするのではなく、小さい要素に分けて移行するカットオーバー戦略。例えば、最初は少数のマイクロサービスまたはユーザーのみを新しいシステムに移行する場合があります。すべてが正常に機能することを確認できたら、残りのマイクロサービスやユーザーを段階的に移行し、レガシーシステムを廃止できるようにします。この戦略により、大規模な移行に伴うリスクが軽減されます。

インダストリー 4.0

接続、リアルタイムデータ、自動化、分析、AI/ML の進歩を通じて、のビジネスプロセスのモダナイゼーションを指すために 2016 年に [Klaus Schwab](#) によって導入された用語。

インフラストラクチャ

アプリケーションの環境に含まれるすべてのリソースとアセット。

Infrastructure as Code (IaC)

アプリケーションのインフラストラクチャを一連の設定ファイルを使用してプロビジョニングし、管理するプロセス。IaC は、新しい環境を再現可能で信頼性が高く、一貫性のあるものにするため、インフラストラクチャを一元的に管理し、リソースを標準化し、スケールを迅速に行えるように設計されています。

産業分野における IoT (IIoT)

製造、エネルギー、自動車、ヘルスケア、ライフサイエンス、農業などの産業部門におけるインターネットに接続されたセンサーやデバイスの使用。詳細については、「[Building an industrial Internet of Things \(IIoT\) digital transformation strategy](#)」を参照してください。

インスペクション VPC

AWS マルチアカウントアーキテクチャでは、VPC (同一または異なる 内 AWS リージョン)、インターネット、オンプレミスネットワーク間のネットワークトラフィックの検査を管理する一元化された VPCs。 [AWS Security Reference Architecture](#) では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

IoT

インターネットまたはローカル通信ネットワークを介して他のデバイスやシステムと通信する、センサーまたはプロセッサが組み込まれた接続済み物理オブジェクトのネットワーク。詳細については、「[IoT とは](#)」を参照してください。

解釈可能性

機械学習モデルの特性で、モデルの予測がその入力にどのように依存するかを人間が理解できる度合いを表します。詳細については、「[AWS を使用した機械学習モデルの解釈](#)」を参照してください。

IoT

「[モノのインターネット](#)」を参照してください。

IT 情報ライブラリ (ITIL)

IT サービスを提供し、これらのサービスをビジネス要件に合わせるための一連のベストプラクティス。ITIL は ITSM の基盤を提供します。

IT サービス管理 (ITSM)

組織の IT サービスの設計、実装、管理、およびサポートに関連する活動。クラウドオペレーションと ITSM ツールの統合については、「[オペレーション統合ガイド](#)」を参照してください。

ITIL

「[IT 情報ライブラリ](#)」を参照してください。

ITSM

「[IT サービス管理](#)」を参照してください。

L

ラベルベースアクセス制御 (LBAC)

強制アクセス制御 (MAC) の実装で、ユーザーとデータ自体にそれぞれセキュリティラベル値が明示的に割り当てられます。ユーザーセキュリティラベルとデータセキュリティラベルが交差する部分によって、ユーザーに表示される行と列が決まります。

ランディングゾーン

ランディングゾーンは、スケーラブルで安全な、適切に設計されたマルチアカウント AWS 環境です。これは、組織がセキュリティおよびインフラストラクチャ環境に自信を持ってワークロー

ドとアプリケーションを迅速に起動してデプロイできる出発点です。ランディングゾーンの詳細については、[安全でスケーラブルなマルチアカウント AWS 環境のセットアップ](#) を参照してください。

大規模な移行

300 台以上のサーバの移行。

LBAC

[「ラベルベースのアクセスコントロール」](#) を参照してください。

最小特権

タスクの実行には必要最低限の権限を付与するという、セキュリティのベストプラクティス。詳細については、IAM ドキュメントの[最小特権アクセス許可を適用する](#) を参照してください。

リフトアンドシフト

[「7R」](#) を参照してください。

リトルエンディアンシステム

最下位バイトを最初に格納するシステム。[エンディアンネス](#) も参照してください。

下位環境

[「環境」](#) を参照してください。

M

機械学習 (ML)

パターン認識と学習にアルゴリズムと手法を使用する人工知能の一種。ML は、モノのインターネット (IoT) データなどの記録されたデータを分析して学習し、パターンに基づく統計モデルを生成します。詳細については、「[機械学習](#)」を参照してください。

メインブランチ

[「ブランチ」](#) を参照してください。

マルウェア

コンピュータのセキュリティまたはプライバシーを侵害するように設計されているソフトウェア。マルウェアは、コンピュータシステムの中断、機密情報の漏洩、不正アクセスにつながる

