[AWS Black Belt Online Seminar]
AWS Step Functions

サービスカットシリーズ

Solutions Architect 今村 優太

AWS 公式 Webinar
https://amzn.to/JPWebinar

過去資料
https://amzn.to/JPArchive

自己紹介

- 名前
  - 今村 優太
- 所属
  - アマゾン ウェブ サービス ジャパン 株式会社
  - 技術統括本部
  - ソリューション アーキテクト
- 好きなAWSのサービス

AWS Lambda

AWS Step Functions

AWS AppSync

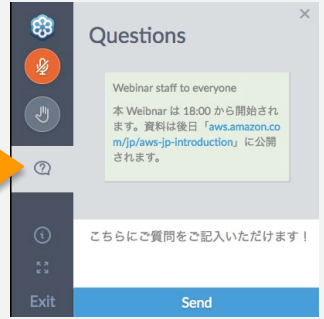
AWS Black Belt Online Seminar とは

「サービス別」「ソリューション別」「業種別」のそれぞれのテーマに分かれて、アマゾン ウェブ サービス ジャパン株式会社が主催するオンラインセミナーシリーズです。

質問を投げることができます！

- 書き込んだ質問は、主催者にしか見えません
- 今後のロードマップに関するご質問は
  お答えできませんのでご了承下さい

① 吹き出しをクリック
② 質問を入力
③ Sendをクリック



Twitter 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2019年5月22日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。最新の情報はAWS公式ウェブサイト(http://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が東京リージョンを使用する場合、別途消費税をご請求させていただきます。

- AWS does not offer binding price quotes. AWS pricing is publicly available and is subject to change in accordance with the AWS Customer Agreement available at http://aws.amazon.com/agreement/. Any pricing information included in this document is provided only as an estimate of usage charges for AWS services based on certain information that you have provided. Monthly charges will be based on your actual use of AWS services, and may vary from the estimates provided.

本日のアジェンダ

- Step Functions 概要
- ステートマシン
- データの入出力
- State の記述
- 実行状況の確認
- 補足や料金詳細など

本日のアジェンダ

- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

aws

The Twelve-Factor App: モダンなアプリ開発のベストプラクティス

(抜粋) Twelve-Factor のプロセスはステートレスかつシェアードナッシング である。永続化する必要のあるすべてのデータは、ステートフルなバックエンドサービス（典型的にはデータベース）に格納しなければならない。

THE TWELVE-FACTOR APP

VI. プロセス

アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

アプリケーションは、実行環境の中で1つもしくは複数の *プロセス* として実行される。

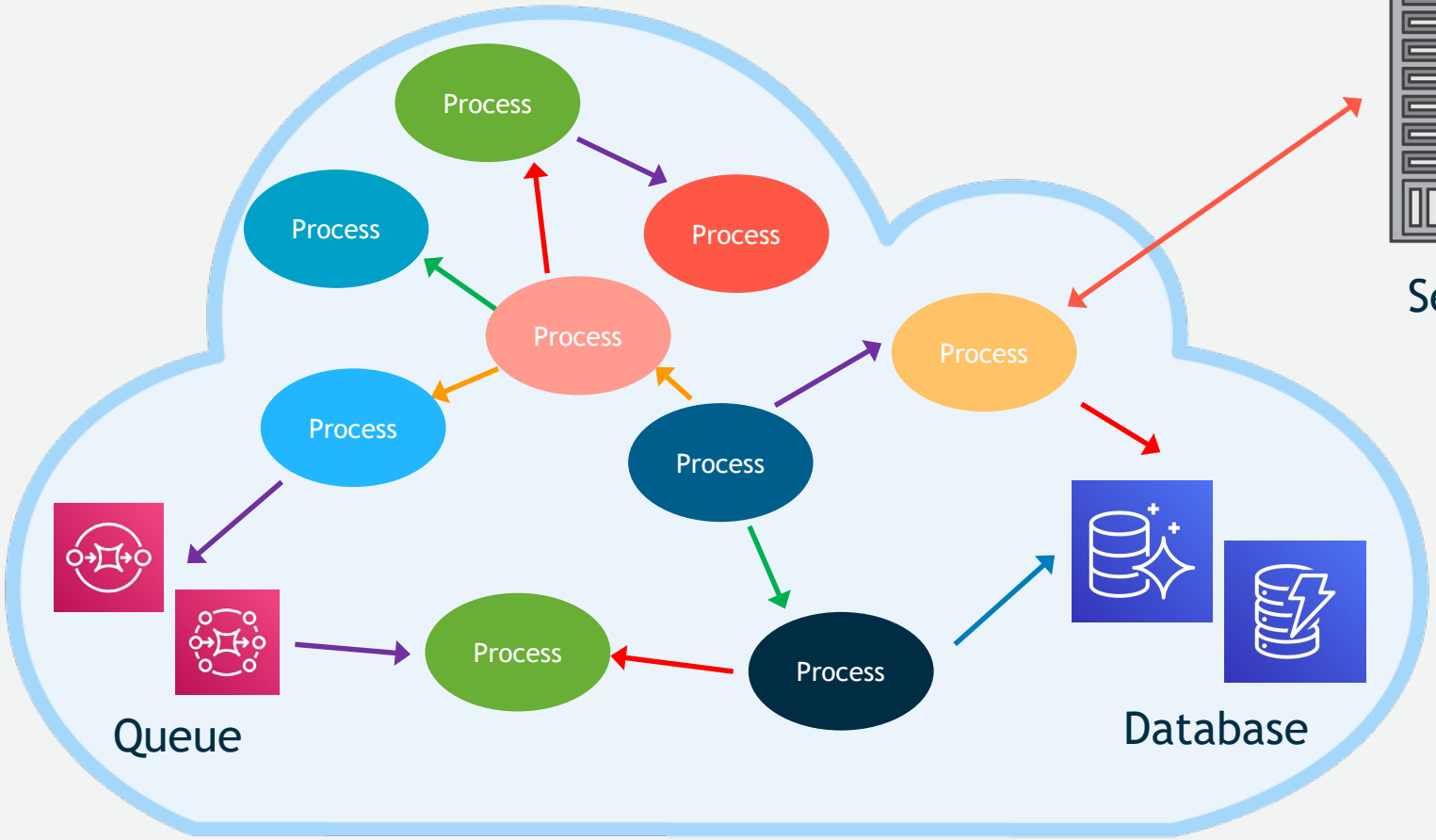
最も単純な場合では、コードは単体のスクリプトであり、実行環境は言語ランタイムがインストールされた開発者のローカルノートPCであり、プロセスはコマンドラインから実行される（例：`python my_script.py`）。対極にあるのは、複数の実行プロセスとしてインスタンス化される多くのプロセスタイプを使う洗練されたアプリケーションの本番デプロイである。

Twelve-Factorのプロセスはステートレスかつシェアードナッシング である。永続化する必要のあるすべてのデータは、ステートフルなバックエンドサービス（典型的にはデータベース）に格納しなければならない。

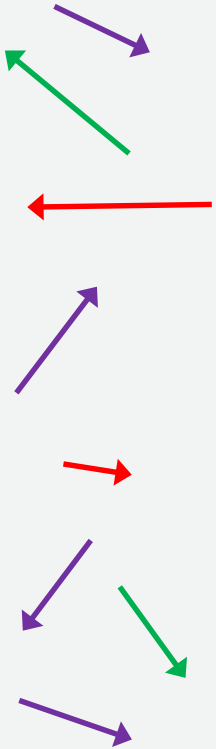
https://12factor.net/ja/

aws

現代のアプリケーション：複数のプロセスが密接に関連



現代のアプリケーションを構成するには...



