aws DEV DAY
ONLINE JAPAN

20-22.10.2020

DEV DAY

In Partnership with intel

aws DEV DAY
ONLINE JAPAN

B - 4

分散システムにおけるSagaパターンの AWS Step Functions による実装

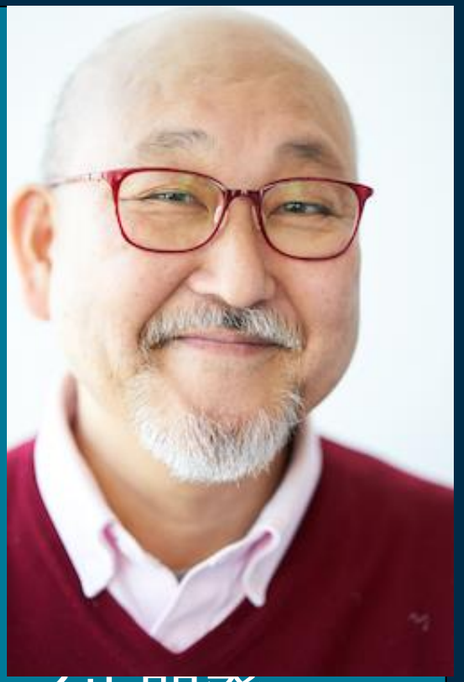
福井 厚 Atsushi Fukui

シニアソリューションアーキテクト
サーバーレススペシャリスト
アマゾンウェブサービスジャパン株式会社

In Partnership with intel

自己紹介

- **名前**
  - 福井 厚（ふくい あつし）Twitter: @afukui
- **所属**
  - アマゾン ウェブ サービス ジャパン株式会社
  - 技術統括本部レディネスソリューション本部
  - シニアソリューションアーキテクト
サーバーレス スペシャリスト
- **関心領域**
  - ソフトウェア アーキテクチャ、オブジェクト指向設計、アジャイル開発
- **好きなAWSサービス**
  - サーバーレステクノロジー全般、 AWS Code シリーズ、AWS Amplify



Related breakouts

10/20

A-1：今日から始める、サーバーレス Well-Architected Framework

11:45-12:25　Track A

10/20

A-2：DX 時代における最適な開発手法サーバレスと DB 選択の勘所

12:35-13:15　Track A

10/20

C-7 :AWS AppSync Advanced Design Pattern

16:15-16:45 Track C

In Partnership with intel

本セッションでお伝えしたいこと

- Sagaパターンについて理解する
- オーケストレーションについて理解する
- AWS Step Functionsとその活用について理解する

本セッションで扱わないこと

- AWS Step Functionsの詳細な機能解説
  こちらについては AWS Black Belt Webinar を参照ください。
  https://aws.amazon.com/jp/blogs/news/webinar-bb-aws-step-functions-2019/

Agenda

Saga パターンとは

AWS Step Functionsとは

AWS Step Functions を利用したSagaパターンの実装例

まとめ

aws DEV DAY
ONLINE JAPAN

Sagaパターンとは

In Partnership with intel

Sagaパターンとは

- Hector Garcia-MolinaとKenneth Salemが1987年に ”長期生存トランザクション(long-lived transactions: LLT)"を扱うデザインパターンとしてSAGASというタイトルで論文を発表
  https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

- “LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions.” (Garcia-Molina, Salem 1987)
  ”LLTは、他のトランザクションと交互に配置できる一連のトランザクションとして記述できる場合、それはsagaである。”

  ” Due to space limitations, we only discuss sagas in a centralized system, although clearly they can be implemented in a distributed database system.”
  1987年時点で、現在のマイクロサービスアーキテクチャを見越したような、分散データベースにも実装できると述べているのがスゴい。

Saga：LLTの課題とポイント

- ほとんどの場合、LLTは深刻なパフォーマンスの問題を引き起こす
- トランザクションであるため、システムはそれらをアトミックアクションとして実行する必要がある
- LLTのオブジェクトにアクセスすることを望む他のトランザクションは、長いロック遅延を被る
- LLTは多くのオブジェクトにアクセスするため、多くのデッドロックが発生し、それに応じて多くのアボートが発生する可能性がある

- 但し、 LLTをアトミックアクションとして実行するという要件を緩和することで、問題を軽減できる場合がある

Sagaパターン

- ”Saga内のトランザクションは相互に関連しており、（非アトミックな）ユニットとして実行する必要がある。Sagaの部分的な実行は望ましくなく、エラーが発生した場合は補償する必要がある。”

- 部分的な実行を修正するには、各SagaトランザクションT(i)に補償トランザクションC(i)を提供する必要がある。 補償トランザクションは、セマンティックの観点から、T(i)によって実行されたアクションをすべて元に戻すが、(他のトランザクションによって変更されている可能性があるため)データベースをT(i)の実行が開始されたときに存在していた状態に戻すとは限らない。

T トランザクション

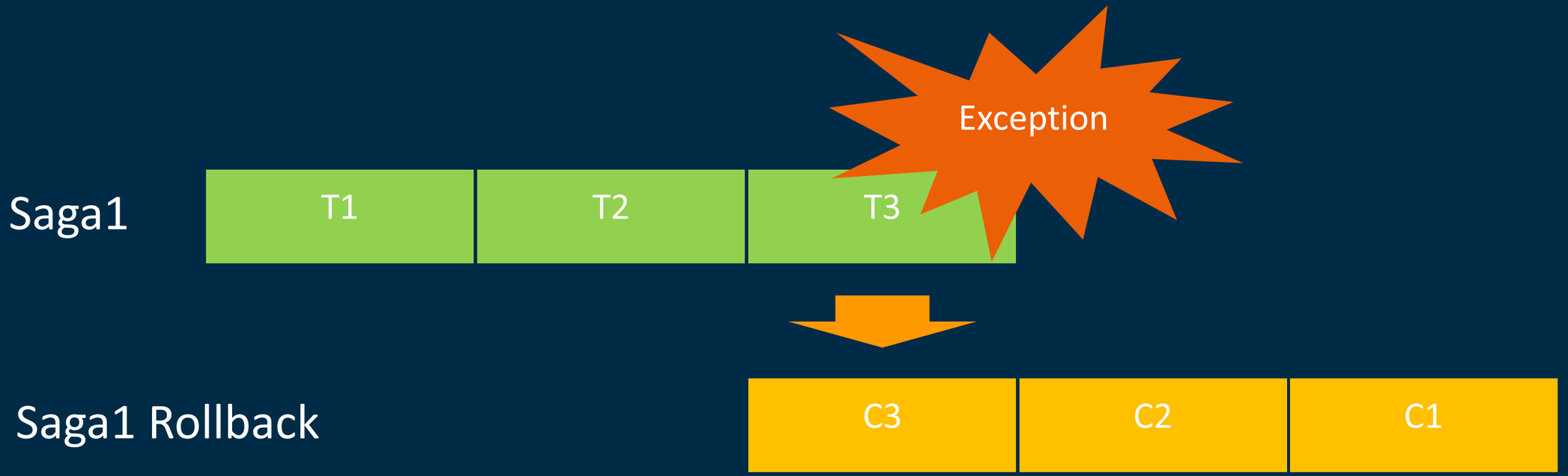
In Partnership with intel

Sagaの実装

- アプリケーションプログラマの観点からは、Sagaの開始と終了、各トランザクションの開始と終了、および補償トランザクションをシステムに知らせるためのメカニズムが必要