可能性があります。マルウェアの例としては、ウイルス、ワーム、ランサムウェア、トロイの木馬、スパイウェア、キーロガーなどがあります。

マネージドサービス

AWS のサービスがインフラストラクチャレイヤー、オペレーティングシステム、プラットフォーム AWS を運用し、ユーザーがエンドポイントにアクセスしてデータを保存および取得します。Amazon Simple Storage Service (Amazon S3) と Amazon DynamoDB は、マネージドサービスの例です。これらは抽象化されたサービスとも呼ばれます。

製造実行システム (MES)

生産プロセスを追跡、モニタリング、文書化、制御するためのソフトウェアシステム。これにより、加工品を現場の完成製品に変換します。

MAP

[「移行促進プログラム」](#) を参照してください。

メカニズム

ツールを作成し、ツールの導入を推進し、調整のために結果を検査する完全なプロセス。メカニズムとは、動作中にそれ自体を強化して改善するサイクルです。詳細については、AWS Well-Architected フレームワークの [「メカニズムの構築」](#) を参照してください。

メンバーアカウント

の組織の一部である管理アカウント AWS アカウントを除くすべての AWS Organizations。アカウントが組織のメンバーになることができるのは、一度に 1 つのみです。

MES

[「製造実行システム」](#) を参照してください。

メッセージキューイングテレメトリトランスポート (MQTT)

リソースに制約のある IoT デバイス用の、[パブリッシュ/サブスクライブ](#) パターンに基づく軽量の machine-to-machine (M2M) 通信プロトコル。

マイクロサービス

明確に定義された API を介して通信し、通常は小規模な自己完結型のチームが所有する、小規模で独立したサービスです。例えば、保険システムには、販売やマーケティングなどのビジネス機能、または購買、請求、分析などのサブドメインにマッピングするマイクロサービスが含まれる場合があります。マイクロサービスの利点には、俊敏性、柔軟なスケーリング、容易なデプロ

イ、再利用可能なコード、回復力などがあります。詳細については、[AWS 「サーバーレスサービスを使用したマイクロサービスの統合」](#)を参照してください。

マイクロサービスアーキテクチャ

各アプリケーションプロセスをマイクロサービスとして実行する独立したコンポーネントを使用してアプリケーションを構築するアプローチ。これらのマイクロサービスは、軽量 API を使用して、明確に定義されたインターフェイスを介して通信します。このアーキテクチャの各マイクロサービスは、アプリケーションの特定の機能に対する需要を満たすように更新、デプロイ、およびスケールできます。詳細については、「[でのマイクロサービスの実装 AWS](#)」を参照してください。

Migration Acceleration Program (MAP)

コンサルティングサポート、トレーニング、サービスを提供する AWS プログラム。組織がクラウドへの移行のための強固な運用基盤を構築し、移行の初期コストを相殺するのに役立ちます。MAP には、組織的な方法でレガシー移行を実行するための移行方法論と、一般的な移行シナリオを自動化および高速化する一連のツールが含まれています。

大規模な移行

アプリケーションポートフォリオの大部分を次々にクラウドに移行し、各ウェーブでより多くのアプリケーションを高速に移動させるプロセス。この段階では、以前の段階から学んだベストプラクティスと教訓を使用して、移行ファクトリー チーム、ツール、プロセスのうち、オートメーションとアジャイルデリバリーによってワークロードの移行を合理化します。これは、[AWS 移行戦略](#) の第 3 段階です。

移行ファクトリー

自動化された俊敏性のあるアプローチにより、ワークロードの移行を合理化する部門横断的なチーム。移行ファクトリーチームには、通常、オペレーション、ビジネスアナリストと所有者、移行エンジニア、デベロッパー、スプリントに取り組む DevOps プロフェッショナルが含まれます。エンタープライズアプリケーションポートフォリオの 20~50% は、ファクトリーのアプローチによって最適化できる反復パターンで構成されています。詳細については、このコンテンツセットの[移行ファクトリーに関する解説](#)と[Cloud Migration Factory ガイド](#)を参照してください。

移行メタデータ

移行を完了するために必要なアプリケーションおよびサーバーに関する情報。移行パターンごとに、異なる一連の移行メタデータが必要です。移行メタデータの例には、ターゲットサブネット、セキュリティグループ、AWS アカウントなどがあります。

移行パターン

移行戦略、移行先、および使用する移行アプリケーションまたはサービスを詳述する、反復可能な移行タスク。例: Application Migration Service を使用して Amazon EC2 AWS への移行をリホストします。

Migration Portfolio Assessment (MPA)

に移行するためのビジネスケースを検証するための情報を提供するオンラインツール AWS クラウド。MPA は、詳細なポートフォリオ評価 (サーバーの適切なサイジング、価格設定、TCO 比較、移行コスト分析) および移行プラン (アプリケーションデータの分析とデータ収集、アプリケーションのグループ化、移行の優先順位付け、およびウェブプランニング) を提供します。[MPA ツール](#) (ログインが必要) は、すべての AWS コンサルタントと APN パートナーコンサルタントが無料で利用できます。

移行準備状況評価 (MRA)