"複数のプロセスを同期実行させたい"

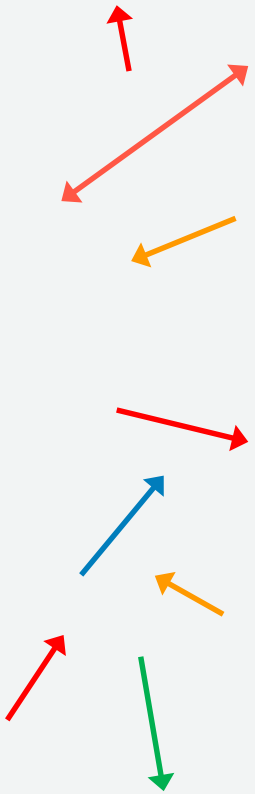
"複数のプロセスを並列実行させたい"

"条件分岐させたい"

"一定条件でリトライさせたい"

"例外やエラーを補足したい"

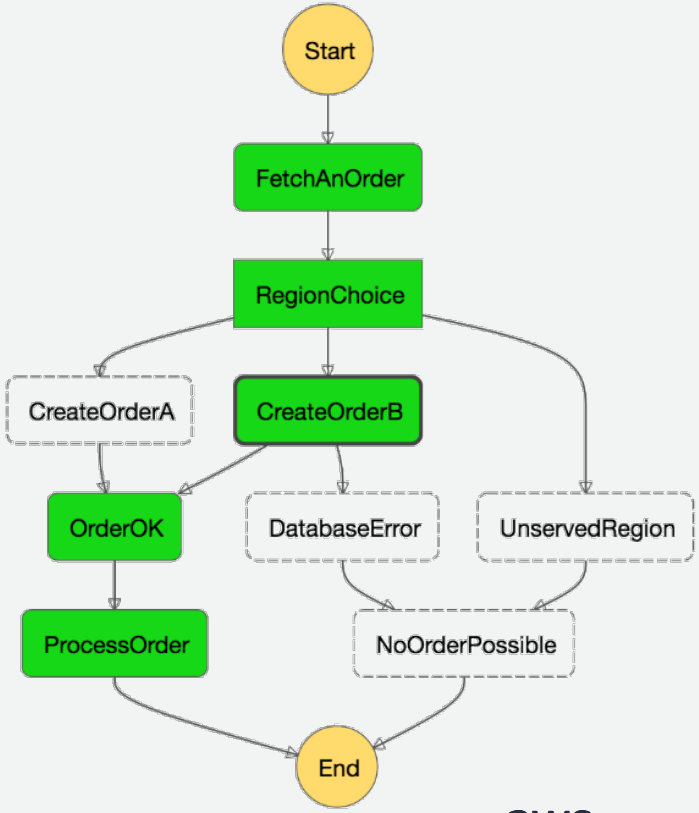
"状態を次のプロセスに連携させたい"



aws

AWS Step Functions

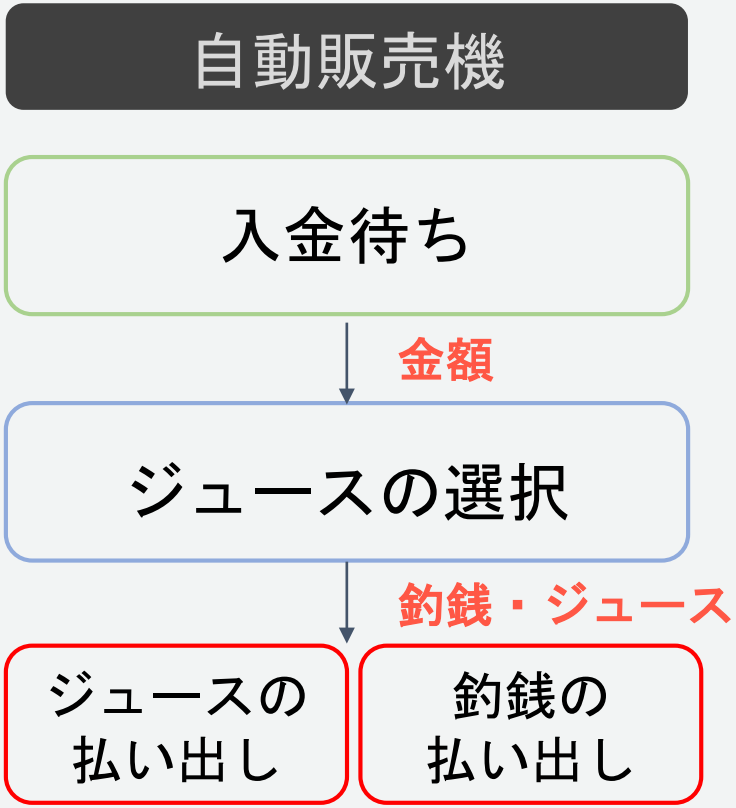
- 分散アプリケーション・マイクロサービスの全体を「ステートマシン」(後述) と呼ばれる仕組みでオーケストレートできる
- 定義したステートマシンは AWS コンソールから「ワークフロー」(右図) という形式で見やすく可視化できる
- ステートマシンの各ステップの実行履歴をログから追跡できる



本日のアジェンダ

- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

日常のステートマシンの例：自動販売機



例えば 3 つのプロセスに分離できる。

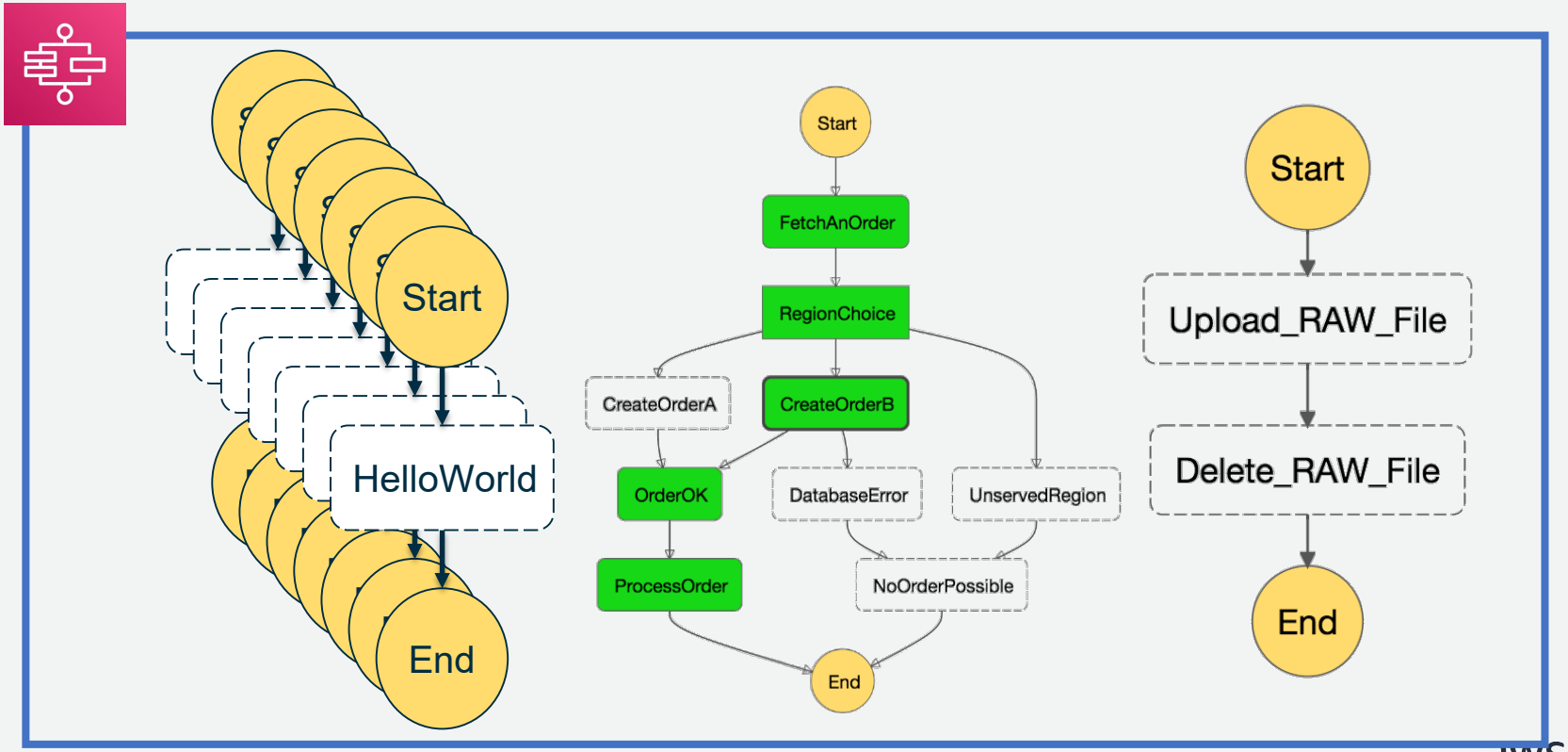
利用者が入金するまで待機する。
ジュースの購入に十分な金額が入金されたら、金額を次のプロセスに伝える。

