WEBアプリケーションにおける AWS Lambdaを用いた 大規模な非同期処理の実践

PyCon JP 2024 2024.9.28(土)

奥寺政貴

BePROUD

自己紹介

- 奥寺政貴(おくでらまさたか)
- 株式会社ビープラウド所属(2020～)
- バックエンドエンジニア
- 仕事では主にPython x Djangoを使用
- 趣味: インド料理屋巡り
- **資料は公開してます**

Be PROUD

話すこと

- AWS Lambda(Python) x Amazon SQSをプロダクションレベルで安定稼働させるためのノウハウの共有
  - → 便利なPythonライブラリの紹介
- 弊社サービスのconnpassのメール送信のアーキテクチャ移行の事例がベース

Be PROUD

動機

- AWS Lambda x Amazon SQSの構成を導入したらconnpassが抱えていた性能問題が改善した
- 実際のサービスに投入しようとすると、非機能要件周りで細かいノウハウが色々必要だった(情報が少なくて苦労した)
- → ノウハウを公開して後続の人達の役に立ちたい

BePROUD

話さないこと

- アーキテクチャの選定時の比較検討
- AWSサービスの基本的な説明(SQS, S3, DynamoDB, SES)
- connpassのサービス概要

※ AWSのサービス名の省略

- AWS Lambda → Lambda
- Amazon SQS → SQS
- Amazon S3 → S3
- Amazon DynamoDB → DynamoDB
- Amazon SES → SES

Be PROUD

お品書き

1. **connpassのメール送信の課題**
2. Lambda x SQSの導入による課題解決
3. Lambda x SQSのプロダクションレベルの知見
    1. SQSの256KB上限問題(データサイズ)
    2. 冪等性の担保
    3. エラーハンドリング

connpassのメール送信の課題

Be PROUD

前提

connpassには多種多様なメール通知がある

メール通知設定

友達

- [ ] フォローしている人が新しくconnpassを使い始めた時
- [x] フォローしている人がイベントを公開した時
- [x] フォローしている人のイベントが募集開始した時
- [ ] フォローしている人がイベントに参加した時

開催イベント

- [x] 開催するイベントが公開された時
- [x] 開催するイベントが募集開始した時
- [x] 開催しているイベントの参加者が増えた時
- [x] 開催しているイベントで参加のキャンセルがあった時
- [x] 開催しているイベントに新しい資料が追加された時
- [x] 開催しているイベントのリマインダー

発表イベント

- [x] 発表したイベントに新しい資料が追加された時

参加イベント

- [x] 補欠から繰り上がった時
- [x] 参加しているイベントが中止された時
- [x] 参加しているイベントに新しい資料が追加された時
- [x] 参加しているイベントのリマインダー
- [x] 参加したイベントが終わった後のおすすめやTips

発表・参加イベント

- [x] 終わって一週間以上経過した、発表・参加したイベントの管理者からのメッセージ

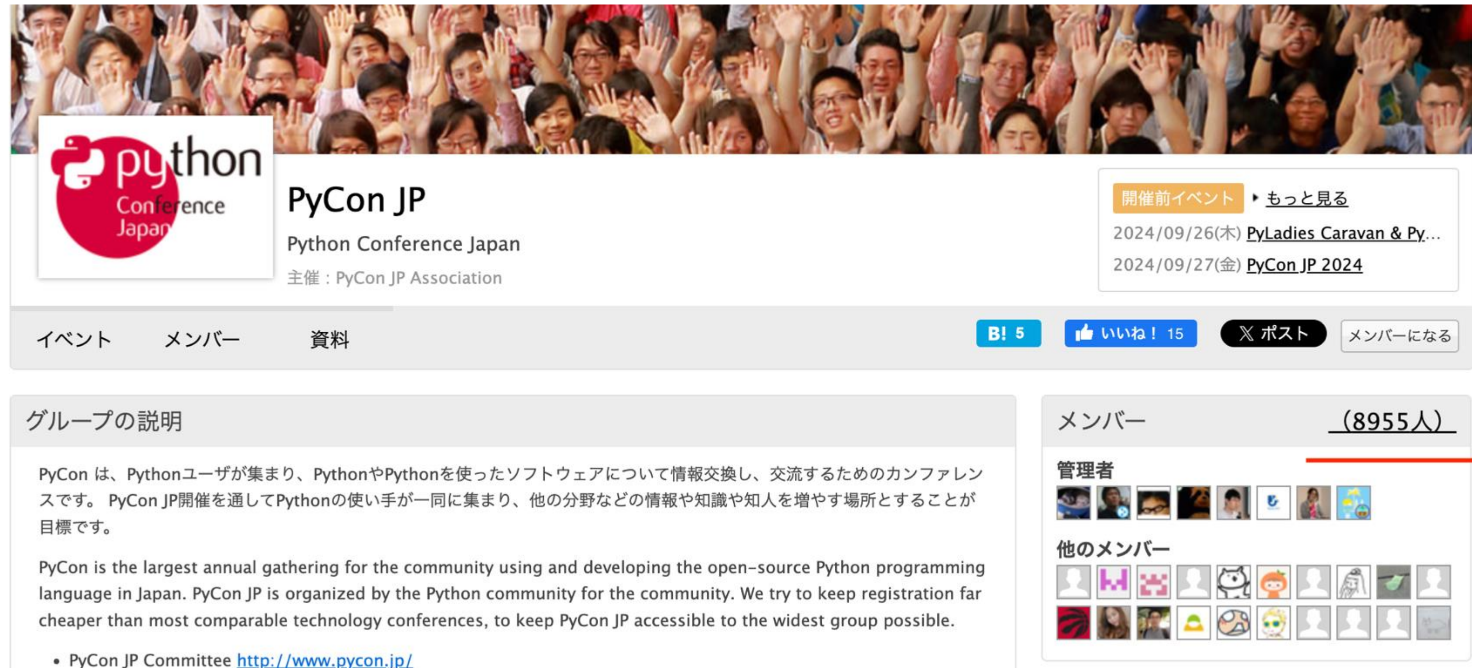
connpassからのお知らせ

- [x] connpassからの重要なお知らせやサービスの更新情報
- [x] connpassのおすすめやTips

Be PROUD

前提

- グループメンバーに一斉送信するメールなどもある
- → 瞬間送信件数1万件以上とかもしばしば



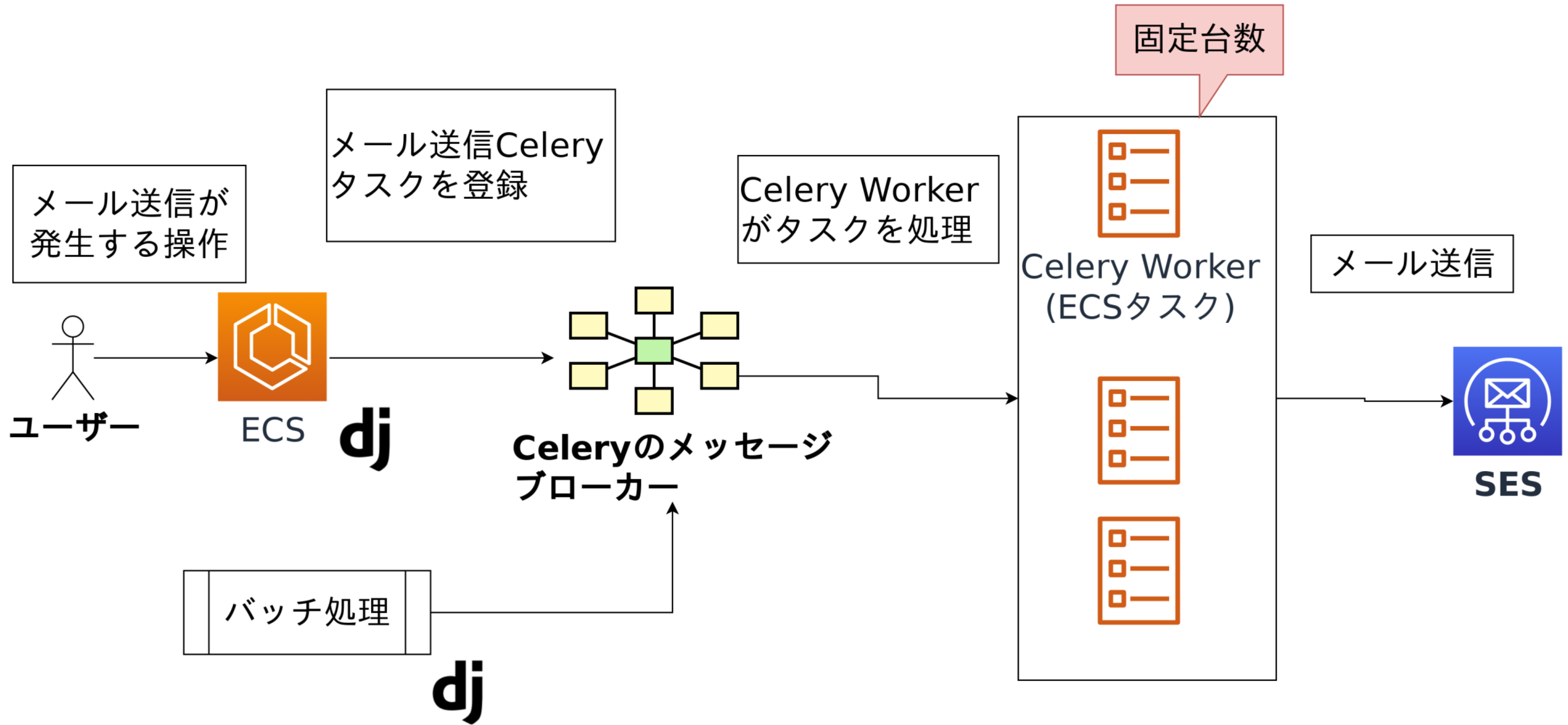
メール送信のアーキテクチャ(移行前)

- 前提: connpassではクラウドはAWSを使用
- 移行前はCeleryによる非同期処理でメール送信
- Celery WorkerはECSタスクで稼働

※ そもそも何でメール送信を非同期処理にするの？という方は以下を参照

https://jisou-programmer.beproud.jp/サーバー構成/91-時間のかかる処理は非同期化しよう.html

Be PROUD


固定台数
メール送信Celery
タスクを登録
メール送信が
発生する操作
Celery Worker
がタスクを処理
Celery Worker
(ECSタスク)
メール送信
ユーザー
ECS
Celeryのメッセージ
ブローカー
SES
バッチ処理

課題

- 瞬間送信件数1万件以上とかもしばしば
- → Celery Workerがタスクを捌き切るのに時間がかかる(=タスクの滞留)

BePROUD

滞留によって起きる問題

1. ユーザーへのメール送信が遅れる
2. 派生してDB負荷が高まる(CPU100%近く)
3. アラートの見守り
    1. 時間を取られる
    2. マルチタスク