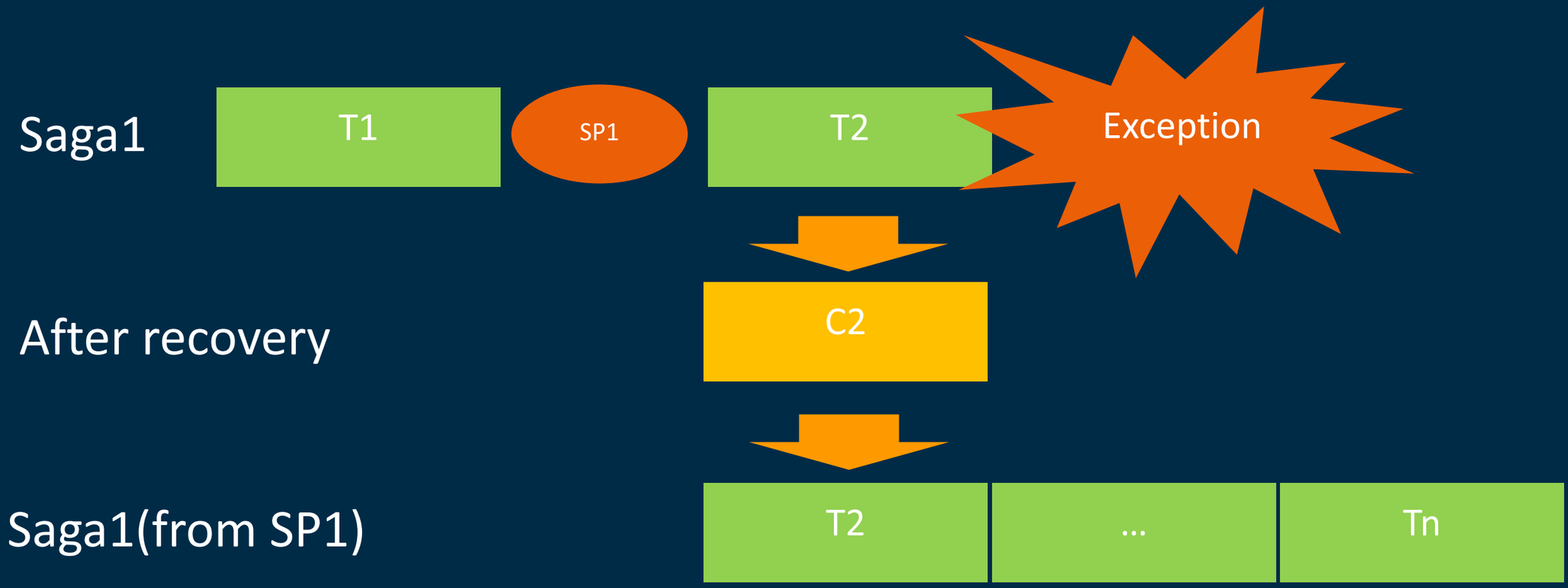
- SagaトランザクションIDや補償トランザクションの名前、エントリポイント、補償トランザクションに必要なパラメータ、セーブポイントなどを保持する必要がある

Sagaの実装(バックワードリカバリ)



In Partnership with intel

Sagaの実装 (フォワードリカバリとSave Point)

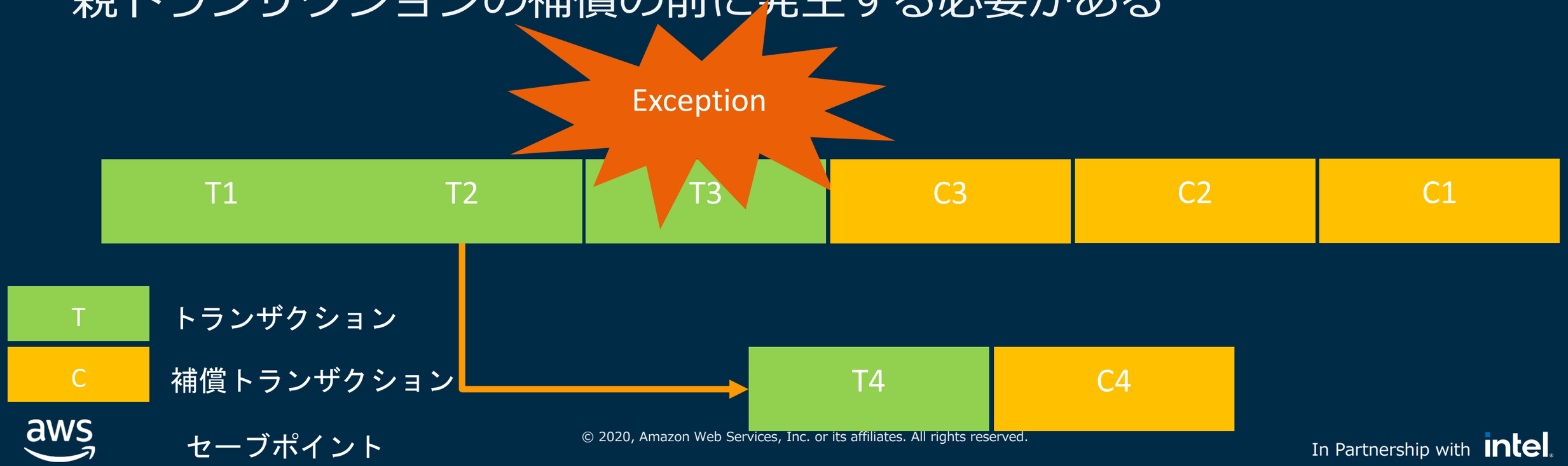


注意：常に可能であるとは限らない

In Partnership with intel

Sagaと並列トランザクション

- Sagaは並列トランザクションを含むように拡張することが可能
- 並列Sagaの各プロセス内で、トランザクションはシーケンシャルなSagaと同様に逆の順序で補償
- 子プロセスのすべての補償は、子プロセスの作成前に実行された親トランザクションの補償の前に発生する必要がある



In Partnership with intel

マイクロサービスアーキテクチャと共に再び注目

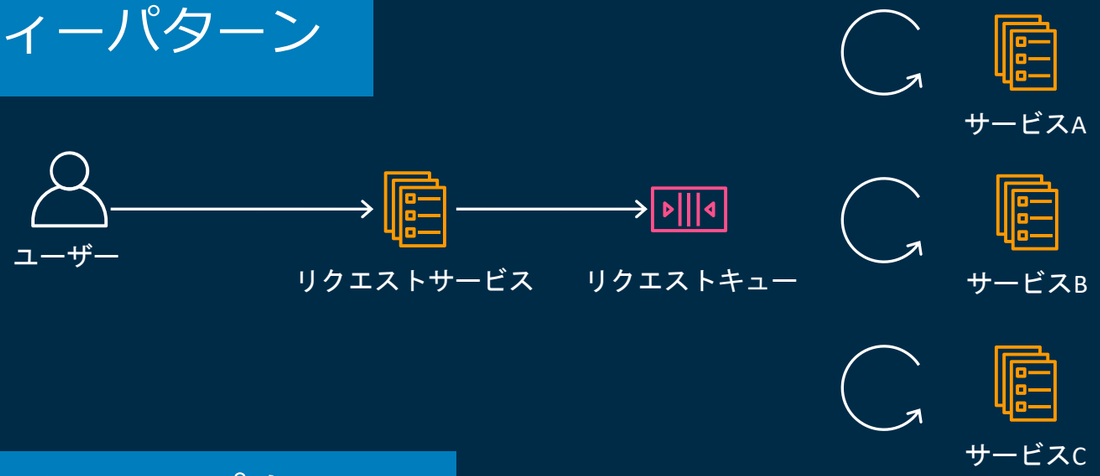
- Chris RichardsonがMicroservices PatternsでSagaパターンを取り上げたことで再び注目を浴びる(*)
- Pattern: Saga (**)
- マイクロサービスで典型的な Database per Seviceパターンを適用している場合、個々のマイクロサービス内ではACIDトランザクションを利用出来るが、サービスをまたがるトランザクションを管理する仕組みが必要 -> Sagaパターン
- 実装例としてコレオグラフィベースのSagaとオーケストレーションベースのSagaを紹介

(*) https://microservices.io/index.html

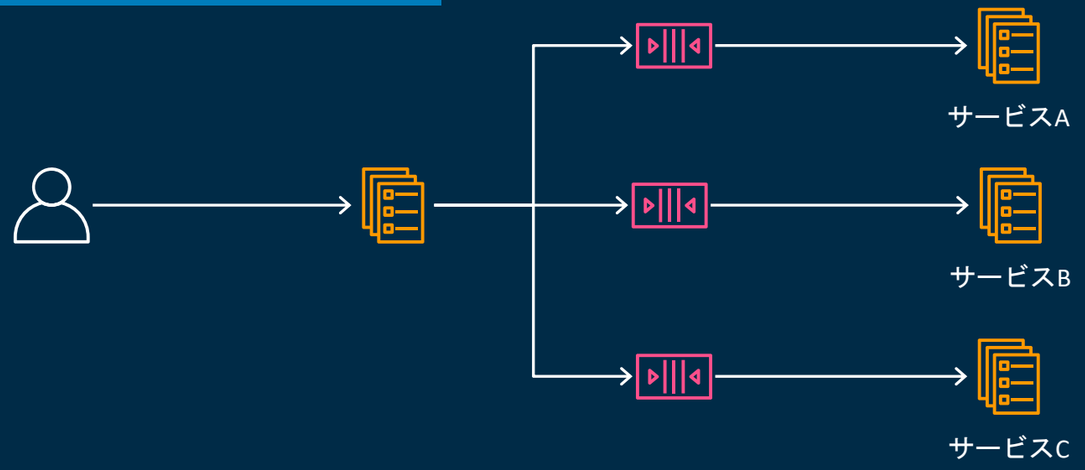
(**) https://microservices.io/patterns/data/saga.html