AWS CAF を使用して、組織のクラウド準備状況に関するインサイトを取得し、長所と短所を特定し、特定されたギャップを埋めるためのアクションプランを構築するプロセス。詳細については、[移行準備状況ガイド](#) を参照してください。MRA は、[AWS 移行戦略](#) の第一段階です。

移行戦略

ワークロードを に移行するために使用されるアプローチ AWS クラウド。詳細については、この用語集の「[7 Rs エントリ](#)」と「[組織を動員して大規模な移行を加速する](#)」を参照してください。

ML

[「機械学習」を参照してください。](#)

モダナイゼーション

古い (レガシーまたはモノリシック) アプリケーションとそのインフラストラクチャをクラウド内の俊敏で弾力性のある高可用性システムに変換して、コストを削減し、効率を高め、イノベーションを活用します。詳細については、「」の「[アプリケーションをモダナイズするための戦略 AWS クラウド](#)」を参照してください。

モダナイゼーション準備状況評価

組織のアプリケーションのモダナイゼーションの準備状況を判断し、利点、リスク、依存関係を特定し、組織がこれらのアプリケーションの将来の状態をどの程度適切にサポートできるかを決定するのに役立つ評価。評価の結果として、ターゲットアーキテクチャのブループリント、モダナイゼーションプロセスの開発段階とマイルストーンを詳述したロードマップ、特定され

たギャップに対処するためのアクションプランが得られます。詳細については、[「」の「アプリケーションのモダナイゼーション準備状況の評価 AWS クラウド」](#)を参照してください。

モノリシックアプリケーション (モノリス)

緊密に結合されたプロセスを持つ単一のサービスとして実行されるアプリケーション。モノリシックアプリケーションにはいくつかの欠点があります。1つのアプリケーション機能エクスペリエンスの需要が急増する場合は、アーキテクチャ全体をスケーリングする必要があります。モノリシックアプリケーションの特徴を追加または改善することは、コードベースが大きくなると複雑になります。これらの問題に対処するには、マイクロサービスアーキテクチャを使用できます。詳細については、[モノリスをマイクロサービスに分解する](#)を参照してください。

MPA

[「移行ポートフォリオ評価」](#)を参照してください。

MQTT

[「Message Queuing Telemetry Transport」](#)を参照してください。

多クラス分類

複数のクラスの予測を生成するプロセス (2 つ以上の結果の 1 つを予測します)。例えば、機械学習モデルが、「この製品は書籍、自動車、電話のいずれですか?」または、「このお客様にとって最も関心のある商品のカテゴリはどれですか?」と聞くかもしれません。

変更可能なインフラストラクチャ

本番ワークロードの既存のインフラストラクチャを更新および変更するモデル。Well-Architected AWS Framework では、一貫性、信頼性、予測可能性を向上させるために、[イミュータブルなインフラストラクチャ](#)の使用をベストプラクティスとして推奨しています。

O

OAC

[「オリジンアクセスコントロール」](#)を参照してください。

OAI

[「オリジンアクセスアイデンティティ」](#)を参照してください。

OCM

[「組織変更管理」](#)を参照してください。

オフライン移行

移行プロセス中にソースワークロードを停止させる移行方法。この方法はダウンタイムが長くなるため、通常は重要ではない小規模なワークロードに使用されます。

OI

「[オペレーション統合](#)」を参照してください。

OLA

「[運用レベルの契約](#)」を参照してください。

オンライン移行

ソースワークロードをオフラインにせずターゲットシステムにコピーする移行方法。ワークロードに接続されているアプリケーションは、移行中も動作し続けることができます。この方法はダウンタイムがゼロから最小限で済むため、通常は重要な本番稼働環境のワークロードに使用されます。

OPC-UA

「[Open Process Communications - Unified Architecture](#)」を参照してください。

オープンプロセス通信 - 統合アーキテクチャ (OPC-UA)

産業オートメーション用の machine-to-machine (M2M) 通信プロトコル。OPC-UA は、データの暗号化、認証、認可スキームを備えた相互運用性標準を提供します。

オペレーショナルレベルアグリーメント (OLA)

サービスレベルアグリーメント (SLA) をサポートするために、どの機能的 IT グループが互いに提供することを約束するかを明確にする契約。

運用準備状況レビュー (ORR)

インシデントや潜在的な障害の理解、評価、防止、または範囲の縮小に役立つ質問とそれに関連するベストプラクティスのチェックリスト。詳細については、AWS Well-Architected フレームワークの「[運用準備状況レビュー \(ORR\)](#)」を参照してください。

運用テクノロジー (OT)

産業運用、機器、インフラストラクチャを制御するために物理環境と連携するハードウェアおよびソフトウェアシステム。製造では、OT と情報技術 (IT) システムの統合が、[Industry 4.0](#) トランスフォーメーションの主要な焦点です。

オペレーション統合 (OI)