→ **解決したい**

BePROUD

お品書き

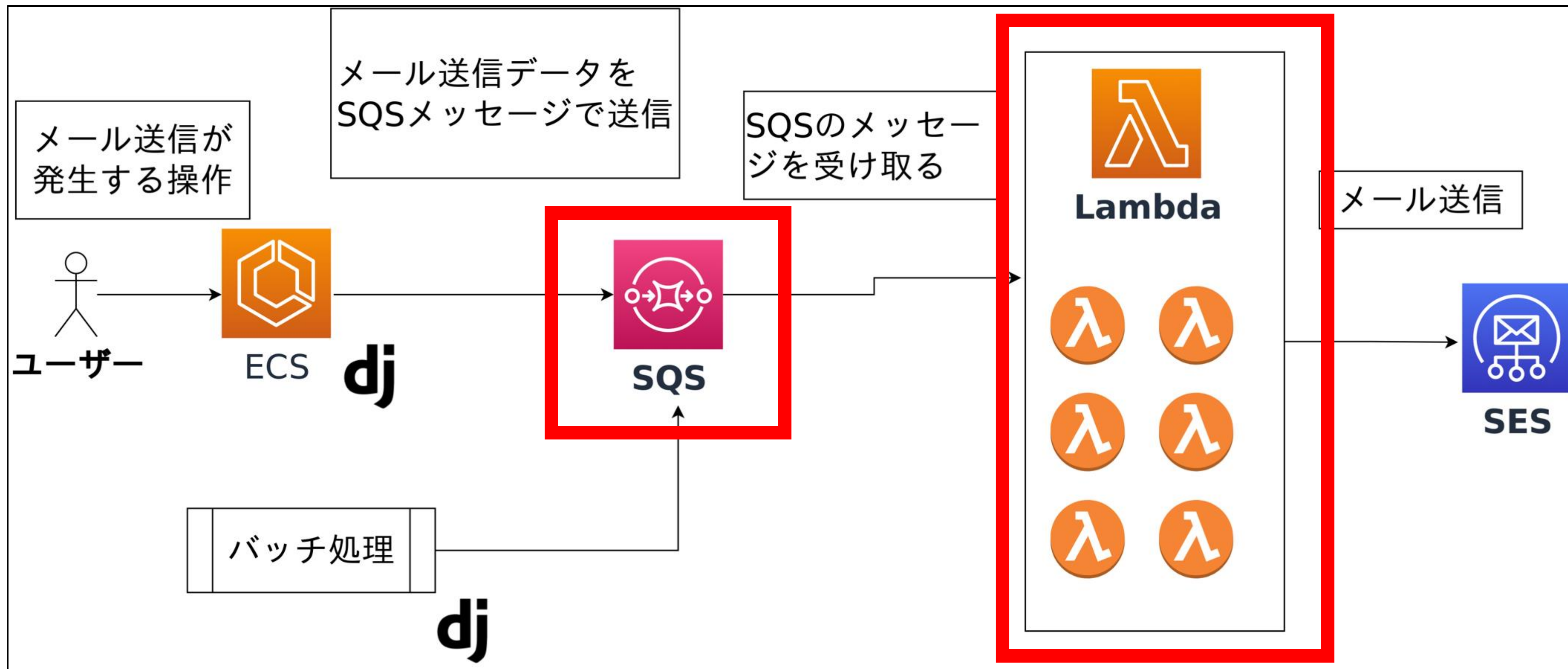
1. connpassのメール送信の課題
2. **Lambda x SQSの導入による課題解決**
3. Lambda x SQSのプロダクションレベルの知見
    1. SQSの256KB上限問題(データサイズ)
    2. 冪等性の担保
    3. エラーハンドリング

Lambda x SQSの導入による課題解決

Be PROUD

Lambda x SQSの構成に

CeleryからLambda x SQS(AWSのメッセージキューサービス)の構成に
※ SQSは並列実行したいので標準キューを使用(= FIFOキューは使わない)



BePROUD

AWS Lambdaとは

- AWSのイベント(SQSやS3など)をトリガーに処理(関数)を実行できるサービス
- 様々な言語での実装に対応している(Java, Ruby, Node.js...)
- AWSサービス間の仲介役としてよく使われるので「糊」と言われたりする

※ 本発表はもちろんPython実装の前提

Be PROUD

Lambdaが課題にフィットすると思ったポイント

需要に応じたオートスケーリング(Automatic scaling)

- 関数実行のリクエスト数が増えると自動でスケールアウトする
- 最大1000インスタンスまで並列実行可能

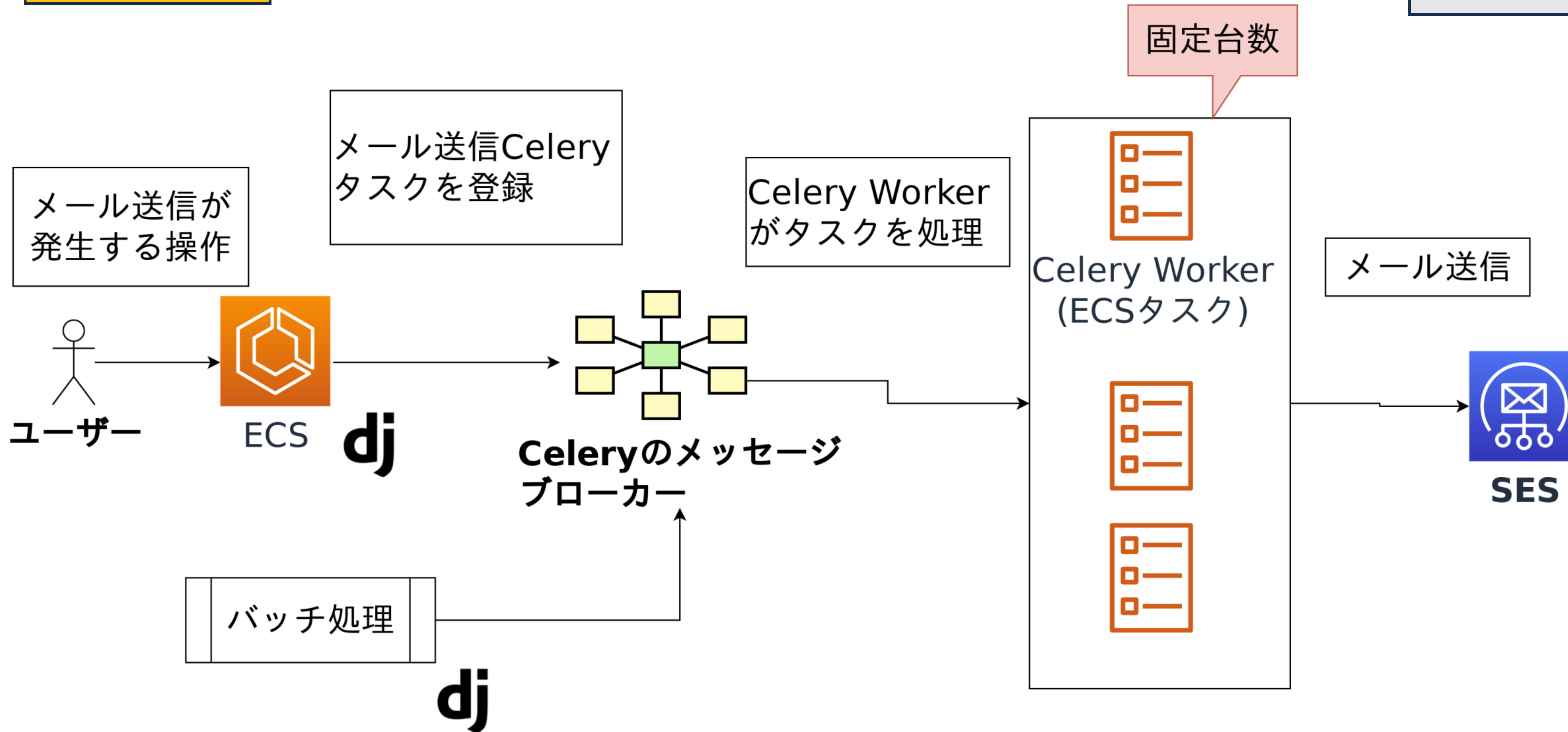
BePROUD

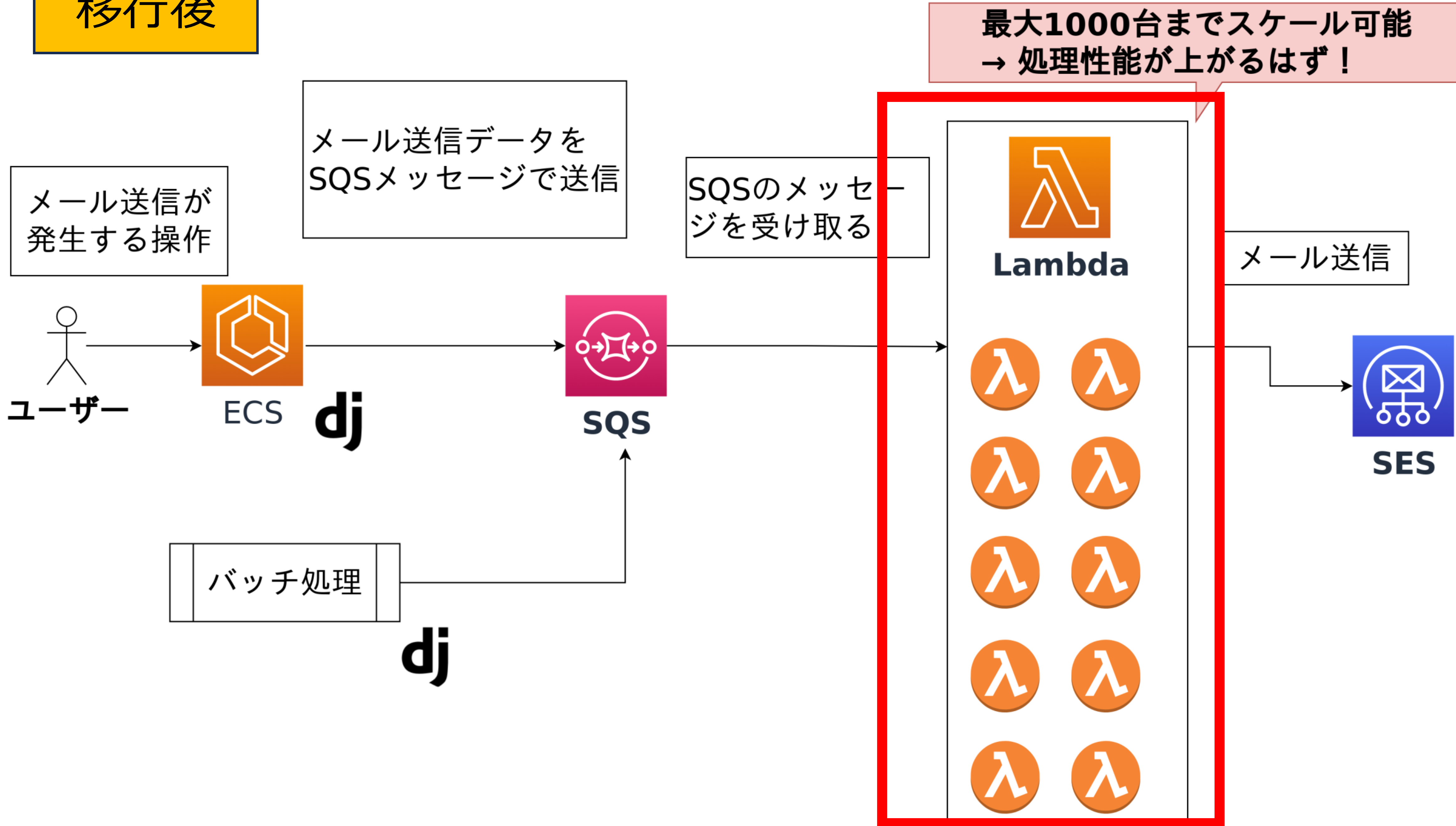
引用元: https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/lambda-concurrency.html

connpassの課題に当てはめてみると

- **必要な時に必要な分だけ**インスタンスが起動するので、**需要が変動しやすく処理時間が短い**メール配信と相性良さそう
- **Lambdaの上限1000インスタンス** >> Celery Worker数(合計10プロセス程度で運用)
  - → 遥かに並列処理能力が上がるはずなので期待できそう