コレオグラフィーとオーケストレーション

コレオグラフィーパターン

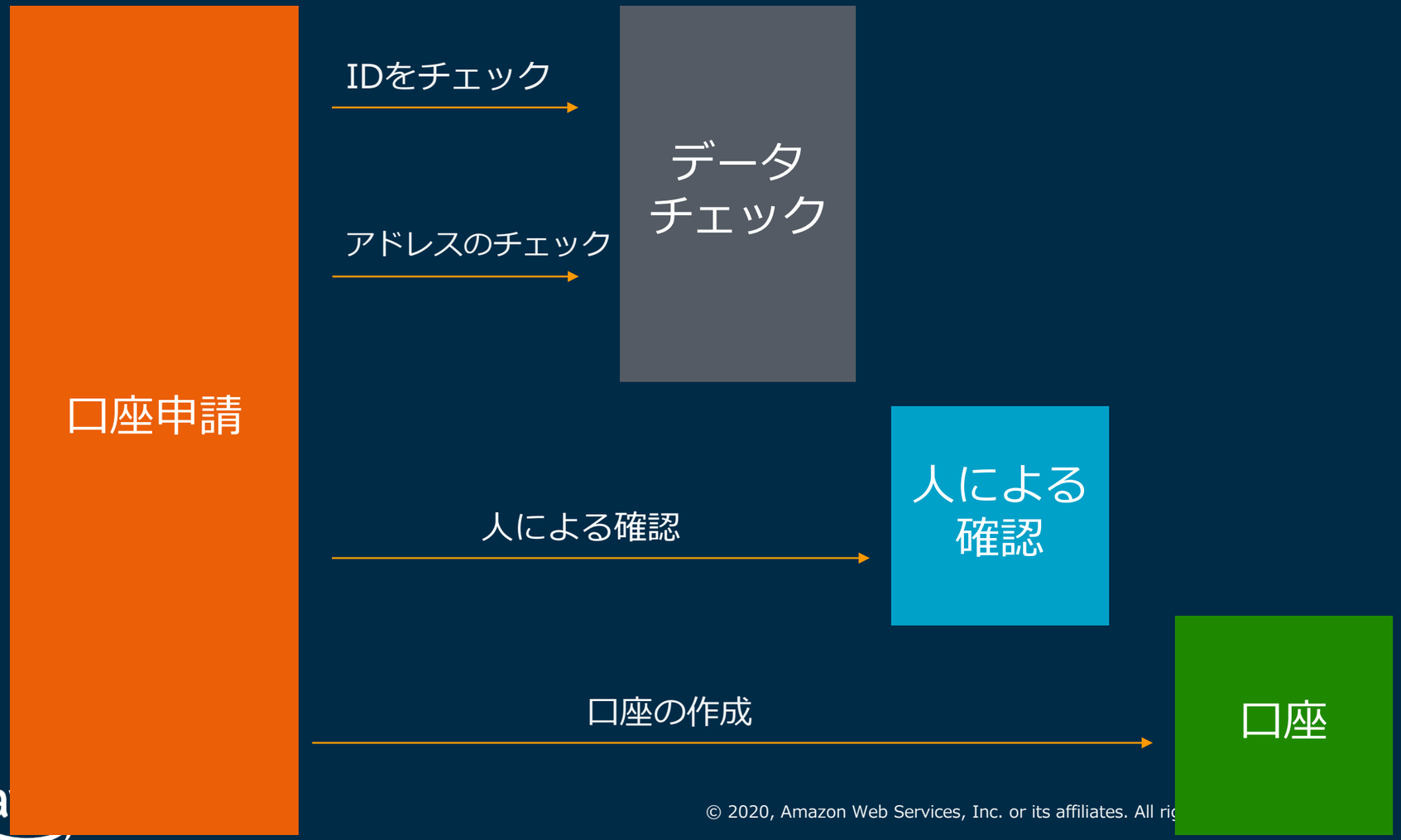


オーケストレーションパターン



In Partnership with intel

オーケストレーション: 1つのプロセスがワークフローの状態を管理し、適切なサービスを順番に呼び出す



In Partnership with intel

オーケストレーション: 1つのプロセスがワークフローの状態を管理し、適切なサービスを順番に呼び出す

In Partnership with intel

aws DEV DAY
ONLINE JAPAN

AWS Step Functionsとは

In Partnership with intel

AWS Step Functions

AWSのフルマネージドなステートマシン

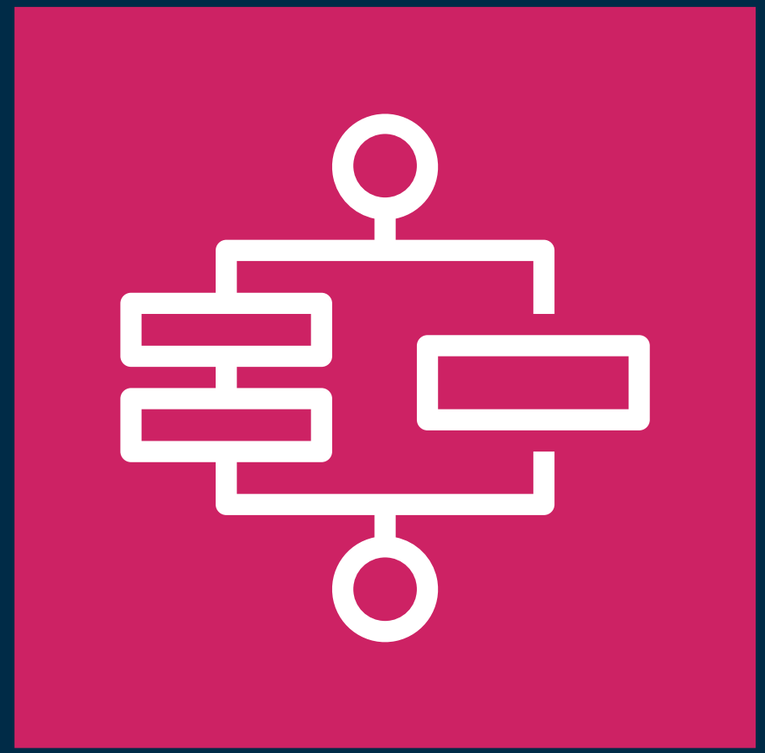
弾力性のあるワークフローオートメーション

組み込みのエラーハンドリング

AWSサービスとの強力な統合

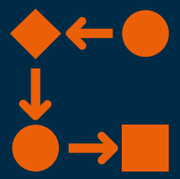
独自のサービスとの統合サポート

実行履歴の監査とビジュアルモニタリング



In Partnership with intel

AWS Step Functionsはどのように動作するか



Step Functionsで構築したワークフローはステートマシンと呼ばれ、ワークフローの個々のステップはステート(状態)と呼ばれる

ステートマシンを実行する時、一つのステートから次のステートへ移動することを状態遷移と呼ぶ

コンポーネントの再利用が可能で、ステートを容易に編集し、要件の変更に応じてtaskステートによってコードを交換することが可能

ステートマシンとは

個別のステート(状態)に分割された各ステップのコレクションを表す

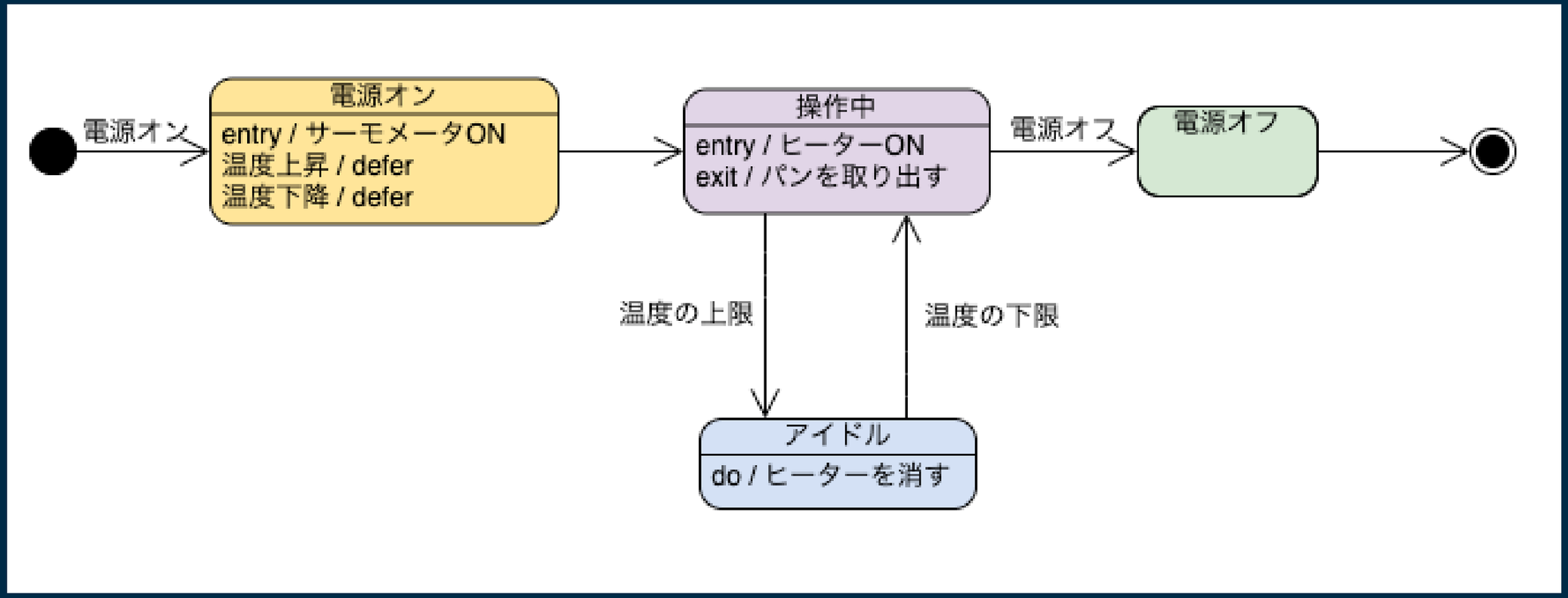
1つの開始状態を持ち、常に1つのアクティブ状態(実行中期間)を持つ

アクティブ状態は入力を受け取り、何らかのアクションを実行し、アウトプットを生成する

ステート間の遷移は前のステートの出力と定義されたルールに基づく

In Partnership with intel

UML State Machine Diagramの例



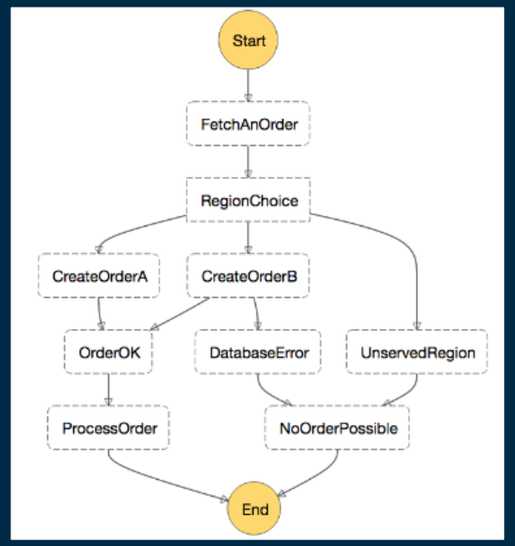
In Partnership with intel

AWS Step Functions

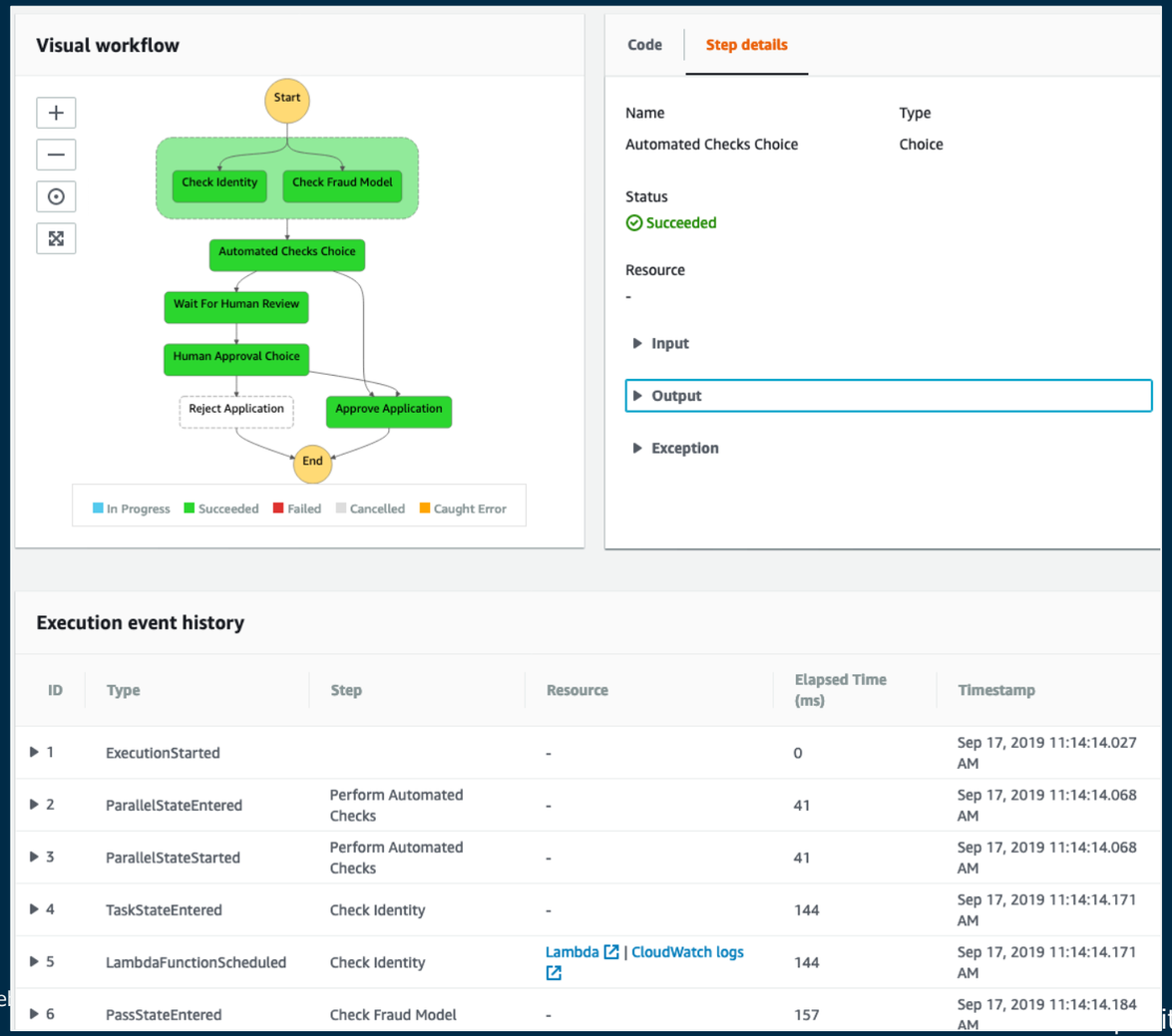
JSONで定義(Amazon States Language)

```