クラウドでオペレーションをモダナイズするプロセスには、準備計画、オートメーション、統合が含まれます。詳細については、[オペレーション統合ガイド](#) を参照してください。

組織の証跡

の組織 AWS アカウント 内のすべての のすべてのイベントをログ AWS CloudTrail に記録する、によって作成された証跡 AWS Organizations。証跡は、組織に含まれている各 AWS アカウントに作成され、各アカウントのアクティビティを追跡します。詳細については、ドキュメントの [「組織の証跡の作成」](#) を参照してください。CloudTrail

組織変更管理 (OCM)

人材、文化、リーダーシップの観点から、主要な破壊的なビジネス変革を管理するためのフレームワーク。OCM は、変化の導入を加速し、移行問題に対処し、文化や組織の変化を推進することで、組織が新しいシステムと戦略の準備と移行するのを支援します。AWS 移行戦略では、クラウド導入プロジェクトに必要な変化のスピードから、このフレームワークは人材アクセラレーションと呼ばれます。詳細については、[OCM ガイド](#) を参照してください。

オリジンアクセスコントロール (OAC)

では CloudFront、Amazon Simple Storage Service (Amazon S3) コンテンツを保護するためのアクセスを制限するための拡張オプションです。OAC は、すべての 内のすべての S3 バケット AWS リージョン、AWS KMS (SSE-KMS) によるサーバー側の暗号化、および S3 バケットへの動的 PUT および DELETE リクエストをサポートします。

オリジンアクセスアイデンティティ (OAI)

では CloudFront、Amazon S3 コンテンツを保護するためのアクセスを制限するオプションです。OAI を使用する場合は、Amazon S3 が認証できるプリンシパル CloudFront を作成します。認証されたプリンシパルは、特定の CloudFront デイストリビューションを介してのみ S3 バケット内のコンテンツにアクセスできます。[OAC](#) も併せて参照してください。OAC では、より詳細な、強化されたアクセスコントロールが可能です。

ORR

[「運用準備状況レビュー」](#) を参照してください。

OT

[「運用技術」](#) を参照してください。

アウトバウンド (送信) VPC

AWS マルチアカウントアーキテクチャでは、アプリケーション内から開始されるネットワーク接続を処理する VPC。[AWS Security Reference Architecture](#) では、アプリケーションとより広範なインターネット間の双方向のインターフェイスを保護するために、インバウンド、アウトバウンド、インスペクションの各 VPC を使用してネットワークアカウントを設定することを推奨しています。

P

アクセス許可の境界

ユーザーまたはロールが使用できるアクセス許可の上限を設定する、IAM プリンシパルにアタッチされる IAM 管理ポリシー。詳細については、IAM ドキュメントの[アクセス許可の境界](#)を参照してください。

個人を特定できる情報 (PII)

直接閲覧した場合、または他の関連データと組み合わせた場合に、個人の身元を合理的に推測するために使用できる情報。PII の例には、氏名、住所、連絡先情報などがあります。

PII

[「個人を特定できる情報」](#)を参照してください。

プレイブック

クラウドでのコアオペレーション機能の提供など、移行に関連する作業を取り込む、事前定義された一連のステップ。プレイブックは、スクリプト、自動ランブック、またはお客様のモダナイズされた環境を運用するために必要なプロセスや手順の要約などの形式をとることができます。

PLC

[「プログラム可能なロジックコントローラー」](#)を参照してください。

PLM

[「製品ライフサイクル管理」](#)を参照してください。

ポリシー

アクセス許可の定義 ([アイデンティティベースのポリシー](#) を参照)、アクセス条件の指定 ([リソースベースのポリシー](#) を参照)、または の組織内のすべてのアカウントに対する最大アクセス許可の定義 AWS Organizations ([サービスコントロールポリシー](#) を参照) が可能なオブジェクト。

多言語の永続性

データアクセスパターンやその他の要件に基づいて、マイクロサービスのデータストレージテクノロジーを個別に選択します。マイクロサービスが同じデータストレージテクノロジーを使用している場合、実装上の問題が発生したり、パフォーマンスが低下する可能性があります。マイクロサービスは、要件に最も適合したデータストアを使用すると、より簡単に実装でき、パフォーマンスとスケーラビリティが向上します。詳細については、[マイクロサービスでのデータ永続性の有効化](#) を参照してください。

ポートフォリオ評価

移行を計画するために、アプリケーションポートフォリオの検出、分析、優先順位付けを行うプロセス。詳細については、「[移行準備状況ガイド](#)」を参照してください。

述語

true または を返すクエリ条件。false 通常は WHERE 句にあります。

述語のプッシュダウン

転送前にクエリ内のデータをフィルタリングするデータベースクエリ最適化手法。これにより、リレーショナルデータベースから取得して処理する必要があるデータの量が減少し、クエリのパフォーマンスが向上します。

