Sansanがメッセージング (SQS) でスケーラビリティを手に入れた話

using C# on Windows;

2017-05-31
AWS Dev Day Tokyo 2017

神原 淳史
@atsukanrock

自己紹介

- 神原 淳史 **@atsukanrock** 
- Development Manager at Sansan
- Interested in
  - Domain-Driven Design
  - C# / .NET
  - Enterprise Integration Patterns

目次

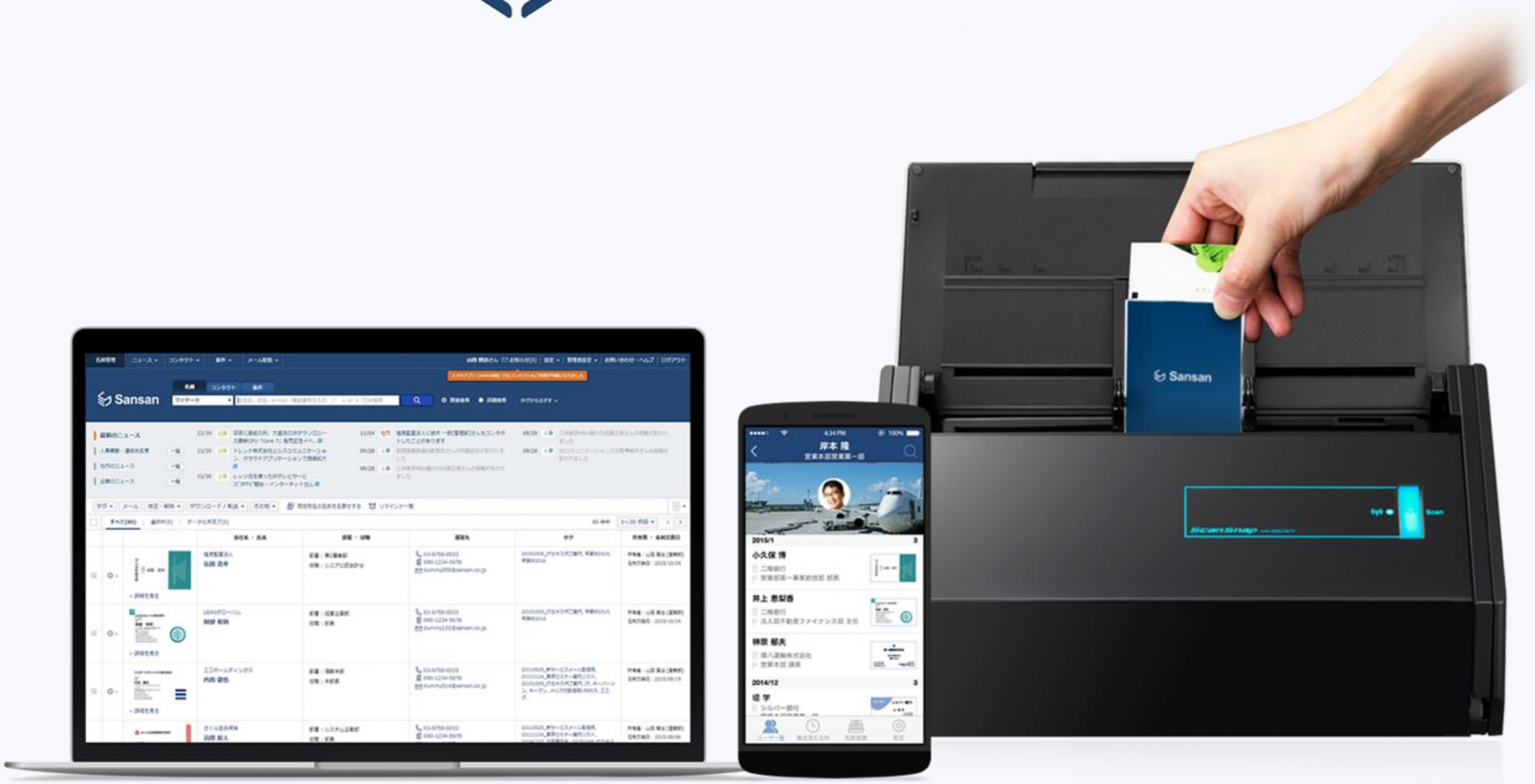
- Sansanサービスの説明
- メッセージングとは
- メッセージングによる課題解決の実例
- メッセージングの注意点
- まとめ