{
    "Comment": "Manage opening an account",
    "StartAt": "Perform Automated Checks",
    "States": {
        "Perform Automated Checks": {
            "Type": "Parallel",
            "Branches": [{
                    "StartAt": "Check Identity",
                    "States": {
                        "Check Identity": {
                            "Type": "Task",
                            "Parameters": {
```

コンソールで視覚化



実行結果をモニタリング



Amazon States Language (ASL)

https://states-language.net/spec.html

```
{
    "Comment": "A simple minimal example",
    "StartAt": "Hello World",
    "States": {
      "Hello World": {
        "Type": "Task",
        "Resource": "arn:aws:lambda...HelloWorld",
        "End": true
      },
      [. . .]
  }
}
```

In Partnership with intel

AWS Step Functions のサービス統合

AWS Lambda

Amazon Elastic Container Service

AWS Batch

Amazon DynamoDB

AWS Glue

Amazon SageMaker

AWS Step Functions

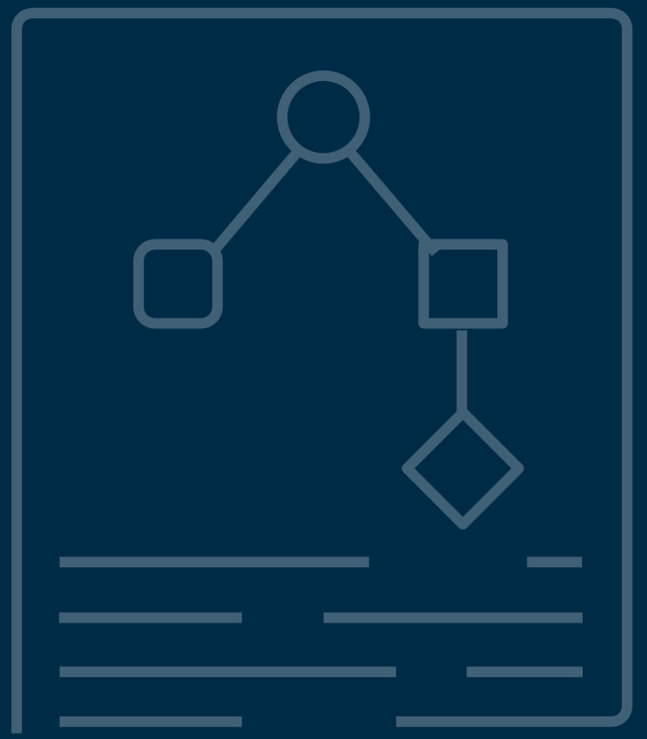
Amazon Simple Notification Service

Amazon Simple Queue Service

In Partnership with intel

ステートタイプ

Task	タスクを実行
Choice	分岐ロジックを追加
Wait	遅延時間を追加
Parallel	並列に分岐を実行
Map	ステートマシンで入力配列のアイテムを個別に処理
Succeed	成功した実行のシグナルと停止
Fail	失敗した実行のシグナルと停止
Pass	入力を出力にパス