予防的コントロール

イベントの発生を防ぐように設計されたセキュリティコントロール。このコントロールは、ネットワークへの不正アクセスや好ましくない変更を防ぐ最前線の防御です。詳細については、Implementing security controls on AWS の [Preventative controls](#) を参照してください。

プリンシパル

アクションを実行し AWS、リソースにアクセスできるのエンティティ。このエンティティは通常、IAM ロール AWS アカウント、またはユーザーのルートユーザーです。詳細については、IAM ドキュメントの [ロールに関する用語と概念](#) 内にあるプリンシパルを参照してください。

プライバシーバイデザイン

エンジニアリングプロセス全体を通してプライバシーを考慮に入れたシステムエンジニアリングのアプローチ。

プライベートホストゾーン

1 つ以上の VPC 内のドメインとそのサブドメインへの DNS クエリに対し、Amazon Route 53 がどのように応答するかに関する情報を保持するコンテナ。詳細については、Route 53 ドキュメントの「[プライベートホストゾーンの使用](#)」を参照してください。

プロアクティブコントロール

非準拠のリソースのデプロイを防止するように設計された[セキュリティコントロール](#)。これらのコントロールは、プロビジョニング前にリソースをスキャンします。リソースがコントロールに準拠していない場合、プロビジョニングされません。詳細については、AWS Control Tower ドキュメントの「[コントロールリファレンスガイド](#)」および「[でのセキュリティコントロールの実装](#)」の「[プロアクティブコントロール](#)」を参照してください。 AWS

製品ライフサイクル管理 (PLM)

設計、開発、発売から成長と成熟まで、製品のデータとプロセスのライフサイクル全体にわたる管理、および辞退と削除。

本番環境

[「環境」](#)を参照してください。

プログラミング可能ロジックコントローラー (NAL)

製造では、マシンをモニタリングし、承認プロセスを自動化する、信頼性が高く、適応性の高いコンピュータです。

仮名化

データセット内の個人識別子をプレースホルダー値に置き換えるプロセス。仮名化は個人のプライバシー保護に役立ちます。仮名化されたデータは、依然として個人データとみなされます。

パブリッシュ/サブスクライブ (pub/sub)

マイクロサービス間の非同期通信を可能にするパターン。スケーラビリティと応答性を向上させます。例えば、マイクロサービスベースの[MES](#)では、マイクロサービスは他のマイクロサービスがサブスクライブできるチャンネルにイベントメッセージを発行できます。システムは、公開サービスを変更せずに新しいマイクロサービスを追加できます。

Q

クエリプラン

SQL リレーショナルデータベースシステムのデータにアクセスするために使用される手順などの一連のステップ。

クエリプランのリグレッション

データベースサービスのオプティマイザーが、データベース環境に特定の変更が加えられる前に選択されたプランよりも最適性の低いプランを選択すること。これは、統計、制限事項、環境設

定、クエリパラメータのバインディングの変更、およびデータベースエンジンの更新などが原因である可能性があります。

R

RACI マトリックス

[責任、説明責任、相談、情報 \(RACI\)](#) を参照してください。

ランサムウェア

決済が完了するまでコンピュータシステムまたはデータへのアクセスをブロックするように設計された、悪意のあるソフトウェア。

RASCI マトリックス

[責任、説明責任、相談、情報 \(RACI\)](#) を参照してください。

RCAC

[「行と列のアクセスコントロール」](#) を参照してください。

リードレプリカ

読み取り専用で使用されるデータベースのコピー。クエリをリードレプリカにルーティングして、プライマリデータベースへの負荷を軽減できます。

再構築

[「7 Rs」](#) を参照してください。

目標復旧時点 (RPO)

最後のデータリカバリポイントからの最大許容時間です。これにより、最後の回復時点からサービスが中断されるまでの間に許容できるデータ損失の程度が決まります。

目標復旧時間 (RTO)

サービスの中断から復旧までの最大許容遅延時間。

リファクタリング

[「7 R」](#) を参照してください。

リージョン

地理的エリア内の AWS リソースのコレクション。各 AWS リージョンは、耐障害性、安定性、耐障害性を提供するために、他のから分離され、独立しています。詳細については、[AWS リージョン「を使用できるアカウントを指定する」](#)を参照してください。

回帰

数値を予測する機械学習手法。例えば、「この家はどれくらいの値段で売れるでしょうか?」という問題を解決するために、機械学習モデルは、線形回帰モデルを使用して、この家に関する既知の事実 (平方フィートなど) に基づいて家の販売価格を予測できます。

リホスト

[「7 R」を参照してください。](#)

リリース

デプロイプロセスで、変更を本番環境に昇格させること。

再配置

[「7 Rs」を参照してください。](#)

プラットフォーム変更

[「7 R」を参照してください。](#)

再購入

[「7 Rs」を参照してください。](#)