Sansanサービスの説明

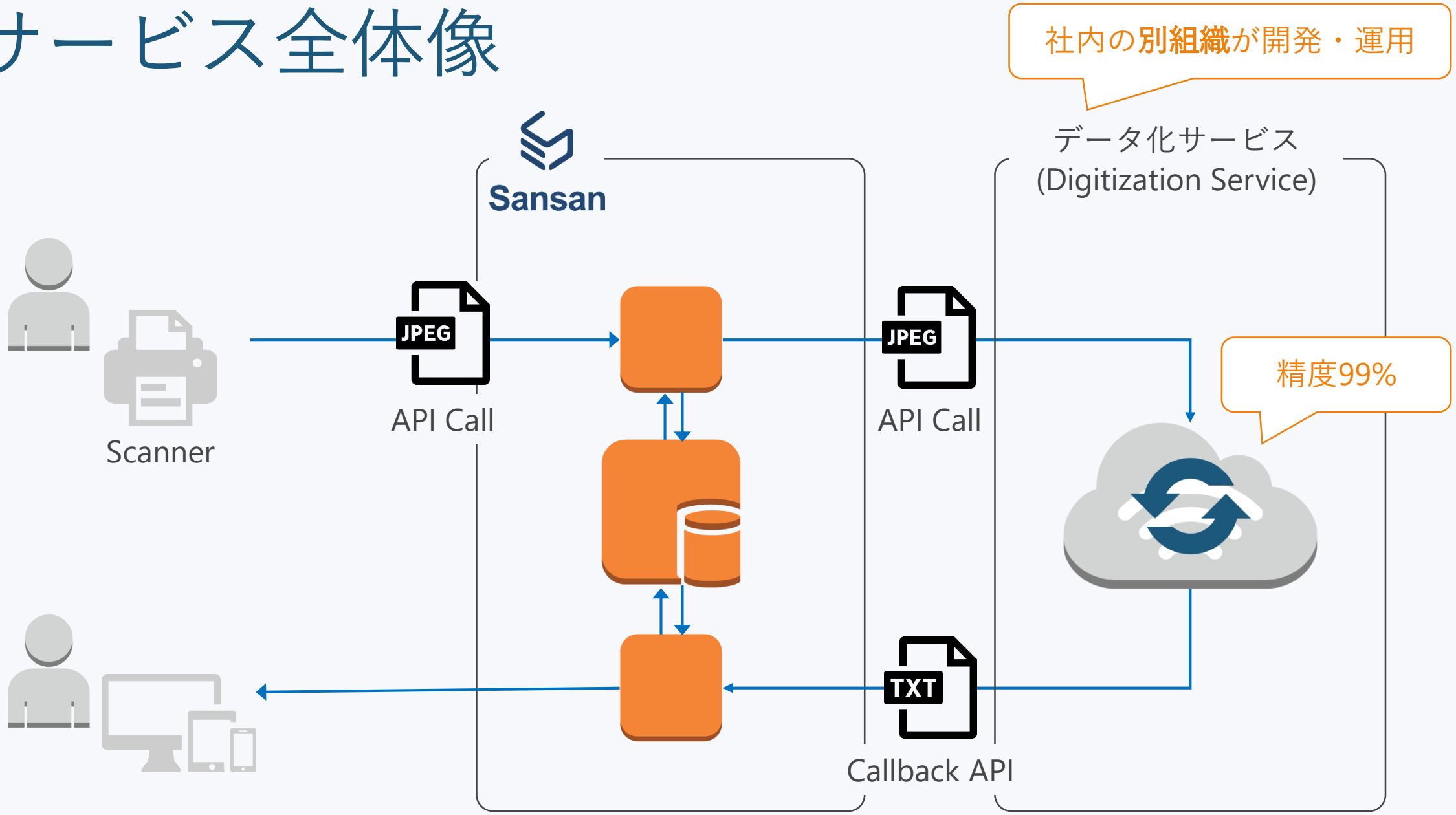
名刺を企業の資産に変える

名刺を企業の資産に変える

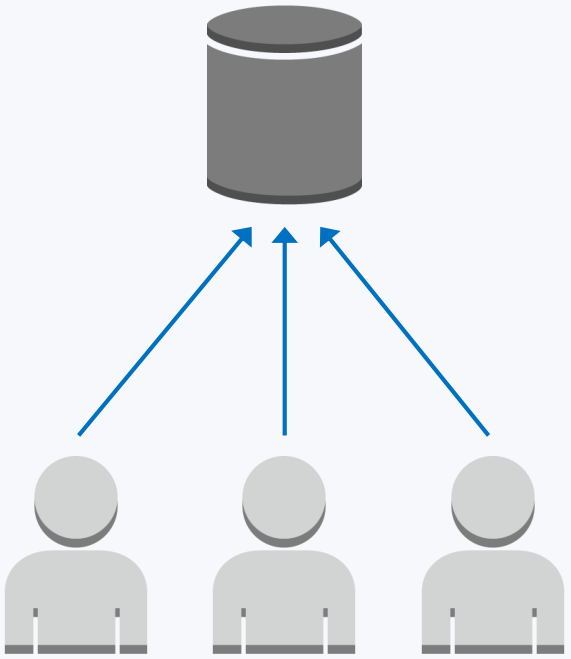
Sansan



サービス全体像


社内の別組織が開発・運用
Sansan
データ化サービス
(Digitization Service)
JPEG
API Call
JPEG
API Call
精度99%
Scanner
TXT
Callback API

マルチテナント型クラウドサービス