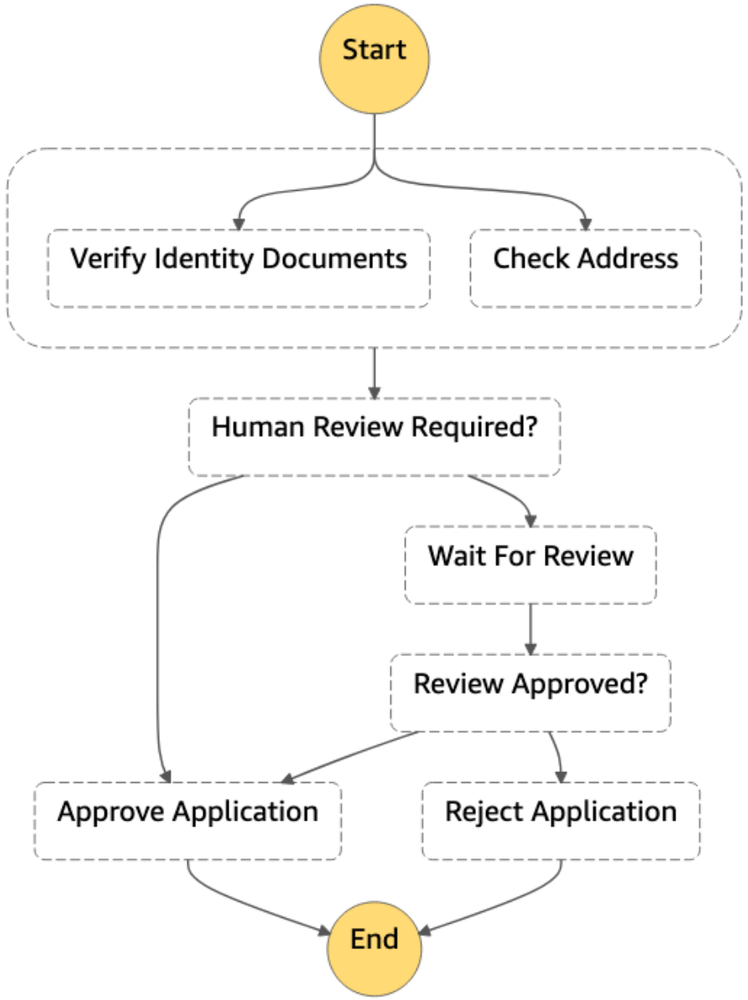
サンプルワークフロー: 銀行口座の開設

タスクの実行

並行ステップ

Choice による分岐

コールバックの待機

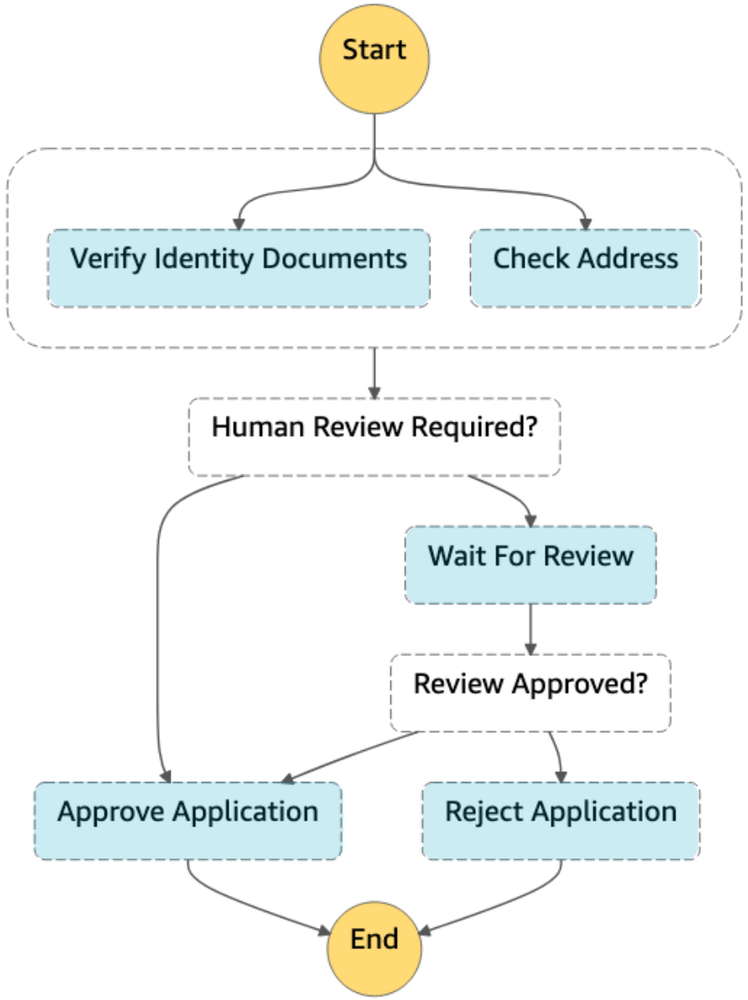


タスクの実行

AWS Lambda関数の呼び出し

アクティビティの実行のために
ポーリングするワーカーを待機

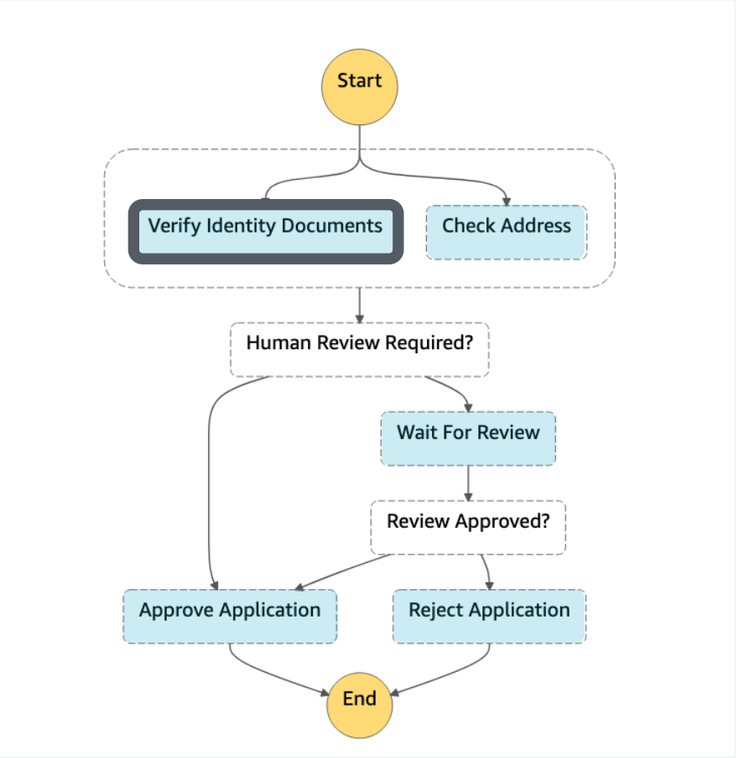
統合されたAWSサービスのAPI
へパラメータを渡す



タスクの実行

例: AWS Lambda関数の実行

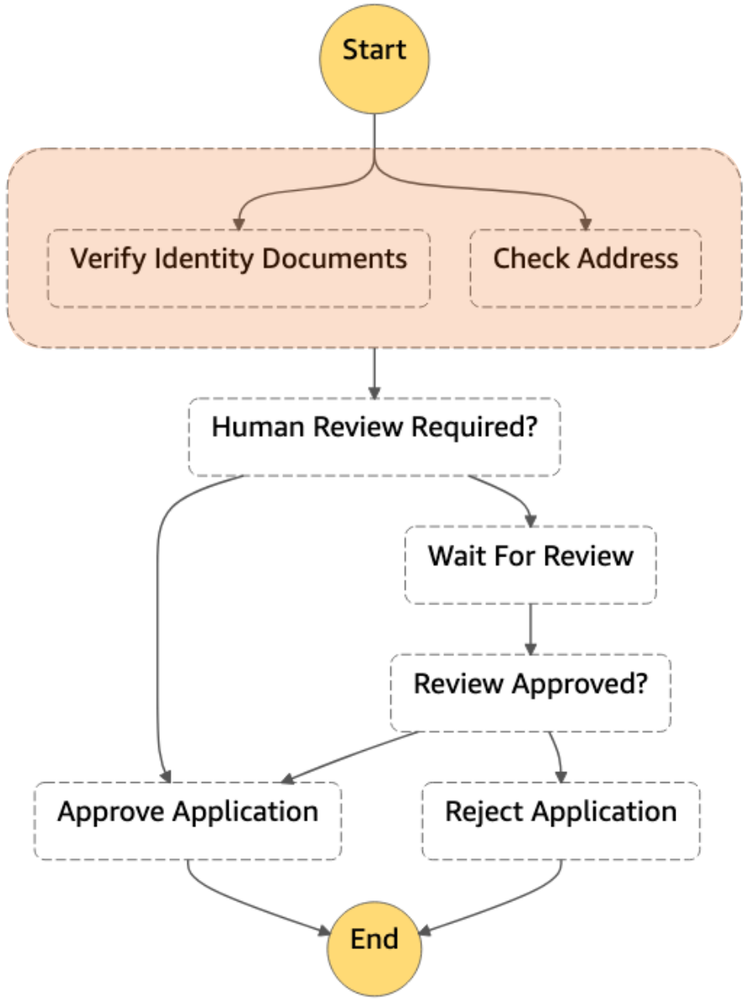
```
"Verify Identity Documents": {
  "Type": "Task",
    "Parameters": {
      "name.$": "$.application.name"
      "identityDoc.$": "$.application.idDocS3path"
    },
    "Resource": "arn:aws:lambda...VerifyIdDocs",
    "End": true
}
```



並列に分岐を実行

並列に実行するために分岐する
ブランチの配列を含む

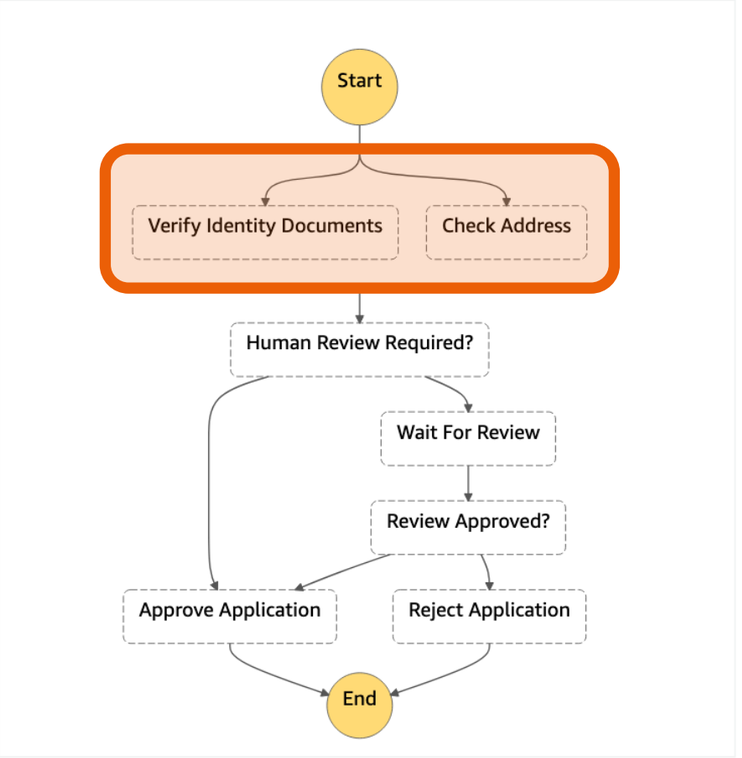
分岐された個々のブランチからの
アウトプット配列を出力



並列に分岐を実行

例: 2つの分岐を並列に実行

```
"Perform Automated Checks": {
 "Type": "Parallel",
  "Branches": [
    {