回復性

中断に耐えたり、中断から回復したりするアプリケーションの機能。で障害耐性を計画する場合、[高可用性](#)と[ディザスタリカバリ](#)が一般的な考慮事項です AWS クラウド。詳細については、[AWS クラウド「レジリエンス」](#)を参照してください。

リソースベースのポリシー

Amazon S3 バケット、エンドポイント、暗号化キーなどのリソースにアタッチされたポリシー。このタイプのポリシーは、アクセスが許可されているプリンシパル、サポートされているアクション、その他の満たすべき条件を指定します。

実行責任者、説明責任者、協業先、報告先 (RACI) に基づくマトリックス

移行活動とクラウド運用に関わるすべての関係者の役割と責任を定義したマトリックス。マトリックスの名前は、マトリックスで定義されている責任の種類、すなわち責任 (R)、説明責任

(A)、協議 (C)、情報提供 (I) に由来します。サポート (S) タイプはオプションです。サポートを含めると、そのマトリックスは RASCI マトリックスと呼ばれ、サポートを除外すると RACI マトリックスと呼ばれます。

レスポンスコントロール

有害事象やセキュリティベースラインからの逸脱について、修復を促すように設計されたセキュリティコントロール。詳細については、Implementing security controls on AWSの[Responsive controls](#)を参照してください。

保持

[「7 Rs」を参照してください。](#)

廃止

[「7 Rs」を参照してください。](#)

ローテーション

定期的に[シークレット](#)を更新して、攻撃者が認証情報にアクセスするのをより困難にするプロセス。

行と列のアクセス制御 (RCAC)

アクセスルールが定義された、基本的で柔軟な SQL 表現の使用。RCAC は行権限と列マスクで構成されています。

RPO

「目標[復旧時点](#)」を参照してください。

RTO

「目標[復旧時間](#)」を参照してください。

ランブック

特定のタスクを実行するために必要な手動または自動化された一連の手順。これらは通常、エラー率の高い反復操作や手順を合理化するために構築されています。

S

SAML 2.0

多くの ID プロバイダー (IdPs) が使用するオープンスタンダード。この機能により、フェデレーテッドシングルサインオン (SSO) が有効になるため、ユーザーは [AWS](#)

Management Console したり、組織内のすべてのユーザーを IAM で作成しなくても AWS API オペレーションを呼び出すことができます。SAML 2.0 ベースのフェデレーションの詳細については、IAM ドキュメントの[SAML 2.0 ベースのフェデレーションについて](#)を参照してください。

SCADA

[「監視コントロールとデータ収集」](#)を参照してください。

SCP

[「サービスコントロールポリシー」](#)を参照してください。

シークレット

では AWS Secrets Manager、暗号化された形式で保存するパスワードやユーザー認証情報などの機密情報または制限付き情報。シークレット値とそのメタデータで構成されます。シークレット値は、バイナリ、1つの文字列、または複数の文字列にすることができます。詳細については、[Secrets Manager ドキュメントの「Secrets Manager シークレットの内容」](#)を参照してください。

セキュリティコントロール

脅威アクターによるセキュリティ脆弱性の悪用を防止、検出、軽減するための、技術上または管理上のガードレール。セキュリティコントロールには、[予防的](#)、[検出的](#)、[???応答的](#)、[プロアクティブ](#)の4つの主なタイプがあります。

セキュリティ強化

アタックサーフェスを狭めて攻撃への耐性を高めるプロセス。このプロセスには、不要になったリソースの削除、最小特権を付与するセキュリティのベストプラクティスの実装、設定ファイル内の不要な機能の無効化、といったアクションが含まれています。

Security Information and Event Management (SIEM) システム

セキュリティ情報管理 (SIM) とセキュリティイベント管理 (SEM) のシステムを組み合わせたツールとサービス。SIEM システムは、サーバー、ネットワーク、デバイス、その他ソースからデータを収集、モニタリング、分析して、脅威やセキュリティ違反を検出し、アラートを発信します。

セキュリティレスポンスの自動化

セキュリティイベントに自動的に応答または修正するように設計された、事前定義されプログラムされたアクション。これらの自動化は、セキュリティのベストプラクティスの実装に役立つ[検出的](#)または[応答的](#)な AWS セキュリティコントロールとして機能します。自動レスポンスアク

シヨンの例としては、VPC セキュリティグループの変更、Amazon EC2 インスタンスへのパッチ適用、認証情報のローテーションなどがあります。

サーバー側の暗号化

送信先にあるデータの、それを受け取る AWS のサービス による暗号化。

サービスコントロールポリシー (SCP)

AWS Organizationsの組織内の、すべてのアカウントのアクセス許可を一元的に管理するポリシー。SCP は、管理者がユーザーまたはロールに委任するアクションに、ガードレールを定義したり、アクションの制限を設定したりします。SCP は、許可リストまたは拒否リストとして、許可または禁止するサービスやアクションを指定する際に使用できます。詳細については、AWS Organizations ドキュメントの「[サービスコントロールポリシー](#)」を参照してください。