Be PROUD

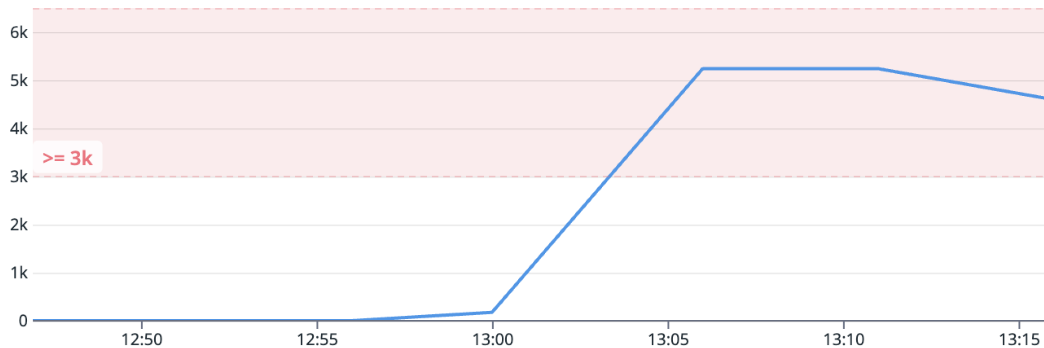

移行前
再掲
固定台数
メール送信Celery
タスクを登録
Celery Worker
がタスクを処理
メール送信が
発生する操作
Celery Worker
(ECSタスク)
メール送信
ユーザー
ECS
Celeryのメッセージ
ブローカー
SES
バッチ処理

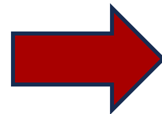

移行後
最大1000台までスケール可能
→ 処理性能が上がるはず！
メール送信データを
SQSメッセージで送信
メール送信が
発生する操作
SQSのメッセージを受け取る
Lambda
メール送信
ユーザー
ECS
SQS
SES
バッチ処理

移行した成果

- ほぼ滞留なくメッセージを処理できるようになった
- キュー数のメトリクスが山なりのグラフからスパイク状のグラフに変わった -> 滞留解消と判断

移行前(Celery)





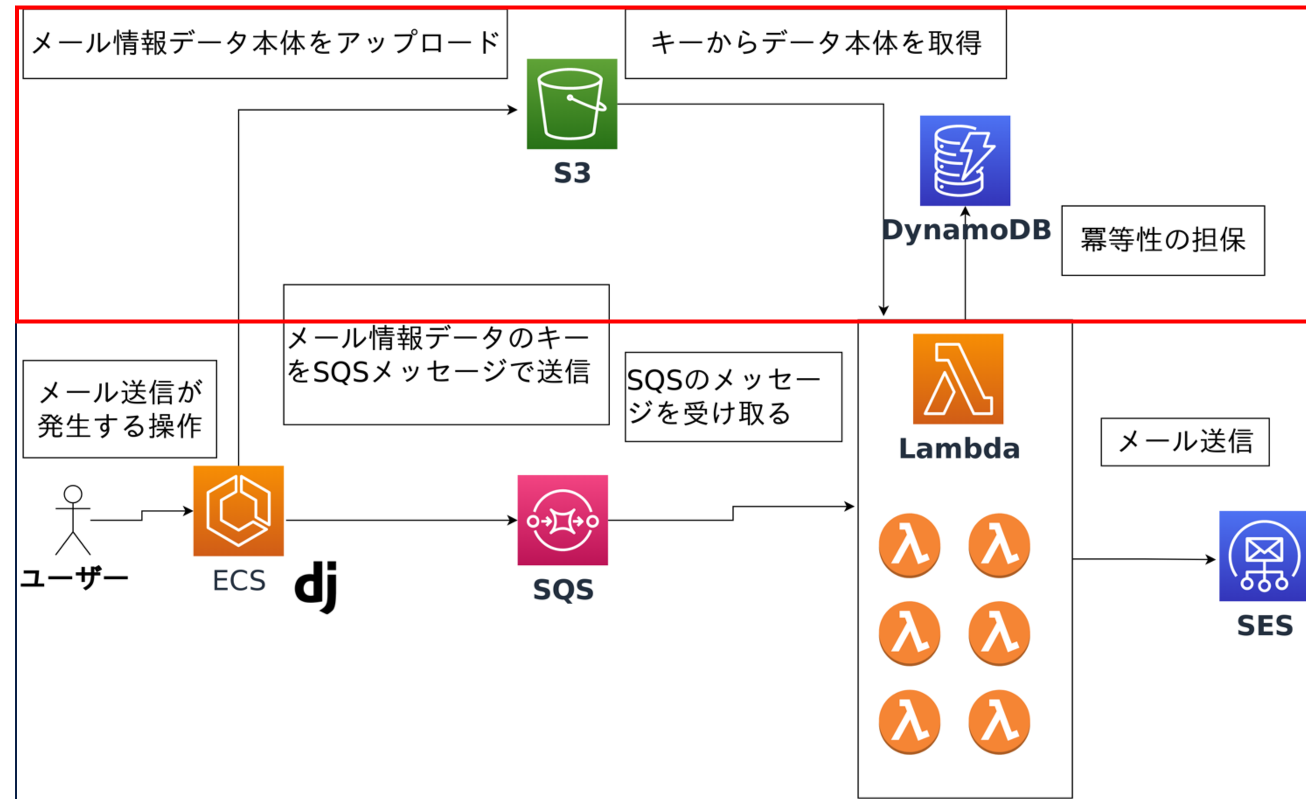
Lambda x SQS移行後

送信済みメッセージの数

正式なインフラ構成

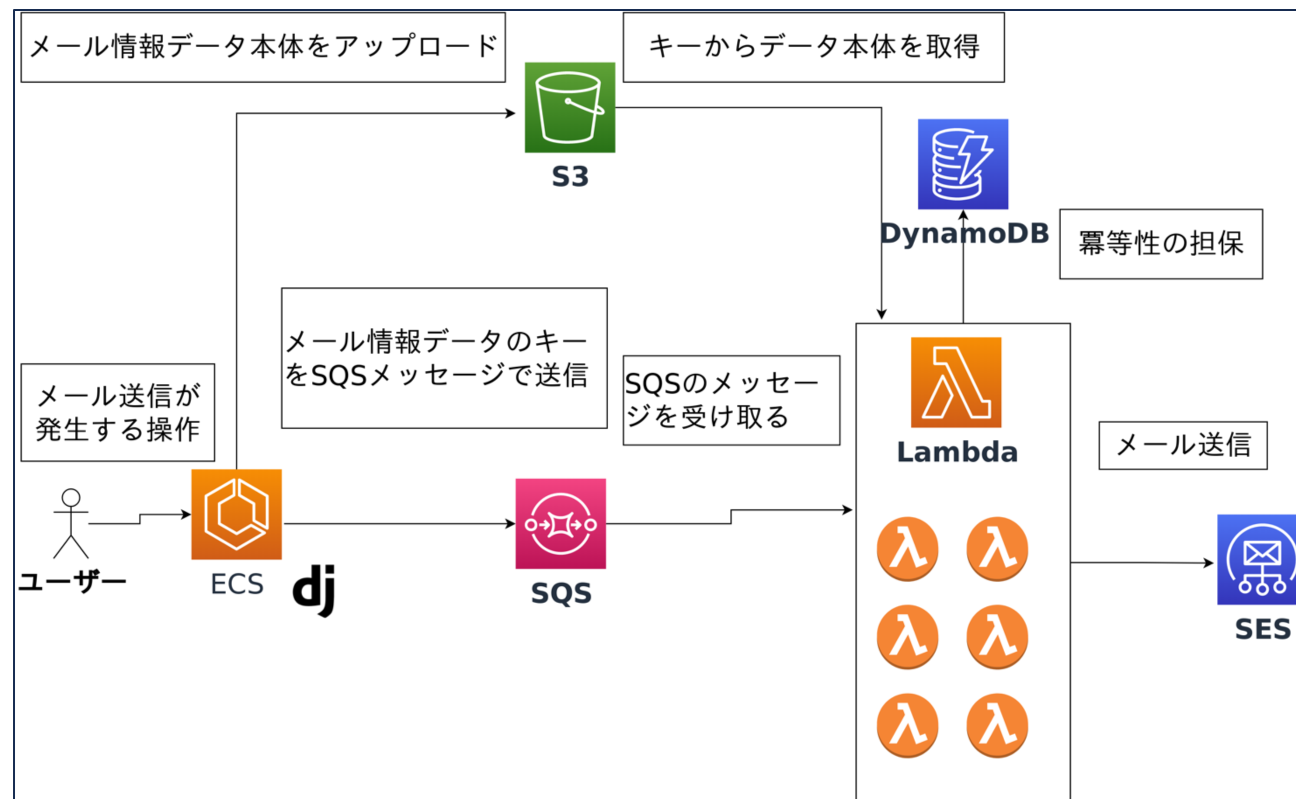
非機能要件(これから話す)まで考慮すると**S3**と**DynamoDB**も必要だった
→登場人物を全部書いた正式なインフラ構成図は以下

※「バッチ処理」は線がゴチャゴチャするので以降省略



【まとめ】2. Lambda x SQSの導入による課題解決

- CeleryからLambda x SQSの構成に移行することでキュー滞留の問題を解消できた🎉
- 移行後のインフラ構成図は以下



お品書き

1. connpassのメール送信の課題
2. Lambda x SQSの導入による課題解決
3. **Lambda x SQSのプロダクションレベルの知見**
    1. SQSの256KB上限問題(データサイズ)
    2. 冪等性の担保
    3. エラーハンドリング

BePROUD

AWS Lambda x SQSのプロダクションレベルの知見

BePROUD

プロダクションレベルとは

趣味で作ったアプリやまだユーザー数が少ないアプリとの違い
→ **非機能要件の考慮ができている**

- データサイズ
- 冪等性
- エラーハンドリング

BePROUD

お品書き

1. connpassのメール送信の課題
2. Lambda x SQSの導入による課題解決
3. Lambda x SQSのプロダクションレベルの知見
    1. **SQSの256KB上限問題(データサイズ)**
    2. **冪等性の担保**
    3. **エラーハンドリング**

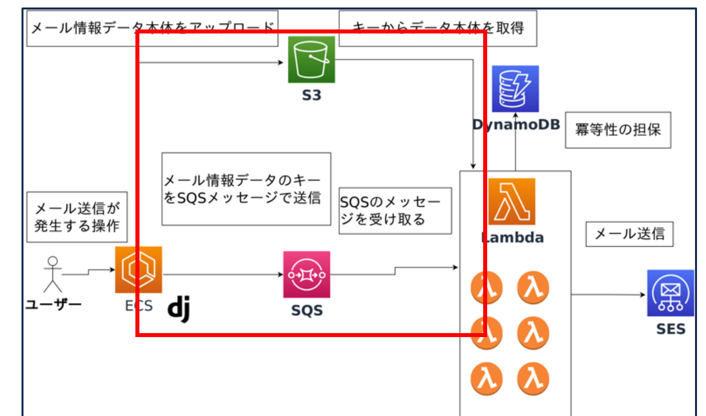
非機能要件の話

お品書き

1. connpassのメール送信の課題
2. どうやって解決したか
    1. AWS Lambda x SQSの構成への移行
3. AWS Lambda x SQSのプロダクションレベルの知見
    1. **SQSの256KB上限問題(データサイズ)**
    2. 冪等性の担保
    3. エラーハンドリング

BePROUD

SQSの256KB上限問題(データサイズ)