      "StartAt": "Verify Identity Documents",
      "States": { "Verify Identity Documents": { … } }
    },
    {
      "StartAt": "Check Address",
      "States": { "Check Address": { … } }
    }
  ]
 },
 "ResultPath": "$.checks",
 "Next": "Human Review Required?"
}
```

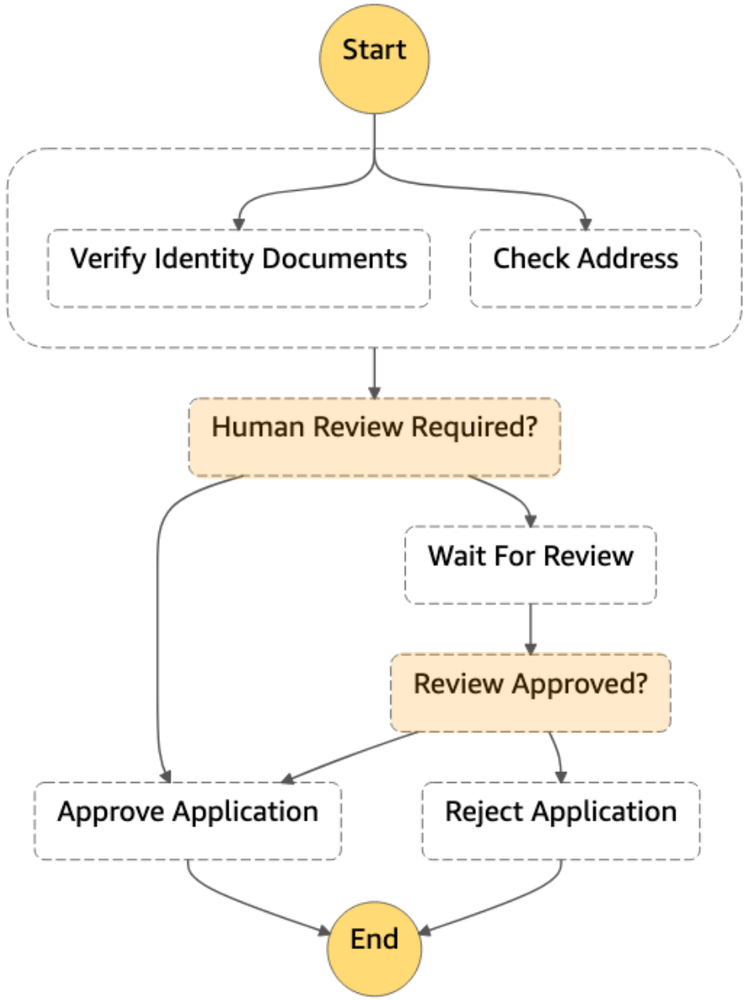


Choiceを作成

プログラミングのswitch文に似ている

choice構文の配列を検査し、入力変数とChoise配列要素の値を比較

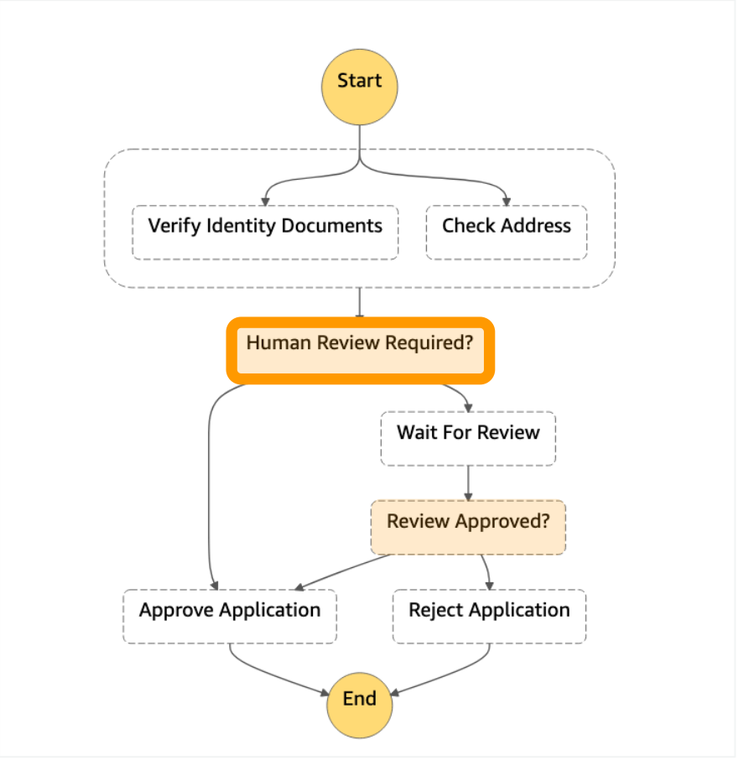
次にどのステートへ遷移するかを決定



choiceの作成

例: 状態のアウトプットに基づいて次のステップを選択

```
"Human Review Required?": {
  "Type": "Choice",
  "Choices": [
    {
      "Variable": "$.checks[0].flagged",
      "BooleanEquals": true,
      "Next": "Wait For Review"
    },
    {
      "Variable": "$.checks[1].flagged",
      "BooleanEquals": true,
      "Next": "Wait For Review"
    }
  ],
  "Default": "Approve Application"
```