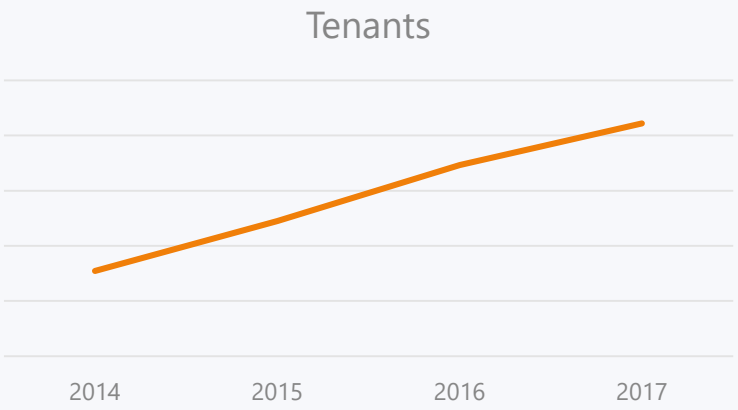


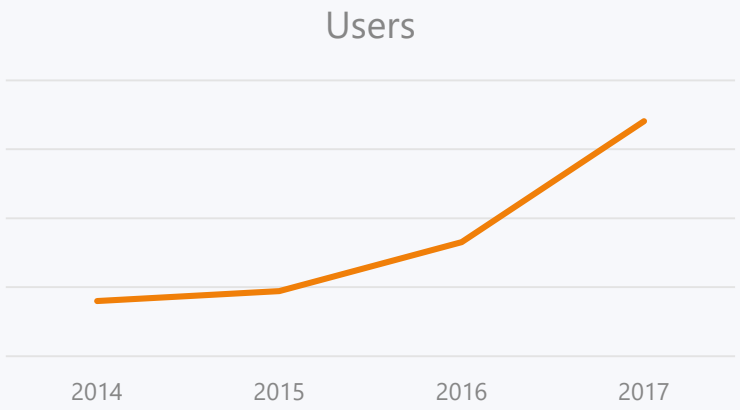
tenant_id	biz_card_id	given_name	family_name	email	...
xxx	11111	太郎	山田	...	...
xxx	22222	二郎	鈴木	...	...
xxx	33333	三郎	佐藤	...	...
xxx	44444	四郎	田中	...	...
yyy	55555	五郎	高橋	...	...
yyy	66666	六郎	山本	...	...
yyy	77777	花子	中村	...	...