利用者がジュースを選択するまで待機する。
ジュースが選択されたら、入金額からジュースの金額を差し引き、釣銭の金額を計算する。
釣銭の金額と、選択されたジュースを次のプロセスに伝える。

釣銭と選択されたジュースを実際に受け取り口に払い出す。
この 2 つのプロセスは並列に実行できる。

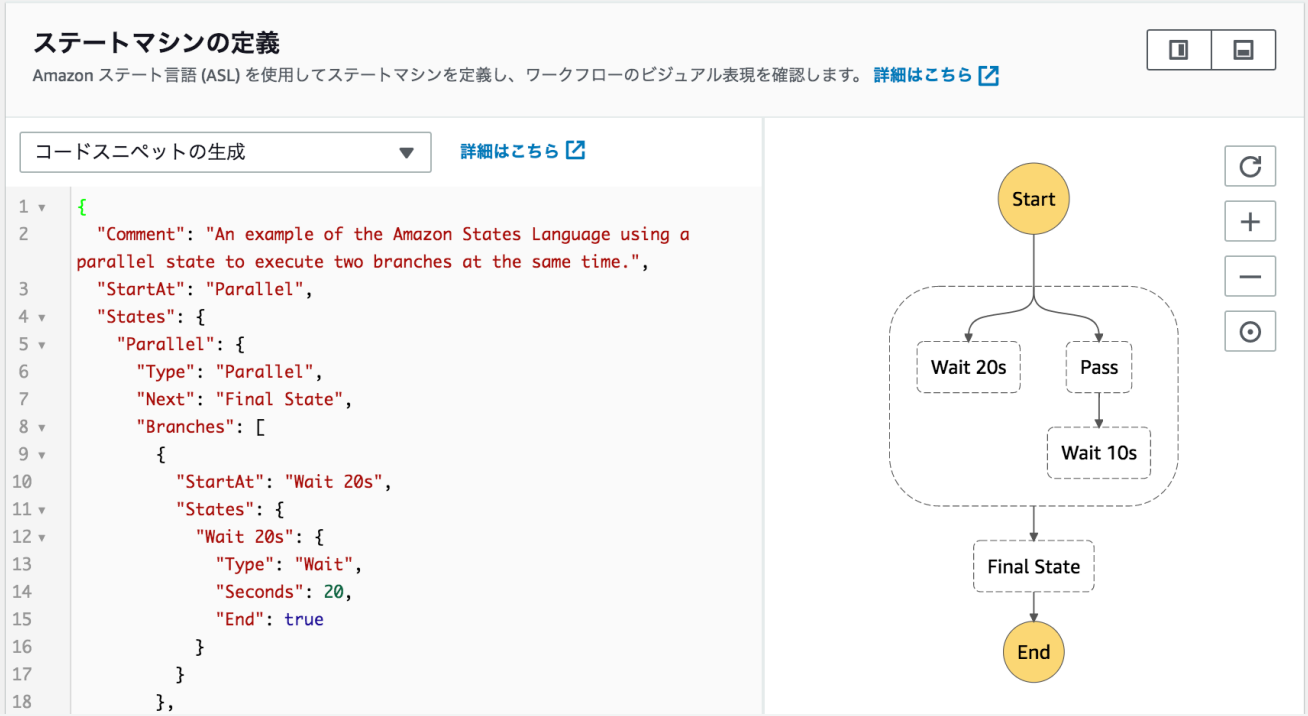
同時実行可能

Step Functions で作成したステートマシンは、複数種類を同時に実行が可能。



ステートマシンの作成

ASL (Amazon States Language) と呼ばれる JSON 形式の言語でワークフローを定義。



ASL のチェックツール

awslabs/statelint　https://github.com/awslabs/statelint

- Ruby ベースの ASL 記述チェックツール
- ‘gem install statelint’ でインストール
- ‘statelint <ASL のファイル名>’ で検証。不正であればエラーを表示。

Parallel を parallel にtypo

```
{
"StartAt": "Parallel",
"States": {
  "parallel": {
    "Type": "Parallel",
    "End": true,
```

検証

```
2 errors:
 StartAt value Parallel not found in States field at State Machine
 No transition found to state State Machine.parallel
```

Step Functions におけるステートマシンの例

アップロードされた画像に対して、自動でタグ付けとサムネイル化を行うようなアプリケーションの場合、このようなステートマシンが考えられる
これらステートマシンを構成する要素のことを State と呼ぶ