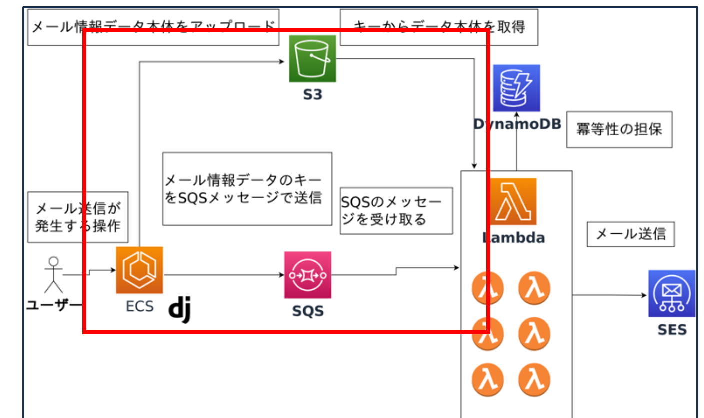


Be PROUD

SQSのメッセージの上限

SQSのメッセージのサイズの上限: 256KB(超概算で5万文字列以上)

”最大メッセージサイズを使用する場合、値を入力します。指定できる指定できる範囲は1KB から256KB です。デフォルト値は、256KBです”

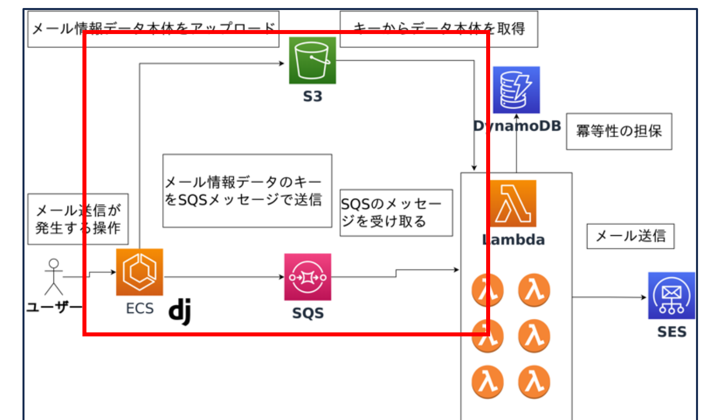


Be PROUD

引用元: https://docs.aws.amazon.com/ja_jp/AWSSimpleQueueService/latest/SQSDeveloperGuide/creating-sqs-standard-queues.html

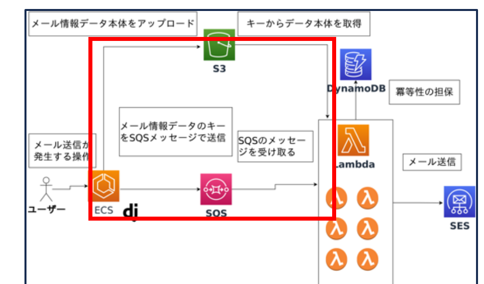
考えられる方針

- 案A. メールサイズが256KBを超えることは現実的に考えにくいので何もしない
- 案B. 念のため256KBを超えても大丈夫なようにしておく



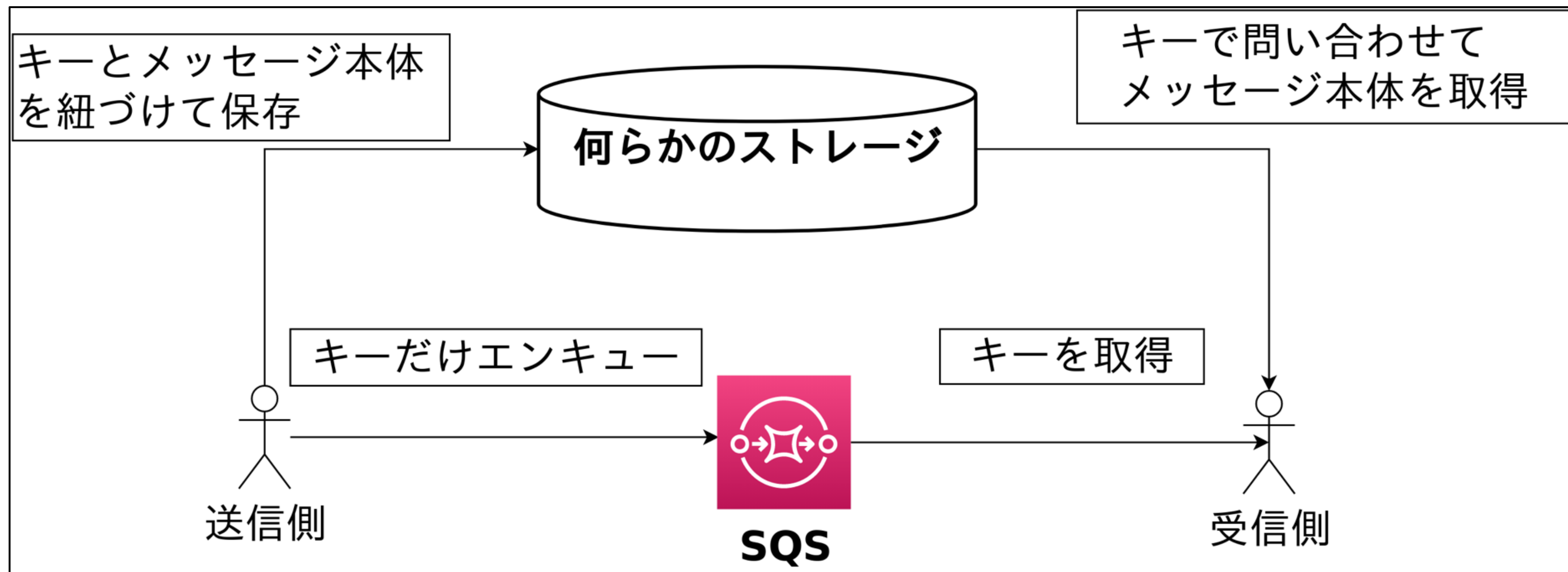
案Bを採用

- 案A. メールの文字数が256KBを超えることは現実的に考えにくいので何もしない
- **案B. 念のため256KBを超えても大丈夫なようにしておく ← 採用**
  - 理由1: 256KB制約を頭の片隅に置いておかないといけないのが気持ち悪い
  - 理由2: 今後、Lambda x SQSの仕組みを他の非同期処理にも展開する可能性がある
    - → 展開先では256KB以上のメッセージを扱うかもしれないので、実績を作っておきたい



解決案

SQSにはデータのキーだけ持たせて、データ本体は何らかのストレージに置くとよさそう

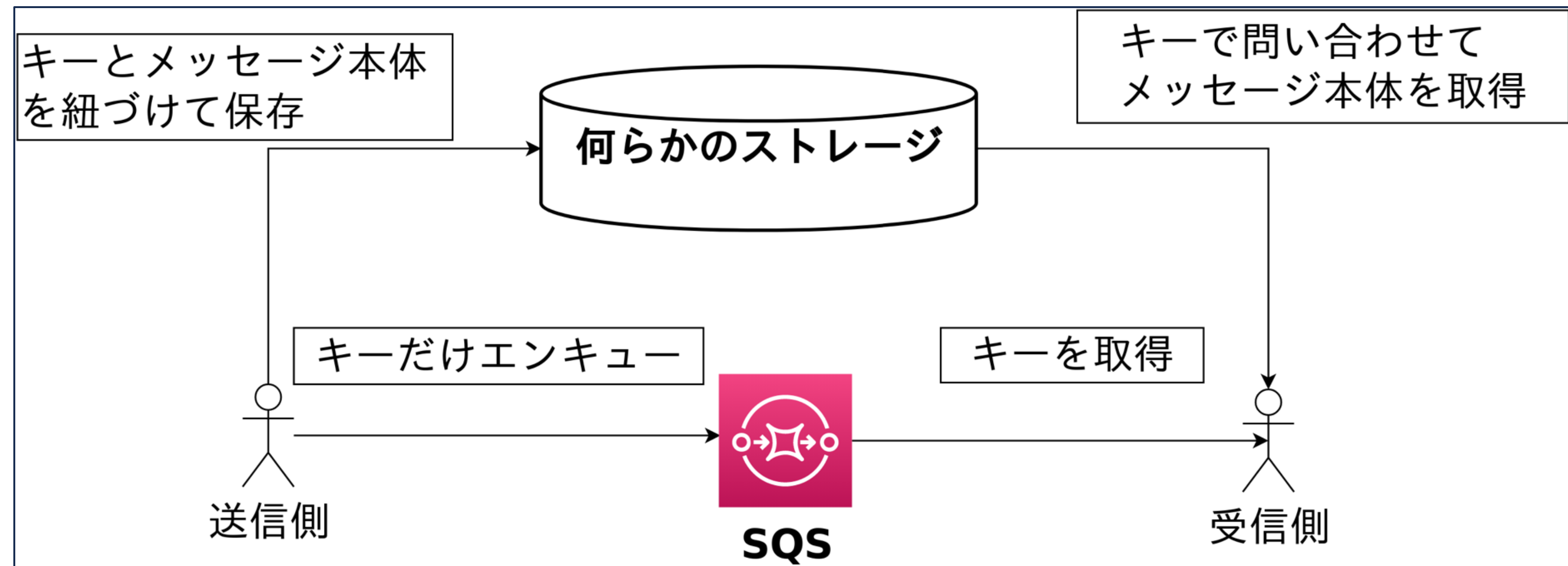


Be PROUD

解決案

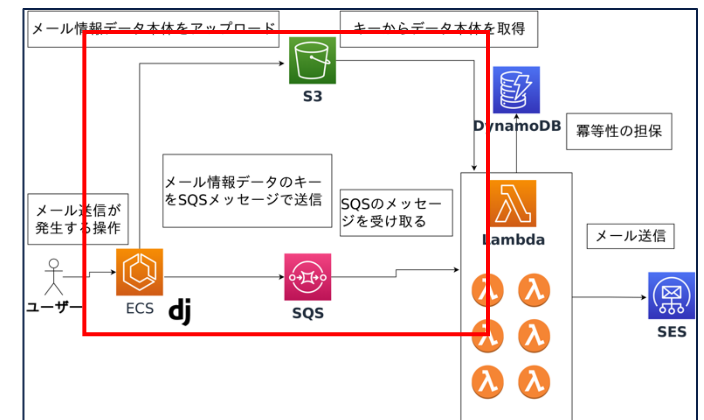
SQSにはデータのキーだけ持たせて、データ本体は何らかのストレージに置くとよさそう

→ アプリケーションの本質とは関係のないところなので、仕組みを自分で作りたくない



amazon-sqs-python-extended-client-lib

- **amazon-sqs-python-extended-client-lib**というライブラリがある
- 2024年2月リリース(それまではJava実装しかなかった)
- AWSの公式ドキュメントでも紹介されている3rdパーティライブラリ
- **pip install amazon-sqs-extended-client**