サービスエンドポイント

のエントリポイントの URL AWS のサービス。ターゲットサービスにプログラムで接続するには、エンドポイントを使用します。詳細については、AWS 全般のリファレンスの「[AWS のサービス エンドポイント](#)」を参照してください。

サービスレベルアグリーメント (SLA)

サービスのアップタイムやパフォーマンスなど、IT チームがお客様に提供すると約束したものを明示した合意書。

サービスレベルインジケータ (SLI)

エラー率、可用性、スループットなど、サービスのパフォーマンス側面の測定。

サービスレベルの目標 (SLO)

サービスレベルのインジケータによって測定される、サービスの状態を表すターゲットメトリクス。

責任共有モデル

クラウドのセキュリティとコンプライアンス AWS について と共有する責任を説明するモデル。AWS はクラウドのセキュリティを担当しますが、お客様はクラウドのセキュリティを担当します。詳細については、[責任共有モデル](#)を参照してください。

SIEM

「[セキュリティ情報とイベント管理システム](#)」を参照してください。

単一障害点 (SPOF)

システムを中断させる可能性のあるアプリケーションの単一の重要なコンポーネントの障害。

SLA

[「サービスレベルアグリーメント」](#) を参照してください。

SLI

[「サービスレベルインジケータ」](#) を参照してください。

SLO

[「サービスレベルの目標」](#) を参照してください。

split-and-seed モデル

モダナイゼーションプロジェクトのスケールアップと加速のためのパターン。新機能と製品リリースが定義されると、コアチームは解放されて新しい製品チームを作成します。これにより、お客様の組織の能力とサービスの拡張、デベロッパーの生産性の向上、迅速なイノベーションのサポートに役立ちます。詳細については、[「」の「アプリケーションをモダナイズするための段階的アプローチ AWS クラウド」](#) を参照してください。

SPOF

[単一障害点](#) を参照してください。

star スキーマ

トランザクションデータまたは測定データを保存するために 1 つの大きなファクトテーブルを使用し、データ属性を保存するために 1 つ以上の小さなディメンションテーブルを使用するデータベースの組織構造。この構造は、[データウェアハウス](#) または [ビジネスインテリジェンス](#) の目的で使用するように設計されています。

strangler fig パターン

レガシーシステムが廃止されるまで、システム機能を段階的に書き換えて置き換えることにより、モノリシックシステムをモダナイズするアプローチ。このパターンは、宿主の樹木から根を成長させ、最終的にその宿主を包み込み、宿主に取って代わるイチジクのつるを例えています。そのパターンは、モノリシックシステムを書き換えるときのリスクを管理する方法として [Martin Fowler](#) により提唱されました。このパターンの適用方法の例については、[コンテナと Amazon API Gateway を使用して、従来の Microsoft ASP.NET \(ASMX\) ウェブサービスを段階的にモダナイズ](#) を参照してください。

サブネット

VPC 内の IP アドレスの範囲。サブネットは、1 つのアベイラビリティゾーンに存在する必要があります。

監視統制とデータ収集 (SCADA)

製造では、ハードウェアとソフトウェアを使用して物理アセットと生産オペレーションをモニタリングするシステム。

対称暗号化

データの暗号化と復号に同じキーを使用する暗号化のアルゴリズム。

合成テスト

ユーザーインタラクションをシミュレートして潜在的な問題を検出したり、パフォーマンスをモニタリングしたりする方法でシステムをテストします。[Amazon CloudWatch Synthetics](#) を使用してこれらのテストを作成できます。

T

タグ

AWS リソースを整理するためのメタデータとして機能するキーと値のペア。タグは、リソースの管理、識別、整理、検索、フィルタリングに役立ちます。詳細については、「[AWS リソースのタグ付け](#)」を参照してください。

ターゲット変数

監督された機械学習でお客様が予測しようとしている値。これは、結果変数のことも指します。例えば、製造設定では、ターゲット変数が製品の欠陥である可能性があります。

タスクリスト

ランブックの進行状況を追跡するために使用されるツール。タスクリストには、ランブックの概要と完了する必要がある一般的なタスクのリストが含まれています。各一般的なタスクには、推定所要時間、所有者、進捗状況が含まれています。

テスト環境

[「環境」](#) を参照してください。

トレーニング

お客様の機械学習モデルに学習するデータを提供すること。トレーニングデータには正しい答えが含まれている必要があります。学習アルゴリズムは入力データ属性をターゲット (お客様が予測したい答え) にマッピングするトレーニングデータのパターンを検出します。これらのパター

ンをキャプチャする機械学習モデルを出力します。そして、お客様が機械学習モデルを使用して、ターゲットがわからない新しいデータでターゲットを予測できます。

トランジットゲートウェイ

VPC と オンプレミス ネットワーク を相互接続するために使用できる、ネットワークの中継ハブ。詳細については、AWS Transit Gateway ドキュメントの「[トランジットゲートウェイとは](#)」を参照してください。