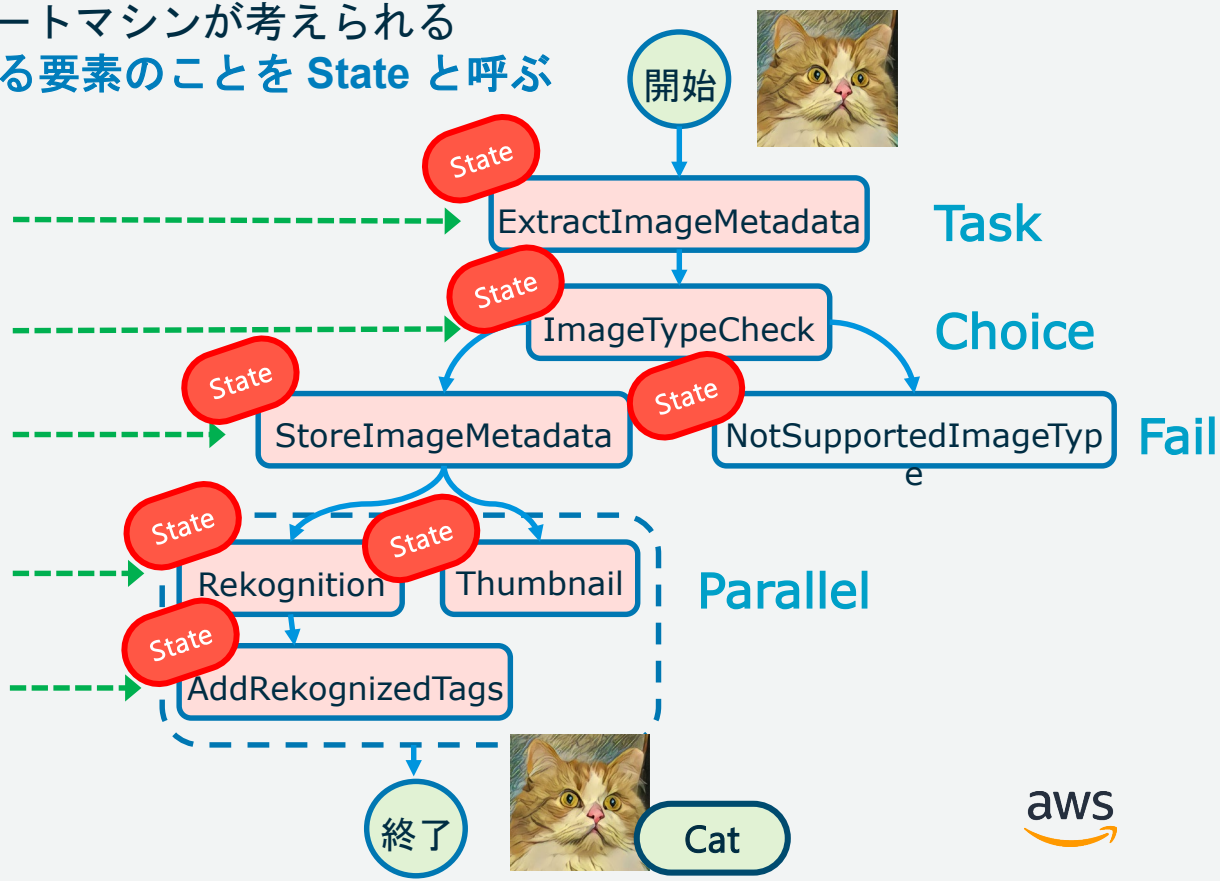
(1) メタデータの抽出

(2) 画像フォーマットの検証

(3) メタデータの保存 (不正な場合はエラー)

(4) 画像解析とサムネイル化

(5) タグの付与



ステートマシンの実行方法 (呼び出し元)

ステートマシンを呼び出す方法として下記が利用可能

- Amazon CloudWatch Events : S3 へのファイル保存や EC2 の起動といったイベントを契機に実行
- Amazon API Gateway : ある API が呼ばれた際のバックエンドとして実行
- マネジメント コンソール : コンソールから手動で実行
- AWS CLI : コマンドラインから実行
- 各種 SDK : Lambda や EC2 に実装したアプリケーションから実行

ステートマシンから呼び出し可能な AWS のサービス

- AWS Lambda : Lambda 関数の実行
- Amazon DynamoDB : 既存のアイテムの取得、新規アイテムの登録
- AWS Batch : ジョブの起動、ジョブ完了の待機
- Amazon Elastic Container Service : ECS/Fargate タスクの実行
- Amazon Simple Notification Service : SNS トピックへのメッセージ送信
- Amazon Simple Queue Service : SQS キューへのメッセージ送信
- AWS Glue : Glue ジョブの実行
- Amazon SageMaker : トレーニングジョブ、トランスフォームジョブの起動

これらに加えて、**Activity** (後述) と呼ばれる、自身で定義したサービスを紐付けることも可能

Activity

サーバーやコンテナ等に実装したアプリケーションからポーリングすることで、
独自定義の処理を実行する仕組み

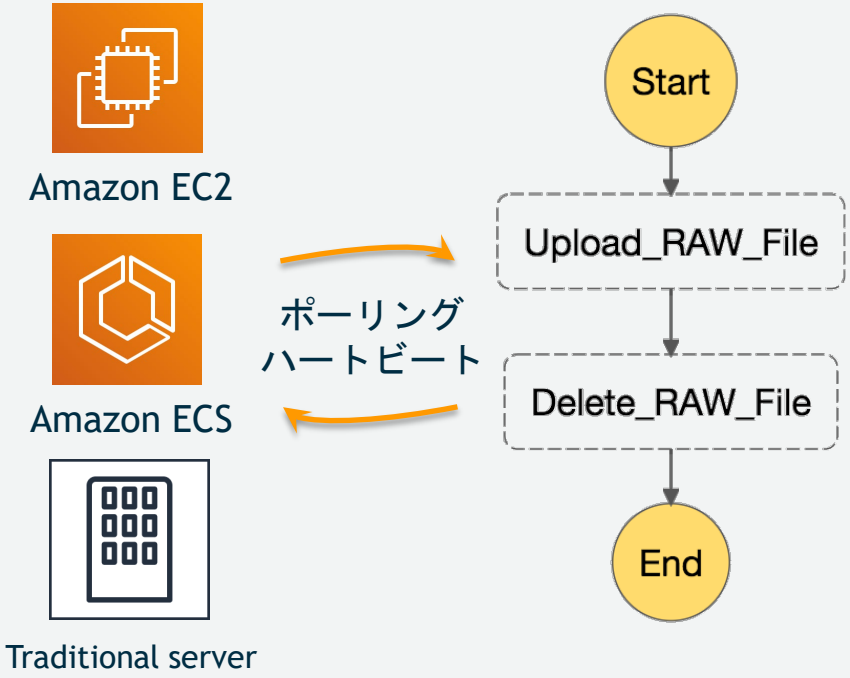
(1) 独自に実装したアプリケーションから
Step Functions を定期ポーリングする

(2) ステートマシンを実行する

(3) Activity の State に達するとポーリング
に対して応答が返却される

(4) アプリケーションで処理を実行し、
結果をステートマシンに通知する

（加えてデッドロック回避のため、アプリケーションから
ハートビートを送り、死活監視を行うことが可能）



本日のアジェンダ

- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

データの入力 (InputPath)

渡された入力のうち、何を State 内で使用するかを指定

入力データを指定
(最大 32,768 文字まで)

```
{
  "title": "Numbers to add",
  "numbers": [ 3, 4 ]
}
```

定義内の InputPath フィールドで State に引き渡す内容を指定。書式は JsonPath 構文 (※) を使う。

ASL

```
{
  "Type": "Task",
  "InputPath": "$.numbers",
  "Resource": "arn:aws:lambda..."
  ...
```

実際に値が State へ入力として引き渡される

```
[ 3, 4 ]
```

[3, 4]

(※) https://github.com/json-path/JsonPath

aws

データの入力 (Parameters)

渡された入力から、「キーと値」のペアを生成して State に渡す

入力データを指定
(最大 32,768 文字まで)

定義内の Parameters フィールドで State に引き渡す内容を指定。
InputPath と異なり JSON が構成できる。
キーと値の JSON 形式で State へ引き渡される

```
{
  "title": "Numbers to add",
  "numbers": [ 3, 4 ]
}
```

ASL

```
{
  "Type": "Task",
  "Parameters": {
    "calc.$": "$.numbers"
  },
  "Resource": "arn:aws:lambda..."
  ...
```

```
{
  "calc": [ 3, 4 ]
}
```

処理結果の受け取り (ResultPath)

State の実行結果をどのようなフィールド名で受け取るか指定

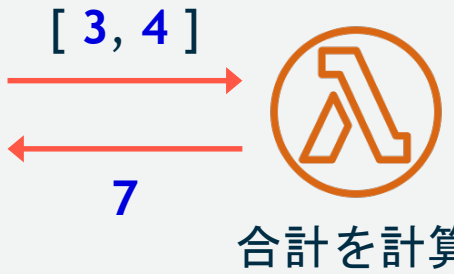
データを入力

```
{
  "title": "Numbers to add",
  "numbers": [ 3, 4 ]
}
```

定義内の ResultPath フィールドで結果を受け取るフィールド名を指定

ASL

```
{
  "Type": "Task",
  "InputPath": "$.numbers",
  "ResultPath": "$.sum",
  ...
```



State の出力に、入力データと、State の実行結果が含められる

```
{
  "title": "Numbers to add",
  "numbers": [ 3, 4 ],
  "sum": 7
}
```


aws

データの出力 (OutputPath)