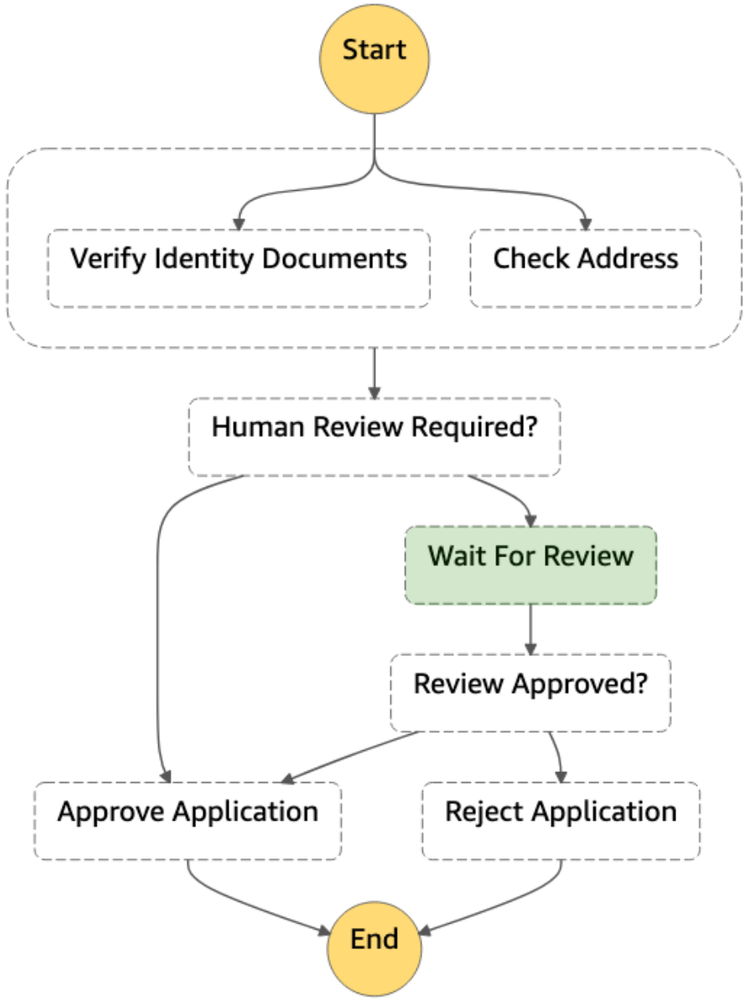


コールバックの待機

Taskトークンを生成し統合サービスにパスする

トークン受領者側プロセスが完了した時、SendTaskSuccessまたはSendTaskFailureをTaskトークンと共に呼び出す

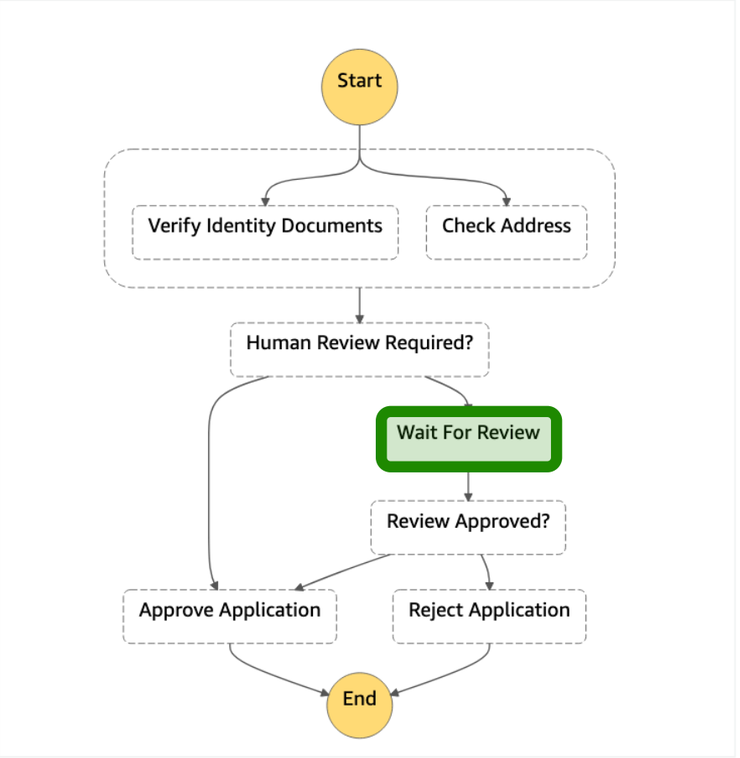
ワークフローが実行を再開する



コールバックの待機

例: ポーズして外部のコールバックを待機

```
"Type": "Task",
"Resource":"arn:aws:states:::lambda:invoke.waitForTaskToken",
"Parameters": {
   "FunctionName": "FlagApplicationForReview",
   "Payload": {
      "applicationId.$": "$.application.id",
      "taskToken.$": "$$.Task.Token"
   }
},
"ResultPath": "$.reviewDecision",
"Next": "ReviewApproved?"
```



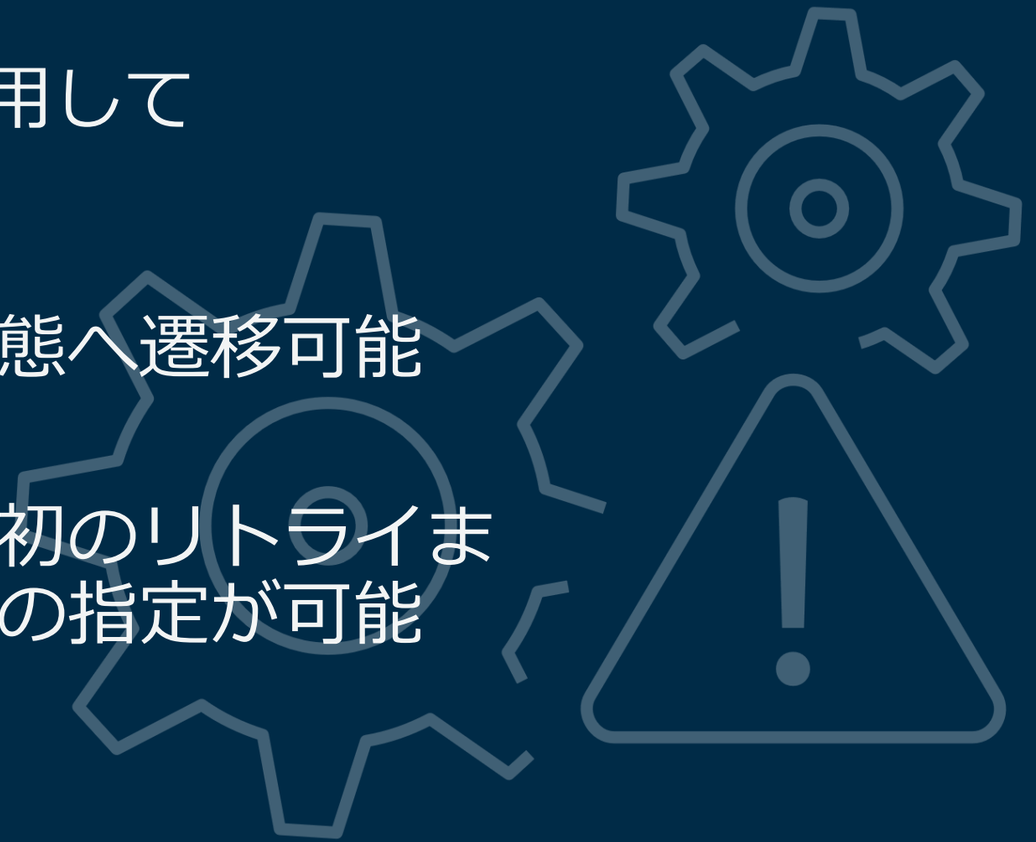
エラーハンドリングとリトライ

タイムアウト、失敗したタスク、権限不足によってエラーは起こり得る

タスクはエラー発生時にBackoffRateを利用してMaxAttemptsに達するまでリトライ可能

タスクは特定のエラーをキャッチし他の状態へ遷移可能

エラーに対してリトライの設定が可能。最初のリトライまでの間隔、最大回数、リトライ間隔の増分の指定が可能



In Partnership with intel

リトライとエラーハンドリングの例

```
"ReleaseInventory": {
    "Comment": "Task to release order items back to inventory",
    "Type": "Task",
    "Resource": "${ReleaseInventoryFunction.Arn}",
    "TimeoutSeconds": 10,
    "Retry": [{
      "ErrorEquals": ["States.ALL"],
      "IntervalSeconds": 1,
      "MaxAttempts": 2,
      "BackoffRate": 2.0
    }],
    "Catch": [{
      "ErrorEquals": ["ErrReleaseInventory"],
      "ResultPath": "$.error",
      "Next": "sns:NotifyReleaseInventoryFail"
    }],
    "Next": "ProcessRefund"
  },
```