トランクベースのワークフロー

デベロッパーが機能ブランチで機能をローカルにビルドしてテストし、その変更をメインブランチにマージするアプローチ。メインブランチはその後、開発環境、本番前環境、本番環境に合わせて順次構築されます。

信頼されたアクセス

ユーザーに代わって AWS Organizations およびそのアカウントで組織内でタスクを実行するために指定するサービスへのアクセス許可を付与します。信頼されたサービスは、サービスにリンクされたロールを必要とときに各アカウントに作成し、ユーザーに代わって管理タスクを実行します。詳細については、ドキュメントの「[AWS Organizations を他の AWS のサービスで使用する AWS Organizations](#)」を参照してください。

チューニング

機械学習モデルの精度を向上させるために、お客様のトレーニングプロセスの側面を変更する。例えば、お客様が機械学習モデルをトレーニングするには、ラベル付けセットを生成し、ラベルを追加します。これらのステップを、異なる設定で複数回繰り返して、モデルを最適化します。

ツーピザチーム

2 つのピザを食べることができる小さな DevOps チーム。ツーピザチームの規模では、ソフトウェア開発におけるコラボレーションに最適な機会が確保されます。

U

不確実性

予測機械学習モデルの信頼性を損なう可能性がある、不正確、不完全、または未知の情報を指す概念。不確実性には、次の 2 つのタイプがあります。認識論的不確実性は、限られた、不完全なデータによって引き起こされ、弁論的不確実性は、データに固有のノイズとランダム性によって引き起こされます。詳細については、[深層学習システムにおける不確実性の定量化](#) ガイドを参照してください。

未分化なタスク

ヘビーリフティングとも呼ばれ、アプリケーションの作成と運用には必要だが、エンドユーザーに直接的な価値をもたらさなかったり、競争上の優位性をもたらしたりしない作業です。未分化なタスクの例としては、調達、メンテナンス、キャパシティプランニングなどがあります。

上位環境

[「環境」](#)を参照してください。

V

バキューミング

ストレージを再利用してパフォーマンスを向上させるために、増分更新後にクリーンアップを行うデータベースのメンテナンス操作。

バージョンコントロール

リポジトリ内のソースコードへの変更など、変更を追跡するプロセスとツール。

VPC ピアリング

プライベート IP アドレスを使用してトラフィックをルーティングできる、2 つの VPC 間の接続。詳細については、Amazon VPC ドキュメントの「[VPC ピア機能とは](#)」を参照してください。

脆弱性

システムのセキュリティを脅かすソフトウェアまたはハードウェアの欠陥。

W

ウォームキャッシュ

頻繁にアクセスされる最新の関連データを含むバッファキャッシュ。データベースインスタンスはバッファキャッシュから、メインメモリまたはディスクからよりも短い時間で読み取りを行うことができます。

ウォームデータ

アクセス頻度の低いデータ。この種類のデータをクエリする場合、通常は適度に遅いクエリでも問題ありません。

ウィンドウ関数

現在のレコードに関連する行のグループに対して計算を実行する SQL 関数。ウィンドウ関数は、移動平均の計算や、現在の行の相対位置に基づく行の値へのアクセスなどのタスクの処理に役立ちます。

ワークロード

ビジネス価値をもたらすリソースとコード (顧客向けアプリケーションやバックエンドプロセスなど) の総称。

ワークストリーム

特定のタスクセットを担当する移行プロジェクト内の機能グループ。各ワークストリームは独立していますが、プロジェクト内の他のワークストリームをサポートしています。たとえば、ポートフォリオワークストリームは、アプリケーションの優先順位付け、ウェーブ計画、および移行メタデータの収集を担当します。ポートフォリオワークストリームは、これらの設備を移行ワークストリームで実現し、サーバーとアプリケーションを移行します。

WORM

[「書き込み 1 回」](#)を参照し、[多くの](#)を読み取ります。

WQF

[「AWS ワークロード認定フレームワーク」](#)を参照してください。

Write Once, Read Many (WORM)

データを 1 回書き込み、データの削除や変更を防ぐストレージモデル。承認されたユーザーは、必要な回数だけデータを読み取ることができますが、変更することはできません。このデータストレージインフラストラクチャは [イミュータブルな](#) と見なされます。

Z

ゼロデイ 익스プロイト

[ゼロデイ脆弱性](#) を利用する攻撃、通常はマルウェア。

ゼロデイ脆弱性

実稼働システムにおける未解決の欠陥または脆弱性。脅威アクターは、このような脆弱性を利用してシステムを攻撃する可能性があります。開発者は、よく攻撃の結果で脆弱性に気付きます。

ゾンビアプリケーション

平均 CPU およびメモリ使用率が 5% 未満のアプリケーション。移行プロジェクトでは、これらのアプリケーションを廃止するのが一般的です。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。