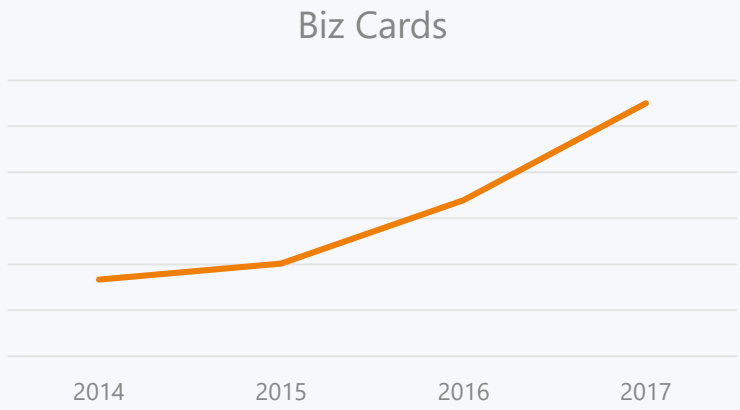
テナント (≒組織) 内で
データ共有

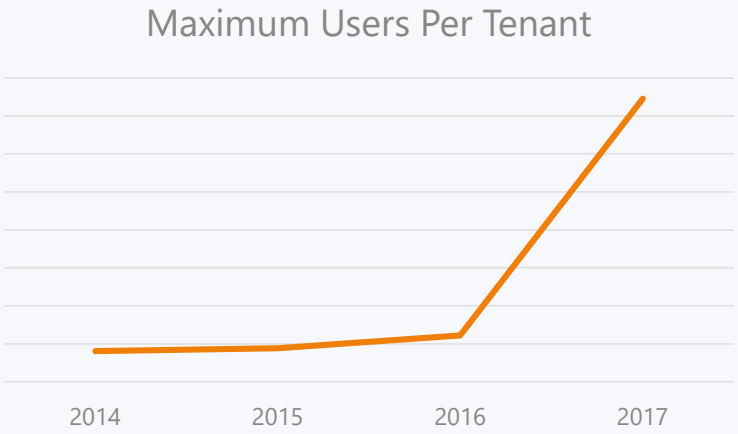
DBテーブルをテナント毎に分割

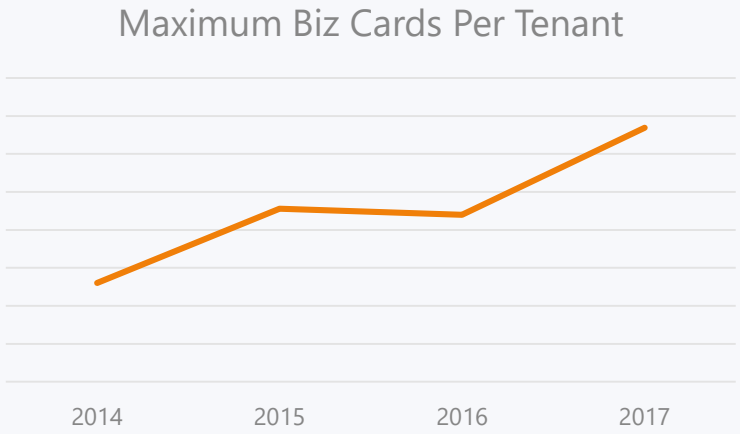
サービス規模の拡大











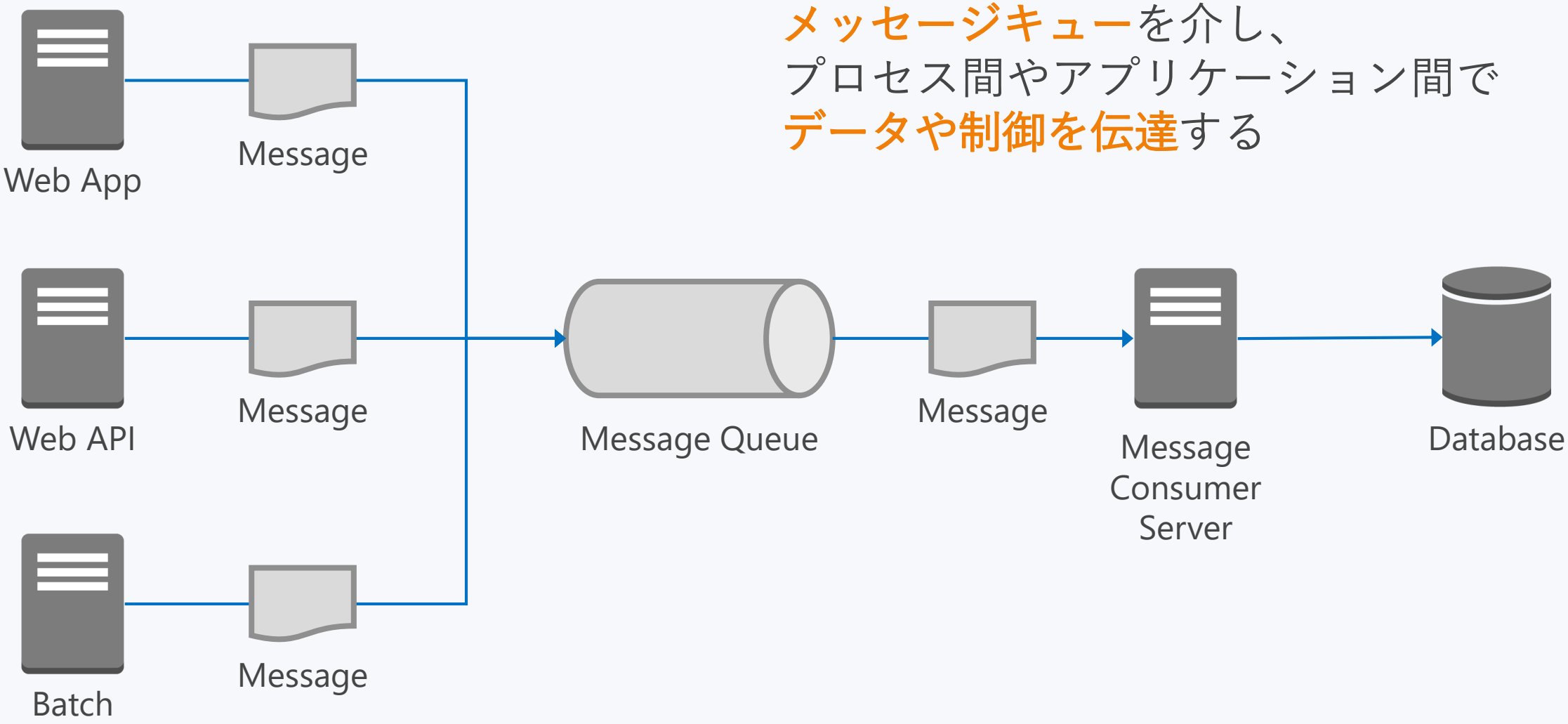
スケーラビリティの
課題が次々に発生

メッセージングとは

前提知識を早巡り

メッセージングとは

メッセージキューを介し、
プロセス間やアプリケーション間で
データや制御を伝達する



特徴

- メッセージは**単なるテキスト**
- 処理失敗したメッセージは**リトライ**される
  - 厳密には、再度処理対象となる