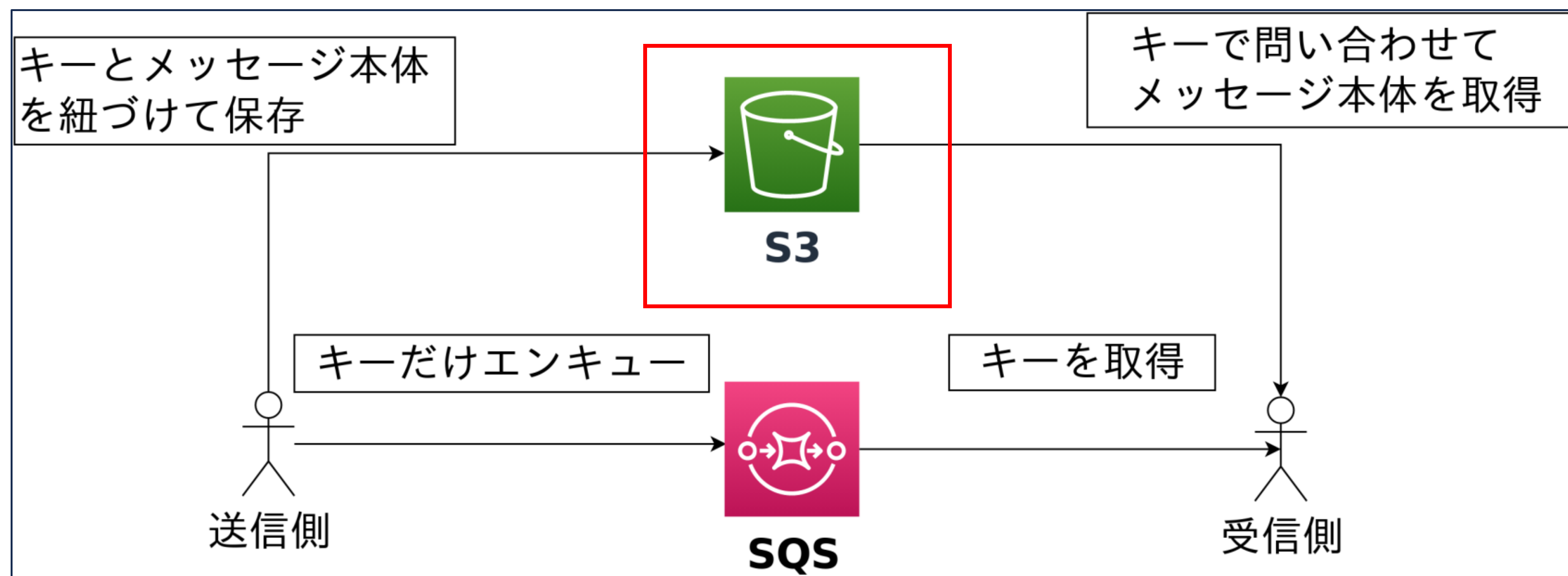


https://docs.aws.amazon.com/ja_jp/AWSSimpleQueueService/latest/SQSDeveloperGuide/extended-client-library-python.html

Be PROUD

amazon-sqs-python-extended-client-lib がやってくれること

仕組みのイメージはそのままで、**「何らかのストレージ」**に
S3(AWSのファイルストレージサービス)を使用



Be PROUD

使い方(SQSメッセージ送信側)

```
import boto3
import sqs_extended_client

sqs_client = boto3.client("sqs", region_name="ap-nor
sqs_client.large_payload_support = "my-bucket-name"
sqs_client.always_through_s3 = True
sqs_client.use_legacy_attribute = False
entries = [] # 実際はメッセージのデータを入れる
sqs_client.send_message_batch(QueueUrl="my-queue-url", Entries=entries)
```

boto3と合わせてインポート

S3バケット名(自分で作る)を設定

メッセージ送信

※ SQSはAPIリクエスト単位の課金なので、なるべく複数メッセージをまとめてバッチ送信したほうが低コスト

Be PROUD

使い方(SQSメッセージ受信側=Lambda側)

```python
import boto3
import sqs_extended_client

sqs_client = boto3.client("sqs", region_name="ap-northeast-1")
sqs_client.large_payload_support = "my-bucket-name"
sqs_client.always_through_s3 = True
sqs_client.use_legacy_attribute = False

def lambda_handler(event, context):
    for record in event["Records"]:
        s3_pointer: str = record["body"]
        message: str = sqs_client.retrieve_message_from_s3(s3_pointer)
```

S3からメッセージ本体を取得

Be PROUD

【補足】Lambdaで外部ライブラリ使う方法

Lambda側で pip install sqs-extended-client できるの？

結論: できる

理由: Lambdaへのソースコードのアップロード方法は以下の3通り

1. AWSマネージメントコンソール上で編集
2. **zipでアップロード**
3. **Dockerイメージでアップロード**

→ **2.か3.なら可能**

Be PROUD

【参考】connpassの場合

2.zipでアップロード を採用
→デプロイ時に **pip install requirements.txt して**zipするスクリプト書くのはちょっと辛い😢

connpassはデプロイに**Terraform**を使用
→ **terraform-aws-modules/lambda** を使用

※ 他にもやり方は色々あるはず

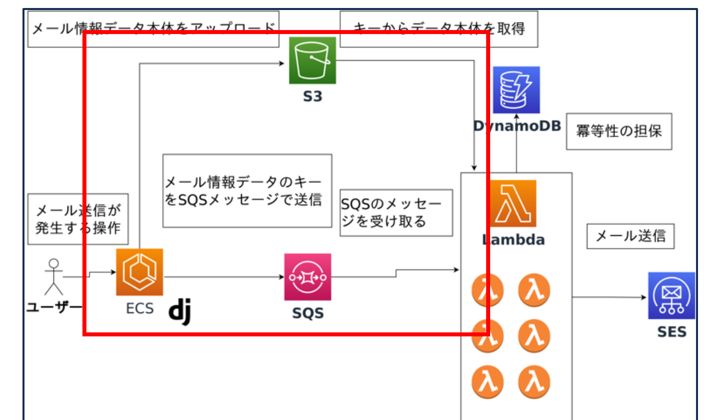
requirements.txtを置いておくだけでTerraformがよしなに処理してくれる

```
lambda_mail
├── main.tf
├── src
    ├── app.py
    └── requirements.txt
```

https://registry.terraform.io/modules/terraform-aws-modules/lambda/aws/latest

【まとめ】3-1.SQSの256KB上限問題

- SQSのメッセージには上限256KBの制約がある
- **amazon-sqs-python-extended-client-lib**を使うとほとんど自分でコードを書かずに制約を乗り越えられるのでオススメ

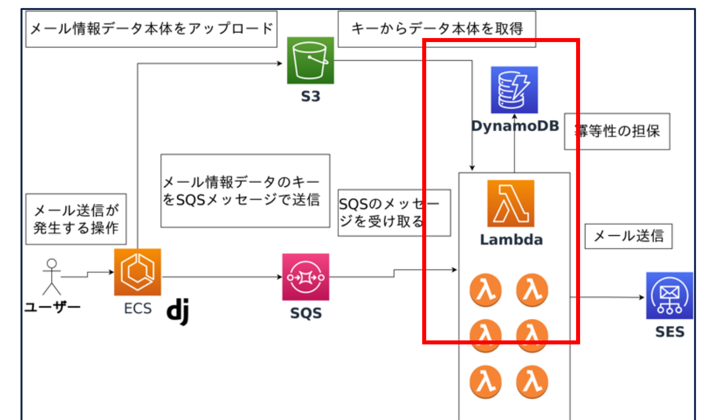


お品書き

1. connpassのメール送信の課題
2. AWS Lambda x SQSの導入による課題解決
3. AWS Lambda x SQSのプロダクションレベルの知見
   1. SQSの256KB上限問題(データサイズ)
   2. **冪等性の担保**
   3. エラーハンドリング

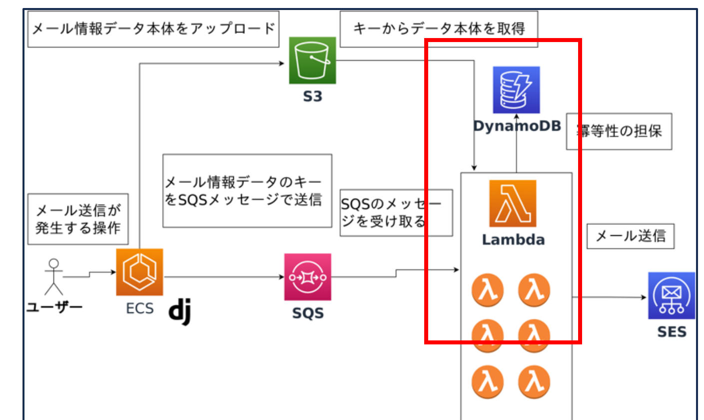
Be PROUD

冪等性の担保



冪等性とは

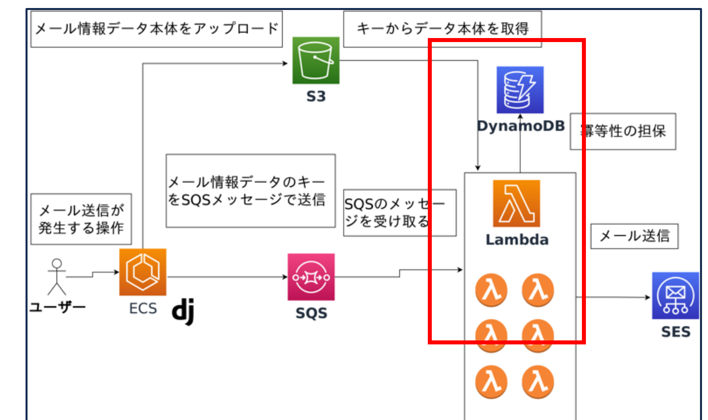
- 文脈によって広い意味で使われる言葉
- **本発表での定義: 同じSQSのメッセージをLambda関数が1回だけ処理すること**
  - 担保されていない場合の悪影響: 同一ユーザーへのメールの多重送信など



Be PROUD

同じメッセージを複数回処理するってあるの？

- ”Lambda は**少なくとも** 1 回のメッセージ配信をサポートしています。場合によっては、再試行メカニズムによって同じメッセージが重複して送信されることがあります。 ”
- → **要するに同じメッセージをLambdaが2回以上処理してしまう可能性はゼロではない**



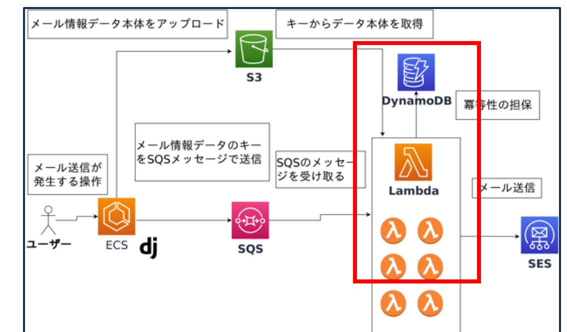
引用元: https://repost.aws/ja/knowledge-center/lambda-function-process-sqs-messages

Be PROUD

同じメッセージを複数回処理するってあるの？

- ”Lambda は少なくとも 1 回のメッセージ配信をサポートしています。場合によっては、再試行メカニズムによって同じメッセージが重複して送信されることがあります。 ”
- → **要するに同じメッセージをLambdaが2回以上処理してしまう可能性はゼロではない**