ErrorEquals: リトライ対象のエラー
IntervalSeconds: 最初のリトライまでの間隔
MaxAttempts: リトライを実施する最大回数
BackoffRate: リトライ間隔の増分係数

ErrorEquals：キャッチするエラー
ResultPath：次のステートの入力となるPath
Next：例外発生時の遷移先ステート

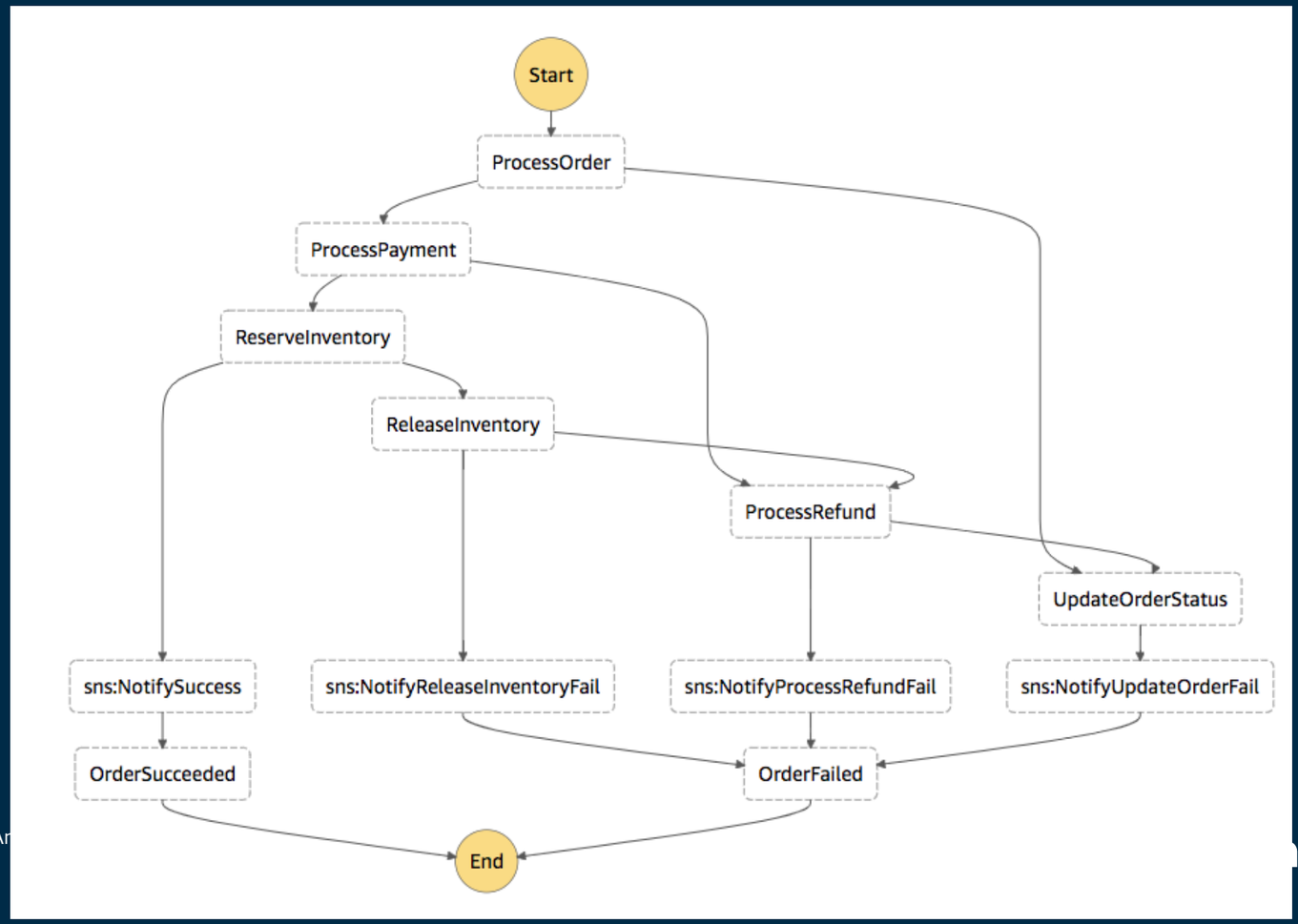
aws DEV DAY
ONLINE JAPAN

AWS Step FunctionsによるSagaパターンの実装

In Partnership with intel

AWS Step FunctionsによるSagaパターンの実装

- **Managing Long Lived Transactions with AWS Step Functions**
  https://github.com/aws-samples/aws-step-functions-long-lived-transactions



サンプルアプリケーションの解説

- e-Commerceの注文処理のサンプルアプリ
  - オーダーステータスの変更
  - クレジットカードトランザクション
  - 商品在庫数の変更

- 個々のステートは 独自のローカルなトランザクションを伴う様々なバックエンドと統合されている
- これらを一連の分散トランザクションとして扱う必要がある

ステートマシンの各ステートと種別

ステート	アクション	処理内容
ProcessOrder	Lambda関数呼び出し	DynamoDBのOrderテーブルに注文情報を保存
ProcessPayment	Lambda関数呼び出し	DynamoDBのPaymentテーブルに支払情報を保存
ReserveInventory	Lambda関数呼び出し	DynamoDBのInventoryテーブルに在庫予約情報をContext付きで保存
sns:NotifySuccess	SNSにTopicを送信	order_idを送信
ReleaseInventory	Lambda関数呼び出し	DynamoDBのInbventryテーブルに在庫予約情報の取り消しをContext付きで保存
sns:NotifyReleaseInventoryFail	SNSにTopicを送信	order_idを送信
ProcessRefund	Lambda関数呼び出し	DynamoDBのPaymentテーブルに返金情報を保存
sns:NotifyProcessRefundFail	SNSにTopicを送信	order_idを送信
UpdateOrderStatus	Lambda関数呼び出し	DynamoDBのOrderテーブルの注文ステータスをPendingに更新
sns:NotifyUpdateOrderFail	SNSにTopicを送信	order_idを送信

トランザクション　補償トランザクション

ステートマシンの各ステートと種別

現在のステート	次のステート	失敗した場合の次のステート
ProcessOrder	ProcessPayment	UpdateOrderStatus
ProcessPayment	ReserveInventory	ProcessRefund
ReserveInventory	sns:NotifySuccess	ReleaseInventory
sns:NotifySuccess	OrderSucceeded	
ReleaseInventory	ProcessRefund	sns:NotifyReleaseInventoryFail
sns:NotifyReleaseInventoryFail	OrderFailed	
ProcessRefund	UpdateOrderStatus	sns:NotifyProcessRefundFail
sns:NotifyProcessRefundFail	OrderFailed	
UpdateOrderStatus	sns:NotifyUpdateOrderFail	sns:NotifyUpdateOrderFail
sns:NotifyUpdateOrderFail	OrderFailed	

トランザクション　補償トランザクション

aws DEV DAY
ONLINE JAPAN

Demo – サンプルアプリケーションの実行

In Partnership with intel

まとめ

- Sagaパターンは長期生存トランザクションのパフォーマンス低下やデッドロックによる失敗発生率の増加に対応するパターン
- マイクロサービスパターンによって再度注目された
- AWS Step Functionsを利用することで、Sagaパターンをオーケストレーションで実装可能
- AWS Step FunctionsにSagaの管理を移譲することで責務を分離し、アプリケーションコードをシンプルに保つことが可能

aws DEV DAY
ONLINE JAPAN

Thank you!

Atsushi Fukui
Twitter: @afukui

In Partnership with intel