次の State に対して引き渡す (出力する) データを指定

データを入力

```
{
  "title": "Numbers to add",
  "numbers": [ 3, 4 ]
}
```

定義内の OutputPath フィールドで次の State に引き渡す内容を指定。

```
{
  "Type": "Task",
  "InputPath": "$.numbers",
  "ResultPath": "$.sum",
 "OutputPath": "$['title', 'sum']"
}
```

ASL

[3, 4]

7

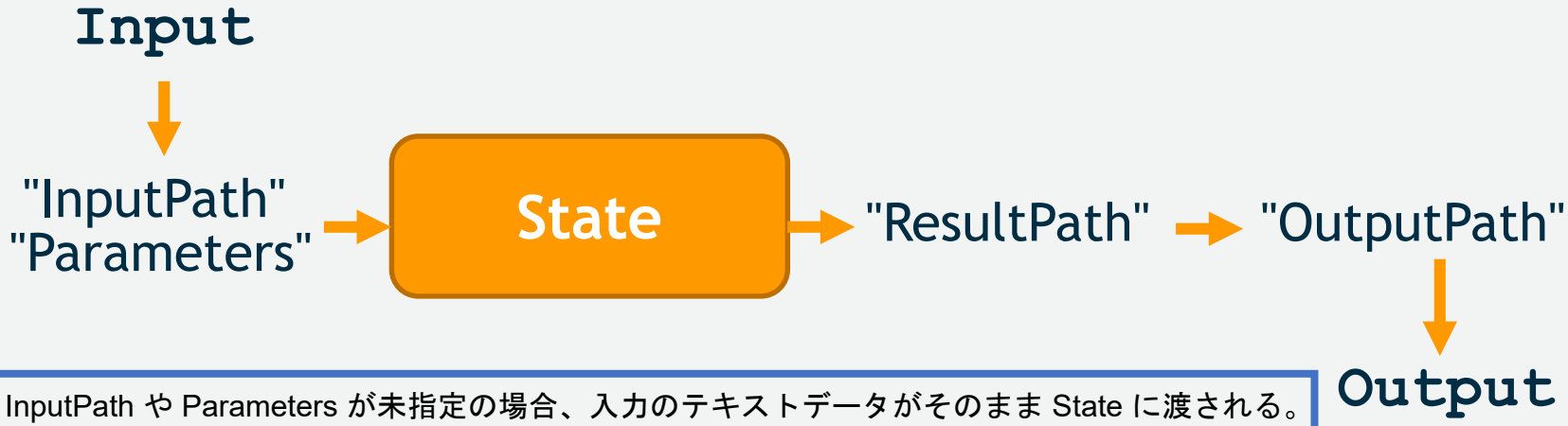
合計を計算

次の State に対して指定した値が引き渡される
(最大 32,768 文字まで)

```
{
  "title": "Numbers to add",
  "sum": 7
}
```


aws

データの入出力まとめ



- InputPath や Parameters が未指定の場合、入力のテキストデータがそのまま State に渡される。
- InputPath と Parameters が両方指定されていた場合、入力データに InputPath が適用され、適用後のデータにさらに Parameters が適用される。
- ResultPath が未指定の場合、入力のテキストデータは削除され State の実行結果が OutputPath に渡される。
- OutputPath が未指定の場合、ResultPath までを加味した結果がそのまま出力になる。
- 入力および出力は JSON 形式で行うのが一般的。

本日のアジェンダ

- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

ステートマシン全体の設定

```
{
  "Comment": "A Hello World example of the
Amazon States Language using a Pass state",
  "StartAt": "HelloWorld",
  "TimeoutSeconds": 600,
  "Version": "1.0",
  "States": {
    "HelloWorld": {
      "Type": "Pass",
      "Result": "Hello World!",
      "End": true
    }
  }
}
```

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-state-machine-structure.html

ステートマシンは ASL で定義するが、下記のようなフィールドを指定することができる。

- Comment : テキストで任意のコメントを入力できる。
- StartAt : 一番最初に実行する State を指定する。
- TimeoutSeconds : 秒数を指定する。ステートマシン全体の実行時間がこの秒数を超過するとタイムアウトエラーになり実行が終了する。
- Version : ASL のバージョンを指定。既定では 1.0。
- States : ステートマシンを構成する State を指定する。複数含めることが可能。

aws

7 種類の State Type

Task	単一の処理を行う
Wait	指定した時間の間、処理をストップする
Pass	入力をそのまま出力へ渡す
Parallel	並列に処理を実行する
Choice	一定の条件により分岐する
Fail	実行結果を失敗として終了する
Succeed	実行結果を成功として終了する

State Type : Task

```
{
  "StartAt": "Put Message",
  "TimeoutSeconds": 3600,
  "States": {
    "Put Message": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:putItem",
      "Parameters": {
        "TableName": "StepFunctionsSample",
        "Item": {
          "MessageId": {
            "S": "this is a test message"
          }
        }
      },
      "End": true
    }
  }
}
```

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-task-state.html

最も基本的な State Type。ステートマシンによって実行される単一の作業単位を表す。Lambda や DynamoDB といったリソースに対して処理の実行を指示する。
下記のフィールドが存在。

- Resource : Lambda や DynamoDB といったサービスに対して実行する処理を ARN 形式で指定
- Parameters : Resource で指定したリソースに渡すパラメータを指定。Lambda や DynamoDB、SNS などサービスにより内容は異なる。
- TimeoutSeconds : 秒数を指定し、実行時間がこの秒数を超えるとタイムアウトエラーになる。
- HeartbeatSeconds : 秒数を指定し、Activity にのみ適用される。Activity が Step Functions の SendTaskHeartbeat API を、ここで指定した秒数以内に定期実行しないと、タイムアウトエラーになる。
- ResultPath : State の出力をどのような名称で受け取るか指定。
- Retry : エラーのハンドリングに使用 (後述)
- Catch : エラーのハンドリングに使用 (後述)

State Type : Wait

```
{
  "StartAt": "Wait State",
  "States": {
    "Wait State": {
      "Type": "Wait",
      "Seconds": 20,
      "Next": "Next State"
    },
    "Next State": {
      "Type": "Task",
      "Resource": "arn:aws:states:::dynamodb:putItem",
      "Parameters": {
        "TableName": "StepFunctionsSample",
        "Item": {
          "MessageId": {
            "S": "this is a test message"
          }
        }
      },
      "End": true
    }
  }
}
```

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-wait-state.html

指定された時間だけ待機して、次の State に
状態遷移させる場合に利用する State Type。
次のうちいずれかを指定する。

- Seconds : 待機する時間を秒単位で指定
- Timestamp : ISO8601 形式で、いつまで待機するかを指定 (例 : 2019-08-18T17:33:00Z)
- SecondsPath : State への入力データのうち、待機する時間 (秒単位) を JsonPath 構文で指定
- TimestampPath : State への入力データのうち、いつまで待機するか、時間を JsonPath 構文で指定

aws

State Type : Pass

```
{
  "StartAt": "Show Axis",
  "States": {
    "Show Axis": {
      "Type": "Pass",
      "Result": {
        "x-axis": 10,
        "y-axis": 20
      },
      "ResultPath": "$.axis",
      "End": true
    }
  }
}
```

何も作業をせずに入力を次の出力に渡す State Type。ステートマシンをデバッグする際にも利用できる。
次のフィールドを利用できる。

- Result : State の実行結果を指定。
- ResultPath : Result をどのようなパラメータ名で受け取るか指定。
- Parameters : 前述のとおり、入力されたデータから「キーと値」の JSON を生成。