→ 同一ユーザーへのメールの多重送信は少量でも致命的なサービス影響なので対策が必要



引用元: https://repost.aws/ja/knowledge-center/lambda-function-process-sqs-messages

BePROUD

解決方法

DynamoDBを使ってロック処理をするのが良いとAWS公式が回答している

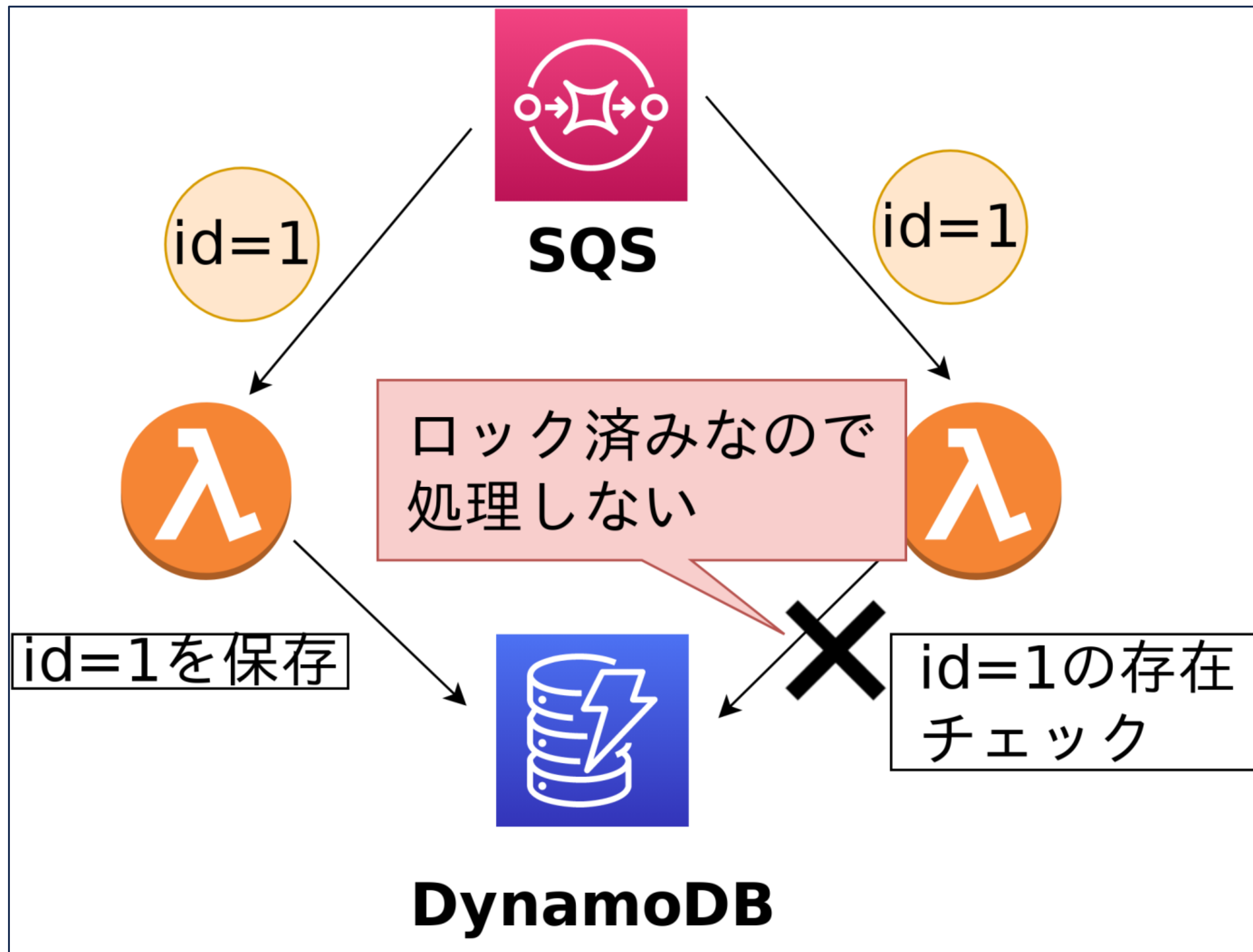
”Then, create a modular function code that iterates through the batch, processes, and deletes successful and duplicate messages. The function stores the messageID of the successful messages in an Amazon DynamoDB table and then verifies that the message was processed earlier.”

※ 日本語の自動翻訳がわかりにくかったので英語のまま引用

Be PROUD

引用元: https://repost.aws/ja/knowledge-center/lambda-function-process-sqs-messages

DynamoDBを使ったロックのイメージ図



Be PROUD

AWSが実装例を提供してくれてはいるが…

AWSが実装例を提供してくれているが、それを参考に自前で実装するのは気が進まなかった

- 理由1: サービスの本質に関わるコードではないので、これをコピペして自分たちのコードベースに入れるのは気が進まない
- 理由2: 厳密なロックを実装するなら追加実装が必要そう
  - DynamoDBへのロック用のレコードの作成と存在チェックを別のコマンドでやっており、エッジケースには対応してなそう

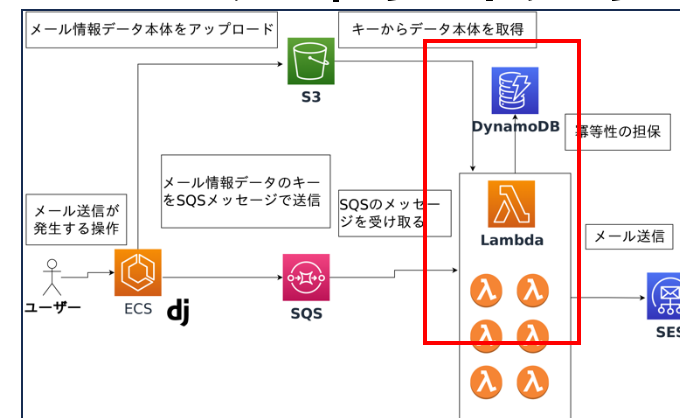
```
def is_duplicate_message(message_id):
    return dynamodb_client.query(
        TableName='ProcessedRecords',
        Select='COUNT',
        KeyConditionExpression='Records = :Records'
        ExpressionAttributeValues={
            ':Records': {'S': message_id}
        }
    )["Count"] != 0

# Processes the message body to upper case.
# @input string body
#
# @param body to be processed
# @return uppercase body
def process_message(body):
    return body.upper()

# Put the message to the DynamoDB Table.
# @input string batch_item_success
#
# @param batch_item_success of the message to put.
# @return Boolean
def push_to_dynamoDB(batch_item_success):
```

引用元: https://repost.aws/ja/knowledge-center/lambda-function-process-sqs-messages

BePROUD

Powertools for AWS Lambda(Python)

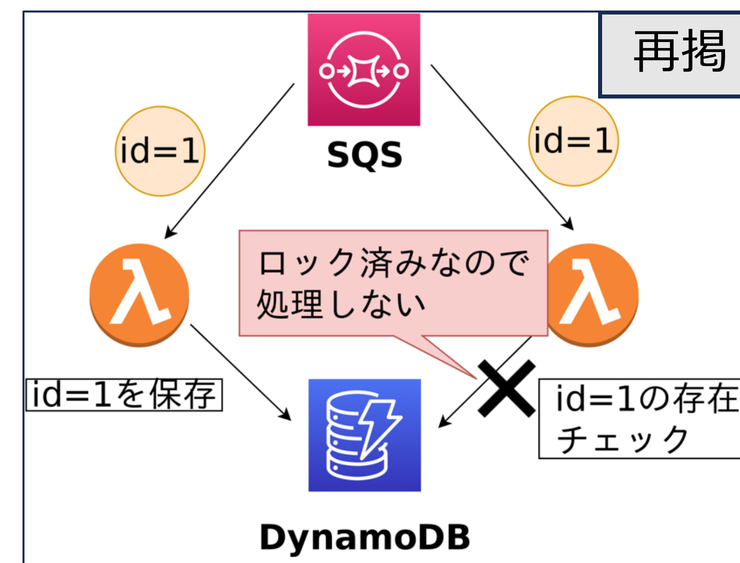
- **Powertools for AWS Lambda(Python)**というライブラリがある
  - ※ 以降**Powertools**と表記
- LambdaのPython実装用の便利ライブラリ集(SQSに限らない)
- SQSに限っても冪等性以外も色々な便利機能をサポートしている
  - エラーハンドリング、構造化ログ
- AWSの公式ドキュメントでも紹介されている3rdパーティライブラリ
- **pip install aws-lambda-powertools**



https://aws.amazon.com/jp/builders-flash/202203/lambda-powertools-python-1/

idempotency

- **Powertools のidempotency**という機能を使う
  - 読み方: アイデンポテンシー
- まさに英語で「冪等性」という意味
- DynamoDBによるロック処理を裏でやってくれる



Be PROUD

https://docs.powertools.aws.dev/lambda/python/latest/utilities/idempotency/

idempotencyが内部でやってること(実装レベル)

※ Powertoolsのコードの引用

```python
        condition_expression = (
            f"{idempotency_key_not_exist} OR {idempotency_expiry_expired} OR ({inprogress_expiry_expired})"
        )

        self.client.put_item(
            TableName=self.table_name,
            Item=item,
            ConditionExpression=condition_expression,
            ExpressionAttributeNames={
                "#id": self.key_attr,
                "#expiry": self.expiry_attr,
                "#in_progress_expiry": self.in_progress_expiry_attr,
                "#status": self.status_attr,
            },
            ExpressionAttributeValues={
                ":now": {"N": str(int(now.timestamp()))},
                ":now_in_millis": {"N": str(int(now.timestamp() * 1000))},
                ":inprogress": {"S": STATUS_CONSTANTS["INPROGRESS"]},
            },
            **self.return_value_on_condition,  # type: ignore
        )
    except ClientError as exc:
        error_code = exc.response.get("Error", {}).get("Code")
        if error_code == "ConditionalCheckFailedException":
            old_data_record = self._item_to_data_record(exc.response["Item"]) if "Item" in exc.response else None
            if old_data_record is not None:
                logger.debug(
                    f"Failed to put record for already existing idempotency key: "
```

やってること

1. ロック用のレコードを存在チェックする
2-.a.なかったらロック作成して処理実行
2-b.あったらログ出して処理スキップ

※ ちゃんとDynamoDBのコマンドは1つで完結させている👍
→ エッジケースも考慮した実装

引用元: https://github.com/aws-powertools/powertools-lambda-python/blob/43da9470929f09f4a8fb8648a23acbc57add99fa/aws_lambda_powertools/utilities/idempotency/persistence/dynamodb.py#L185

実際に作成されるロック用のレコード

```
{
        "expiration": {
            "N": "1711585286"
        },
        "id": {
            "S": "connpass-dev-send-mail.app.record_handler#4333322e911562c79fa79513a0830445"
        },
        "data": {

        },
        "in_progress_expirat
            "N": "17114997855
        },
        "status": {
            "S": "COMPLETED"
        }
    }
```

Lambda関数名

SQSのメッセージID

Lambda関数名をprefixにつけてくれるから関数を跨ってidが競合しない
→ 複数種類のLambda関数に対してテーブル1つの設計でもOK

BePROUD

Powertoolsの導入方法(簡単！)

Be PROUD

その前に

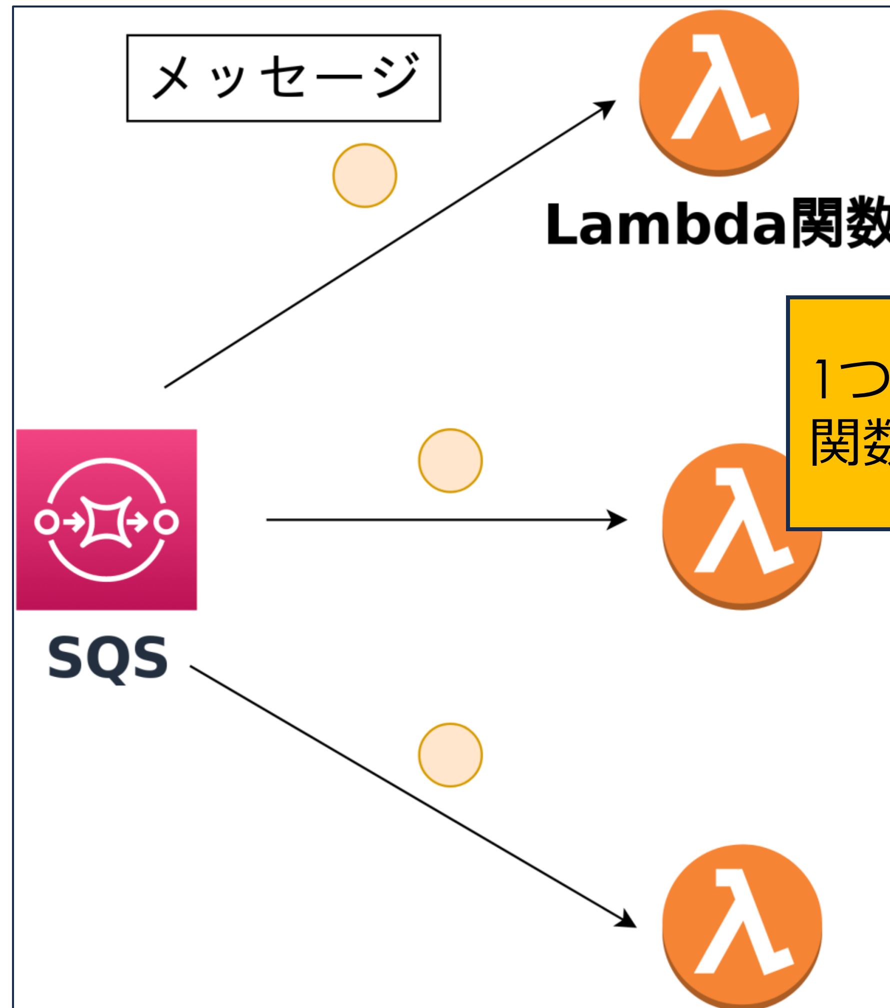
Be PROUD

Lambda x SQSの処理のイメージが前提知識として大事なので説明

Be PROUD

Lambda x SQSの処理の間違ったイメージ

間違ったイメージ



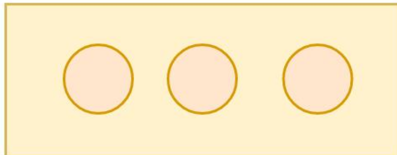
1つのメッセージを1つのLambda関数が個別に処理する

Lambda x SQSの処理の正しいイメージ

正しいイメージ

ポーリング

複数件取得



メッセージを1個ずつループで処理

```python
def lambda_handler(event, context):
    for record in event["Records"]:
        message = record["body"]
        # do something
```