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-pass-state.html

aws

State Type : Parallel

```
{
  "Comment": "Parallel Example.",
  "StartAt": "LookupCustomerInfo",
  "States": {
    "LookupCustomerInfo": {
      "Type": "Parallel",
      "End": true,
      "Branches": [{
        "StartAt": "LookupAddress",
        "States": {
          "LookupAddress": {
            "Type": "Task",
            "Resource": "(..省略..)",
            "End": true
          }
        }
      },
      {
        "StartAt": "LookupPhone",
        "States": {
          "LookupPhone": {
            "Type": "Task",
            "Resource": "(..省略..)",
            "End": true
          }
        }
      }]
    }
  }
}
```

並列に State を実行するときに使用する State Type。並列実行される State のうち、いずれかが失敗すると、Parallel の State 全体が失敗扱いになる点に注意。

- Branches : 配列で、並列に実行する State を指定
- ResultPath : Parallel の State の出力をどのようなパラメータで受け取るか指定。
- Retry : Branches 内の State のエラーのハンドリングに使用 (後述)
- Catch : Branches 内の State のエラーのハンドリングに使用 (後述)

Parallel の State の出力は配列形式で生成される。
たとえば、入力が **[3, 2]** に対して、加算の State と減算の State を並列に実行した場合、出力は **[5, 1]** といった結果が得られる。

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-parallel-state.html

State Type : Choice

```
{
  "StartAt": "Choice State",
  "States": {
    "Choice State": {
      "Type" : "Choice",
      "Choices": [
        {
          "Variable": "$.choice",
          "NumericEquals": 1,
          "Next": "Succeed State"
        }
      ],
      "Default": "Fail State"
    },
    "Succeed State": {
      "Type" : "Succeed"
    },
    "Fail State": {
      "Type": "Fail",
      "Error": "DefaultStateError",
      "Cause": "No Matches!"
    }
  }
}
```

条件分岐のロジックを記載するときに使用する State Type。

- Default : どの条件にも合致しなかったときに実行する State を指定
- Choices : 配列形式で条件分岐の内容を指定する
  - Variable : State への入力のうち指定したフィールドを条件分岐に利用する
  - Next : 条件に合致したときに遷移する State を指定
  - NumericEquals : Variable に対する比較演算子と、比較する値を指定。
    その他にも以下の比較演算子が利用可能。

- And
- BooleanEquals
- Not
- NumericEquals
- NumericGreaterThan
- NumericGreaterThanEquals
- NumericLessThan
- NumericLessThanEquals
- Or
- StringEquals
- StringGreaterThan
- StringGreaterThanEquals
- StringLessThan
- StringLessThanEquals
- TimestampEquals
- TimestampGreaterThan
- TimestampGreaterThanEquals
- TimestampLessThan
- TimestampLessThanEquals

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-choice-state.html

State Type : Succeed

```
{
  "StartAt": "Choice State",
  "States": {
    "Choice State": {
      "Type" : "Choice",
      "Choices": [
        {
          "Variable": "$.choice",
          "NumericEquals": 1,
          "Next": "Succeed State"
        }
      ],
      "Default": "Fail State"
    },
    "Succeed State": {
      "Type" : "Succeed"
    },
    "Fail State": {
      "Type": "Fail",
      "Error": "DefaultStateError",
      "Cause": "No Matches!"
    }
  }
}
```

ステートマシン実行を正常停止させる State Type。
主に Choice State Type の遷移先に使われる。
特に追加のフィールドはなく、停止させるための State Type であるため、Next や End フィールドは不要。

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-succeed-state.html

aws

State Type : Fail

```
{
  "StartAt": "Choice State",
  "States": {
    "Choice State": {
      "Type" : "Choice",
      "Choices": [
        {
          "Variable": "$.choice",
          "NumericEquals": 1,
          "Next": "Succeed State"
        }
      ],
      "Default": "Fail State"
    },
    "Succeed State": {
      "Type" : "Succeed"
    },
    "Fail State": {
      "Type": "Fail",
      "Error": "DefaultStateError",
      "Cause": "No Matches!"
    }
  }
}
```

ステートマシン実行を失敗としてマークさせる State Type。停止させるための State Type であるため、Next や End フィールドは不要。
次のフィールドを使用できる。

- Error : エラー名を指定。Retry や例外の Catch に使用できる。
- Cause : エラーの理由を文字列で指定

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/amazon-states-language-fail-state.html

aws

State の Retry

```
{
  "StartAt": "Parallel",
  "States": {
    "Parallel": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "FailState",
          "States": {
            "FailState": {
              "Type": "Fail"
            }
          }
        }
      ],
      "Retry":  [
        {
          "ErrorEquals": ["States.ALL"],
          "IntervalSeconds": 3,
          "BackoffRate": 2.0,
          "MaxAttempts": 4
        }
      ]
    }
  }
}
```

State Type が **Task** もしくは **Parallel** の場合、Retry というフィールドを定義できる。State でエラーが発生した際に再試行させる際に利用する。

- ErrorEquals : リトライの対象とするエラーを指定 (後述)
- IntervalSeconds : 最初のリトライまでの秒数。
- BackoffRate : リトライ間隔を増加させる乗数。左記の設定の場合、1 回目のリトライは **3 秒後**に行われ、再び失敗する。2 回目のリトライは、**1 回目のリトライからさらに 3 x 2.0 = 6 秒後**に行われる。3 回目のリトライは、**2 回目のリトライから 3 x 2.0 x 2.0 = 12 秒後**、4 回目のリトライは、3 回目のリトライから **3 x 2.0 x 2.0 x 2.0 = 24 秒後**に行われる。
- MaxAttempts : リトライを行う回数。ここで指定した回数リトライが失敗すると、State の実行が失敗となる。

State のエラー処理 (Catch)

```
{
  "StartAt": "Parallel",
  "States": {
    "Parallel": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "FailState",
          "States": {
            "FailState": {
              "Type": "Fail",
              "Error": "An Error Occurred",
              "Cause": "Unknown"
            }
          }
        }
      ],
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "Next": "Fallback",
        "ResultPath": "$.error"
      }]
    },
    "Fallback": {
      "Type": "Pass",
      "End": true
    }
  }
}
```

State Type が **Task** もしくは **Parallel** の場合、Catch というフィールドを定義できる。

- ErrorEquals : エラーの条件を指定。
- Next : エラー条件に該当した際に遷移させる State を指定。
- ResultPath : エラーの内容を出力に含める際に指定する。左記の場合、以下のような出力が得られる。

▼ 出力

```
{
  "error": {
    "Error": "An Error Occurred",
    "Cause": "Unknown"
  }
}
```


aws

ErrorEquals によるエラー判定

Retry や Catch フィールドでは、どのようなエラーのときに条件に合致したとみなすか、ErrorEquals フィールドで指定する

```
{
  "StartAt": "Parallel",
  "States": {
    "Parallel": {
      "Type": "Parallel",
      "End": true,
      ..(省略)..
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "Next": "Fallback",
        "ResultPath": "$.error"
      }]
    },
    "Fallback": {
      "Type": "Pass",
      "End": true
    }
  }
}
```

- States.ALL : すべてのエラーに合致する
- States.Timeout : State がタイムアウトした際に合致する
- States.TaskFailed : State 実行に失敗した際に合致する
- States.Permissions : State 実行の権限が無かった際に合致する

ErrorEquals によるエラー判定 (独自実装の例外)

State から Lambda 関数を実行する場合、関数で実装した例外を条件とすることも可能

```
{
  "StartAt": "CreateAccount",
  "States": {
    "CreateAccount": {
      "Type": "Task",
      "Resource": "...(省略)...",
      "Next": "NextState",
      "Catch": [{
          "ErrorEquals": ["AccountAlreadyExistsException"],
          "Next": "FallbackState"
        }]
    },
    ...(省略)...
  }
}
```

AccountAlreadyExistsException の例外が発生した場合を捕捉 (Catch)

Python

```
class AccountAlreadyExistsException(Exception): pass

def create_account(event, context):
    raise AccountAlreadyExistsException('Account is in use!')
```

Java

```
package com.example;

public static class AccountAlreadyExistsException extends Exception {
    public AccountAlreadyExistsException(String message) {
        super(message);
    }
}

package com.example;

import com.amazonaws.services.lambda.runtime.Context;

public class Handler {
    public static void CreateAccount(String name, Context context) throws AccountAlreadyExistsException {
        throw new AccountAlreadyExistsException ("Account is in use!");
    }
}
```

その他、C# や Node.js など、例外の表現については、言語ごとの Lambda 関数の記述方式に従う。

aws

本日のアジェンダ

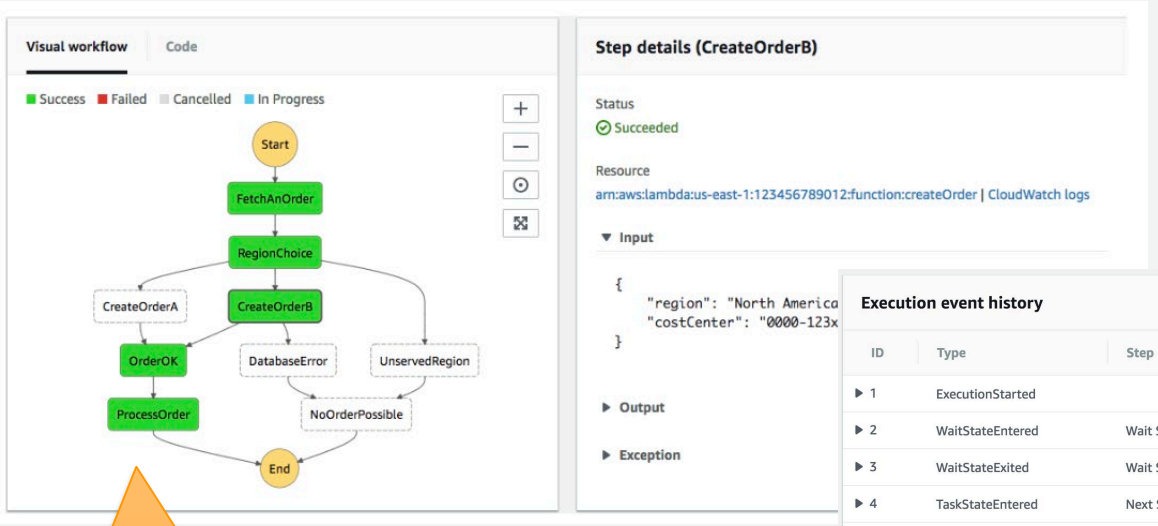
- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

実行の状況の確認と中断

ステートマシンの実行には任意の名称を付与することができる
実行ごとに状況の確認や、途中で中断させることができる

ステートマシンの実行結果

State ごとにどのような実行結果であったのか、および全体の実行履歴を確認することが可能。



緑色になった State は、
実行され成功したものを表す

Execution event history

ID	Type	Step	Resource	Elapsed Time (ms)	Timestamp
▶ 1	ExecutionStarted		-	0	May 19, 2019 04:04:59.450 AM
▶ 2	WaitStateEntered	Wait State	-	45	May 19, 2019 04:04:59.495 AM
▶ 3	WaitStateExited	Wait State	-	50048	May 19, 2019 04:05:49.498 AM
▶ 4	TaskStateEntered	Next State	-	50063	May 19, 2019 04:05:49.513 AM
▶ 5	TaskScheduled	Next State	-	50063	May 19, 2019 04:05:49.513 AM
▶ 6	TaskStarted	Next State	-	50120	May 19, 2019 04:05:49.570 AM
▶ 7	TaskSucceeded	Next State	-	50206	May 19, 2019 04:05:49.656 AM
▶ 8	TaskStateExited	Next State	-	50206	May 19, 2019 04:05:49.656 AM
▶ 9	ExecutionSucceeded		-	50206	May 19, 2019 04:05:49.656 AM

ステートマシンの実行通知

ステートマシンの実行ステータスが変化した際に、CloudWatch Events に対してイベントを発行することができる

サービス名：Step Functions
イベントタイプ：Step Functions Execution Status Change
を選択。
その他に、タイムアウトなど、どのステータスのときにイベントを発行するか、またどのステートマシンを対象にするかを選択できる。

パフォーマンスの監視

Amazon CloudWatch より以下の 7 種類のメトリクスが監視できる
定義したステートマシンごとに確認することが可能

ExecutionTime	実行の開始時点から終了時点までの間隔 (ミリ秒単位)。
ExecutionThrottled	制限 (後述) に該当しスロットリングされた実行の数。
ExecutionsAborted	中断された実行の数。
ExecutionsFailed	失敗した実行の数。
ExecutionsStarted	開始された実行の数。
ExecutionsSucceeded	正常に完了した実行の数。
ExecutionsTimedOut	タイムアウトした実行の数。

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/procedure-cw-metrics.html

aws

本日のアジェンダ

- ❑ Step Functions 概要
- ❑ ステートマシン
- ❑ データの入出力
- ❑ State の記述
- ❑ 実行状況の確認
- ❑ 補足や料金詳細など

AWS Step Functions が利用可能なリージョン

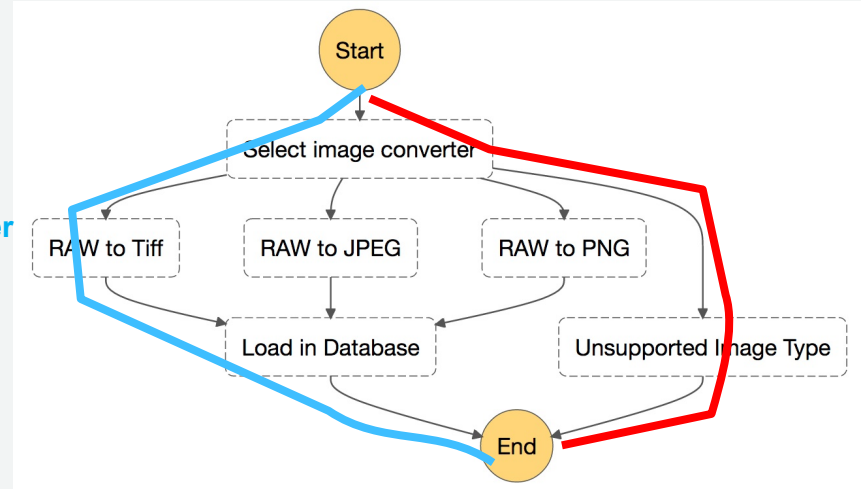
AWS Step Functions は 21 のリージョン全てでご利用可能

(2019/05/22現在、大阪ローカルリージョンを除く)

料金 (2019/05/22 時点)

- 状態遷移 1,000 回あたり $0.025 (Tokyo Region)
  - 状態遷移 1 回あたり $0.000025
  - 毎月 4,000 回までの状態遷移は無料
  - 価格はリージョンごとに異なる
  - リトライは追加の状態遷移として計算

状態遷移=4回
- **Select image converter**
- **RAW to Tiff**
- **Load in Database**
- **End**



状態遷移=3回
- **Select image converter**
- **Unsupported Image Type**
- **End**

(例) 1 ヶ月の状態遷移の合計数: 3 回の状態遷移 x 10 万回の実行 = **30 万回**の状態遷移
1 ヶ月の料金: (300,000 – 4,000 回の無料枠) x 状態遷移 1 回あたり 0.000025 USD = **7.40 USD**