BePROUD

改めてPowertoolsの導入方法(簡単！)

Be PROUD

素朴なLambda関数のコード

```python
def lambda_handler(event, context):
    for record in event["Records"]:
        s3_pointer: str = record["body"]
        message: str = sqs_client.retrieve_message_from_s3(s3_pointer)
```

SQSメッセージ1個分の処理を別の関数にする

```python
def record_handler(record):
    s3_pointer: str = record["body"]
    message: str = sqs_client.retrieve_message_from_s3(s3_pointer)

def lambda_handler(event, context):
    for record in event["Records"]:
        record_handler(record)
```

DynamoDBのテーブル名を設定
(自分で作る)

```python
# 省略(Powertoolsのimport分など)
persistence_layer = DynamoDBPersistenceLayer(table_name="idempotency-table")

@idempotent_function(data_keyword_argument="record", persistence_store=persistence_layer)
def record_handler(record):
    s3_pointer: str = record.body
    message: str = sqs_client.retrieve_message_from_s3(s3_pointer)


def lambda_handler(event, context):
    with processor(records=event["Records"], handler=record_handler):
        processor.process()
    return processor.response()
```

@idempotent_functionデコレータをSQSメッセージ1個分の処理の関数に付与

```python
# 省略(Powerto
persistence_la

@idempotent_function(data_keyword_argument="record", persistence_store=persistence_layer)
def record_handler(record):
    s3_pointer: str = record.body
    message: str = sqs_client.retrieve_message_from_s3(s3_pointer)


def lambda_handler(event, context):
    with processor(records=event["Records"], handler=record_handler):
        processor.process()
    return processor.response()
```

```python
# 省略(Powertoolsのimport分など)
persistence_layer = DynamoDBPersistenceLayer(table_name="idempotency-table")

@idempotent_function(data_keyword_argument="record", persistence_store=persistence_layer)
def record_handler(record):
    s3_pointer: str = rec
    message: str = sqs_cl


def lambda_handler(event, context):
    with processor(records=event["Records"], handler=record_handler):
        processor.process()
    return processor.response()
```

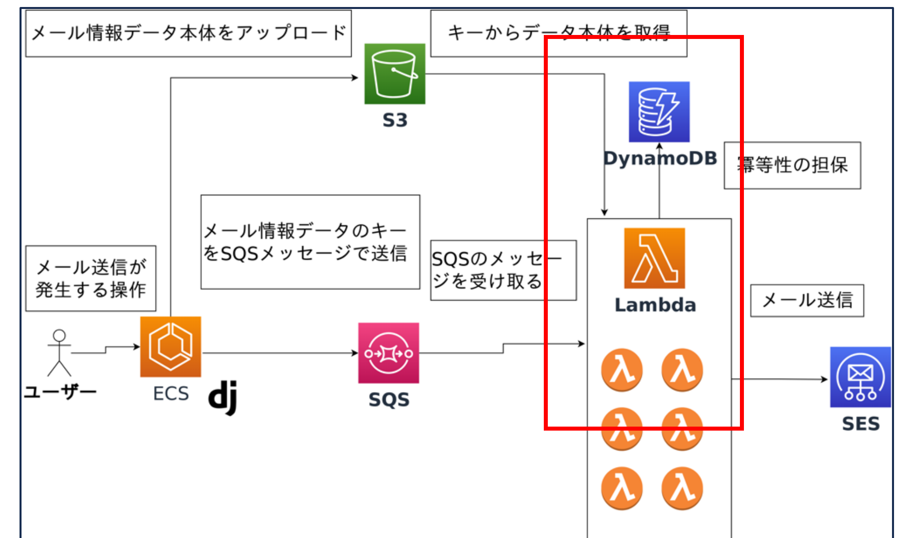
for文を**with processor...という**Powertools特有の書き方に変える

導入完了🎉

これだけでSQSメッセージ単位での処理が冪等に

【まとめ】3-2.冪等性の担保

- Lambda x SQSは同じメッセージが多重実行されてしまう可能性がある(=冪等性が担保されていない)
- 従って冪等性は実装する側が担保する必要がある
- **Powertools**の**idempotency**を使うとコーディングレスに実現できるのでオススメ

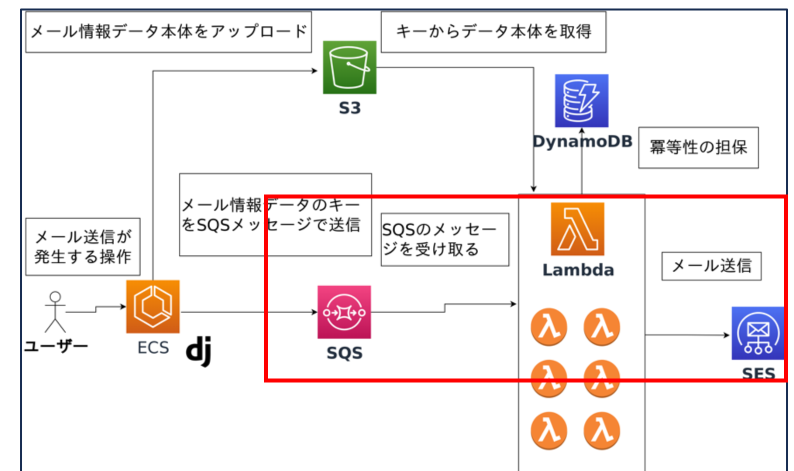


お品書き

1. connpassのメール送信の課題
2. AWS Lambda x SQSの導入による課題解決
3. AWS Lambda x SQSのプロダクションレベルの知見
    1. 前提: SQS x Lambdaの処理のイメージ
    2. SQSの256KB上限問題(データサイズ)
    3. 冪等性の担保
    4. **エラーハンドリング**

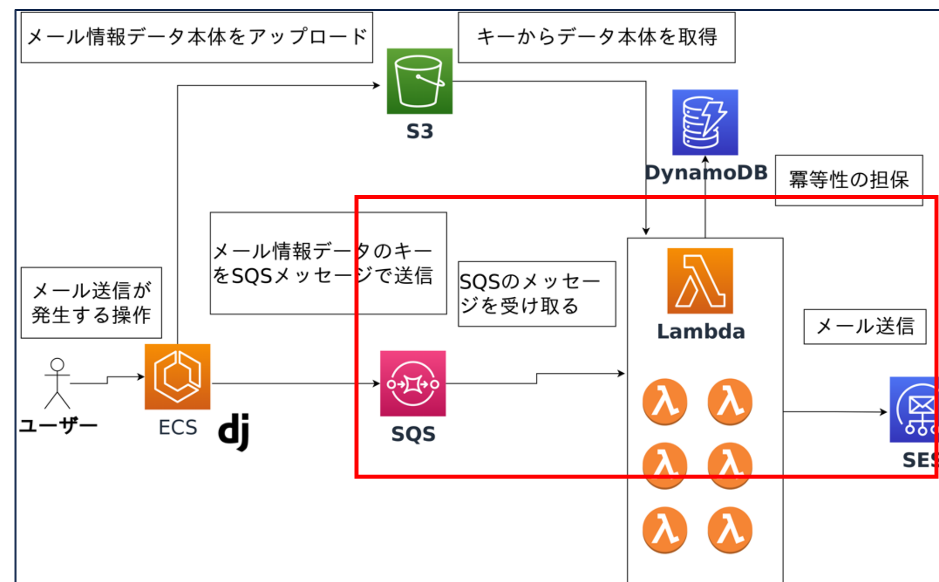
BePROUD

エラーハンドリング



前提

プロダクションレベルのシステム: 処理に失敗した場合はリトライしたい



【再掲】Lambda x SQSの処理の正しいイメージ

正しいイメージ

ポーリング

複数件取得

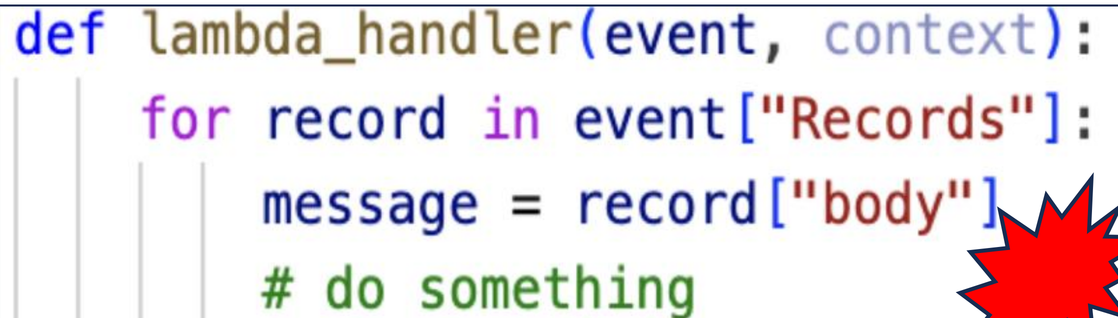
メッセージを1個ずつループで処理

```
def lambda_handler(event, context):
    for record in event["Records"]:
        message = record["body"]
        # do something
```

ループ中にエラーが発生したら？

考えられる挙動

1. 処理が異常終了して後続のメッセージは処理されない
2. 成功したのも含めて全メッセージ再処理される
3. 失敗したメッセージだけ再処理される

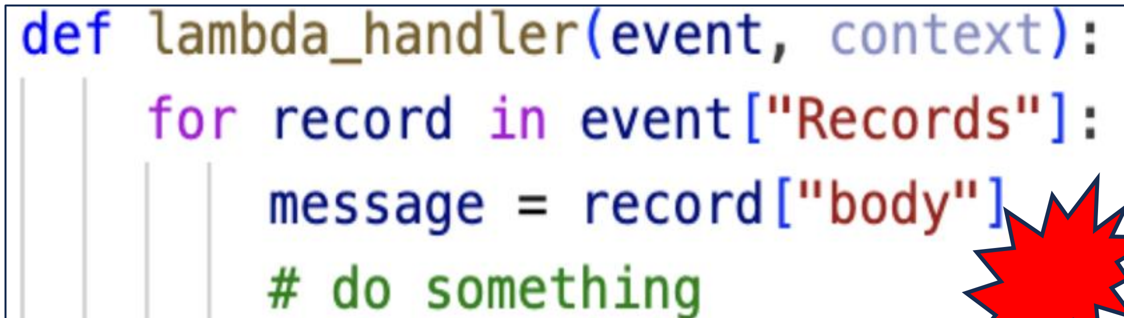

```
def lambda_handler(event, context):
    for record in event["Records"]:
        message = record["body"]
        # do something
```