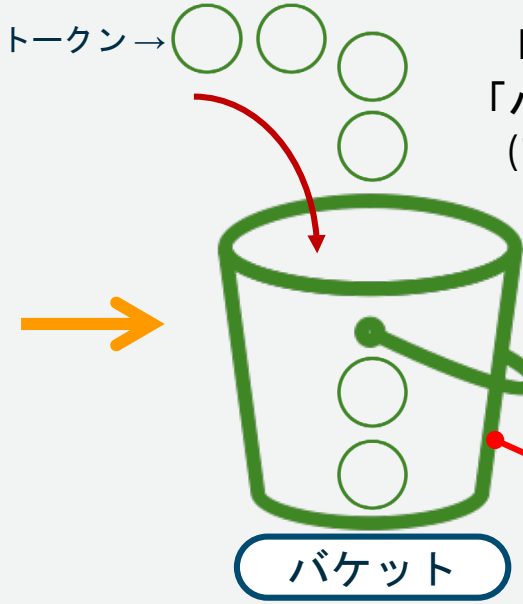
AWS Step Functions の主な制限

- SLA : 99.9 %
- ステートマシンの最大数 : 10,000 (1 アカウントあたり)
- Activity の最大数 : 10,000 (1 アカウントあたり)
- 最大リクエストサイズ : 1 MB (1 API リクエストあたり)
  - API リクエストに含めることが可能な合計データのサイズ
- ステートマシンの最大実行時間 : 1 年 (1 実行あたり)
- ステートマシンの実行履歴の最大サイズ : 25,000 イベント (1 実行あたり)
  - 実行履歴の上限に達するとステートマシンの実行は失敗する。
- State への入出力の最大データサイズ : 32,768 文字
- State 遷移の合計数 : 1 アカウント・1 リージョン内のすべてのステートマシンに対して State 遷移の合計数が計算され、**トークンバケットアルゴリズム (後述)** にもとづいた下記の制限が適用される。
  - バケットサイズ : 800 (Tokyo リージョンの場合)
  - 1 秒ごとの補充レート : 400 (Tokyo リージョンの場合)

https://docs.aws.amazon.com/ja_jp/step-functions/latest/dg/limits.html

スロットリング - トークンバケットアルゴリズム

一部のスロットリングはトークンバケットアルゴリズムに基づく
「レート」と「バースト」の2つの設定値に従って行われる



「レート」=トークンの補充スピード=
「バケットに追加されるトークン数/秒」
(アカウントレベルの初期値 … 400 /秒)

ステートマシンで
State 遷移が発生

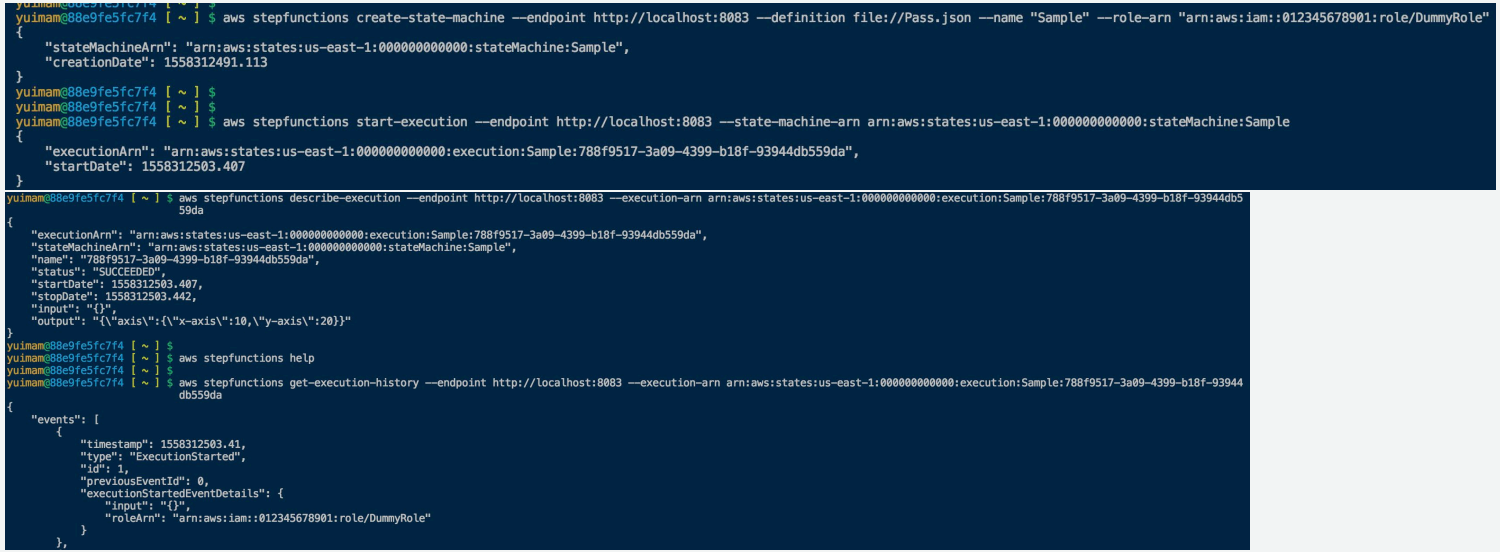
1つの State 遷移は
1つのトークンを消費

トークンを使い果たすと State 遷移が制限(スロットリング)され失敗する

「バースト(バースト上限)」=バケット内のトークンの初期値 兼 バケット最大サイズ
(アカウントレベルの初期値 … 800)

aws

AWS Step Functions Local



- 開発者のローカル環境でテストするためのツールで、ローカル環境に Step Functions のテスト用ランタイムが起動する。
- JAR パッケージもしくは Docker コンテナとして利用可能で、AWS SAM Local といったその他のローカル環境向けテストツールとも連携できる。
- AWS CLI のオプションでエンドポイントを指定できるので、起動した Step Functions Local のサーバーを指定することで、コマンドラインから Step Functions の動作をローカル環境で検証できる。

その他類似サービスとの使い分け

- Amazon Simple Queue Service (SQS)
  - マネージドなメッセージ キューイングサービス
  - サービス間のメッセージの管理に、スケーラブルで信頼性が高いキューが必要な場合は SQS
  - 処理の追跡や、サービス間のメッセージの受け渡しなど、アプリケーション開発に役立つ機能をマネージドに利用したい場合は Step Functions
- Amazon Simple Workflow Service (SWF)
  - Decider と呼ばれる、プログラミングベースでワークフロー制御を行うサービス
  - Java もしくは Ruby の AWS Flow Framework と併用するのが一般的
  - Step Functions と比較して複雑化するため、新規開発では Step Functions を使うことを推奨

https://aws.amazon.com/jp/step-functions/faqs/

ユースケース

相互に依存するような複数の処理を組み合わせたい場合に最適

- データプロセシング：複数のデータストアを利用する分析や機械学習
- eコマース：在庫追跡や注文処理
- 動画処理：サムネイルの生成、ビデオのエンコーディング
- バッチ処理：ゲノム情報解析のような学術領域
- ウェブアプリケーション：複雑なユーザー登録プロセス
- 管理者承認：管理者が承認した場合に限り処理を行う

…etc

まとめ：AWS Step Functions のメリット

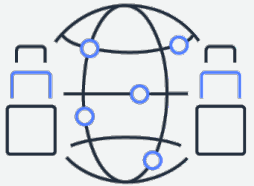
アプリケーションをすぐに構築可能

アプリケーションを数分で構築し、アプリケーションが意図したとおりに動作するように各ステップの実行を可視化および追跡できる

コーディング不要

アプリケーションを構成する関数全体を管理するための余分なコーディングが不要になる

耐久性の向上



各ステップの状態を追跡し、リトライやロールバックによりエラーに対応できる

ご視聴ありがとうございました

AWS 公式 Webinar
https://amzn.to/JPWebinar

過去資料
https://amzn.to/JPArchive