Be PROUD

AWSのデフォルトの挙動は2番目

考えられる挙動

1. 処理が異常終了して後続のメッセージは処理されない
2. **成功したのも含めて全メッセージ再処理される**
3. 失敗したメッセージだけ再処理される



```
def lambda_handler(event, context):
    for record in event["Records"]:
        message = record["body"]
        # do something
```

https://repost.aws/ja/knowledge-center/lambda-retrying-valid-sqs-messages

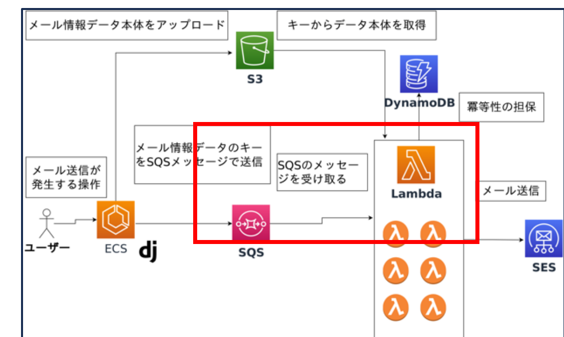
2.の仕様の良い点と悪い点

良い点

- リトライはしてくれるので**1.**よりはgood

悪い点

- 成功したメッセージまで再処理する
  - 「冪等性の担保」をしてない場合: 処理が多重実行されてしまう🔥
  - 担保してる場合: 多重実行はされないがロックに引っかかるだけの無駄な処理が発生して処理効率が下がる😢



BePROUD

2.の仕様の良い点と悪い点

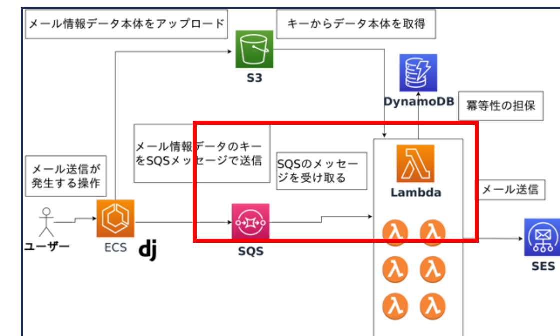
良い点

- リトライはしてくれるので**1.**よりはgood

悪い点

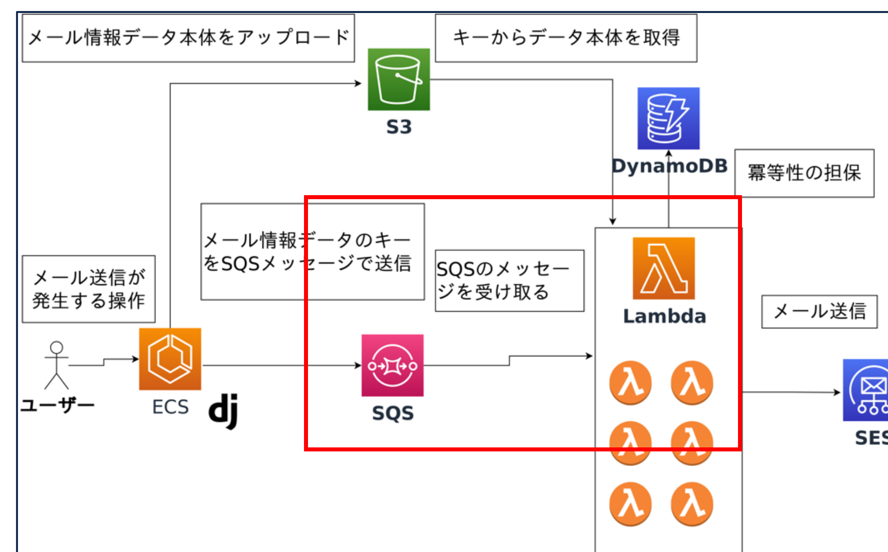
- 成功したメッセージまで再処理する
  - 「冪等性の担保」をしてない場合: 処理が多重実行されてしまう🔥
  - 担保してる場合: 多重実行はされないがロックに引っかかるだけの無駄な処理が発生して処理効率が下がる😢

→ 理想は「3.失敗したメッセージだけ再処理される」



batchItemFailures

- Lambda x SQS用の失敗したメッセージのみをリトライする仕組み
- 2021年11月リリースの比較的新しい機能

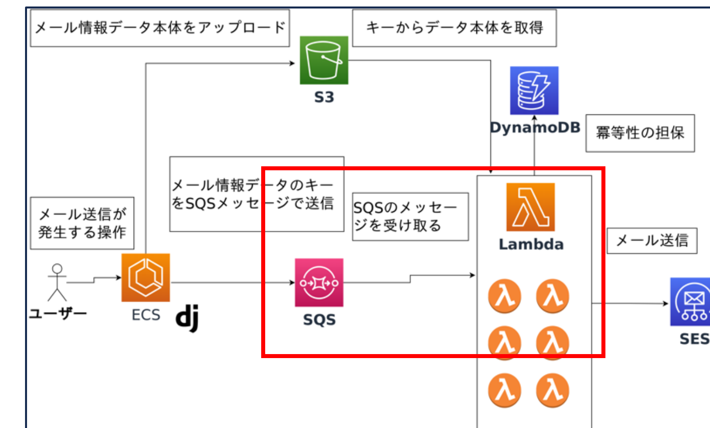


https://docs.aws.amazon.com/lambda/latest/dg/services-sqs-errorhandling.html#services-sqs-batchfailurereporting

batchItemFailuresの仕様

- Lambda関数の戻り値で、処理に失敗したSQSメッセージのIDのリストを以下の形式でreturnする
- → 返したidのメッセージだけキューに戻される(=リトライされる)

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```



https://docs.aws.amazon.com/lambda/latest/dg/services-sqs-errorhandling.html#services-sqs-batchfailurereporting

ネストが深くなる...

実装自体は難しくはないが素直に実装するとネストが深くなり可読性が下がる

```
def lambda_handler(event, context):
    batch_item_failures = []
    for record in event["Records"]:
        try:
            # do something
            pass
        except Exception:
            batch_item_failures.append({"itemIdentifier": record["messageId"]})
    return {"batchItemFailures": batch_item_failures}
```

BePROUD

ここでもPowertoolsが便利

SQSメッセージ1個分を処理する関数の中で例外が送出されると
Powertoolsが自動で**batchItemFailures**の形式で返却してくれる

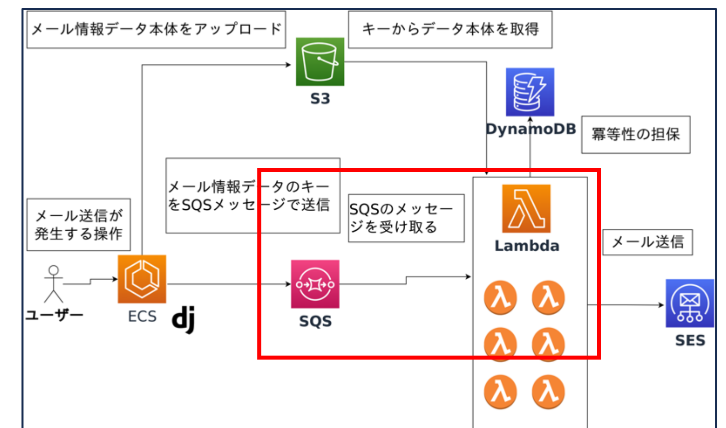
```
@idempotent_function(data_keyword_argument="record", persistence_store=persistence_layer)
def record_handler(record):
    s3_pointer: str = record.body
    message: str = sqs_client.retrieve_message_from_s3(s3_pointer)

def lambda_handler(event, context):
    with processor(records=event["Records"], handler=record_handler):
        processor.process()
    return processor.response()
```

この中で送出された
例外が対象

Be PROUD

https://docs.powertools.aws.dev/lambda/python/latest/utilities/batch/#partial-failure-mechanics

公式ドキュメント通りにPowertoolsを使っていればbatchItemFailuresのための特別な実装は不要🎉



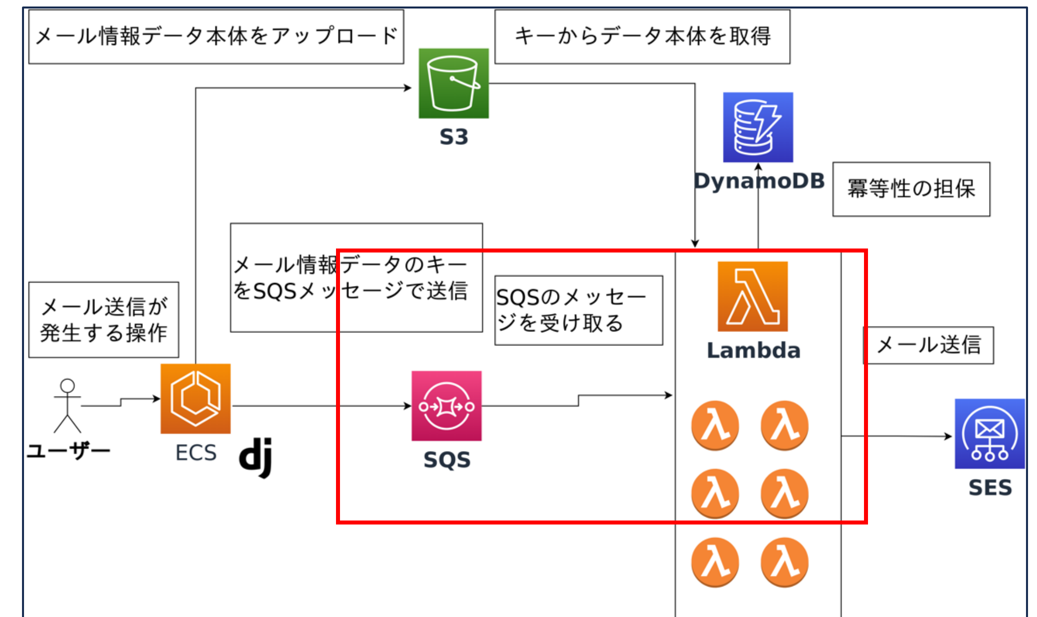
Appendix

今回触れられなかったがエラーハンドリング関連で重要なトピック

- **デッドレターキュー**
  - リトライ回数に関わる
  - リトライで救えないエラーのハンドリング(↔ 偶発的なネットワークエラーなど)
- **可視性タイムアウト(visibility timeout)**
  - リトライのインターバルに関わる
- **Lambda関数のタイムアウト**
  - 可視性タイムアウトとセットでリトライのインターバルに関わる
- **Lambda関数のロギング・モニタリング**
  - LambdaもWEBアプリケーション同様にエラーログの収集・監視が必要

【まとめ】3-3.エラーハンドリング

- Lambda x SQSには**batchItemFailures**という処理に失敗したメッセージだけリトライできる機能がある
- **Powertools**を導入すればほぼコーディングゼロで**batchItemFailures**を実現できるのでオススメ



Be PROUD

発表全体のまとめ

- Lambda x SQSの構成はスケーラビリティに優れていて、非同期処理の選択肢として非常に有力
- 但しプロダクションレベルでの導入には細かい考慮事項が色々ある
  - → 自前で対応すると意外と時間と労力がかかりそう
- LambdaのPython実装はライブラリが充実している
  - **amazon-sqs-python-extended-client-lib**
  - **Powertools for AWS Lambda(Python)**
- → 活用して非機能要件の対応はコーディングレスに済ませよう
- → サービスにとって本質的な部分の実装に集中できる😁

BePROUD