- 一定回数失敗し続けたメッセージは**別キューに退避**される (Dead Letter Queue)

ミドルウェア

	Amazon SQS Standard Queue	Amazon SQS FIFO Queue	Azure Storage	Azure Service Bus	MSMQ
PaaS	Yes	Yes	Yes	Yes	No
Transaction	No	No	No	Local	Distributed
At-most-once	No	Yes	No	Yes	Yes
FIFO	Best effort	Yes	Best effort	Yes (w/ session)	Yes

現Sansanで採用

メッセージングによる課題解決の実例

スケーラビリティを手に入れる

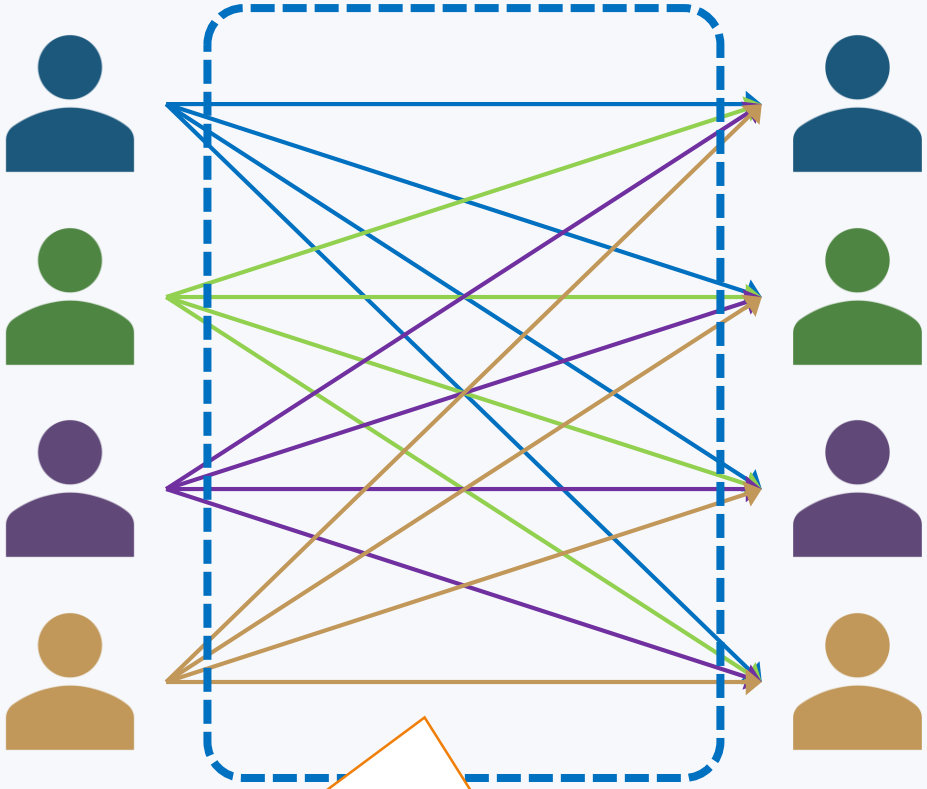
抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

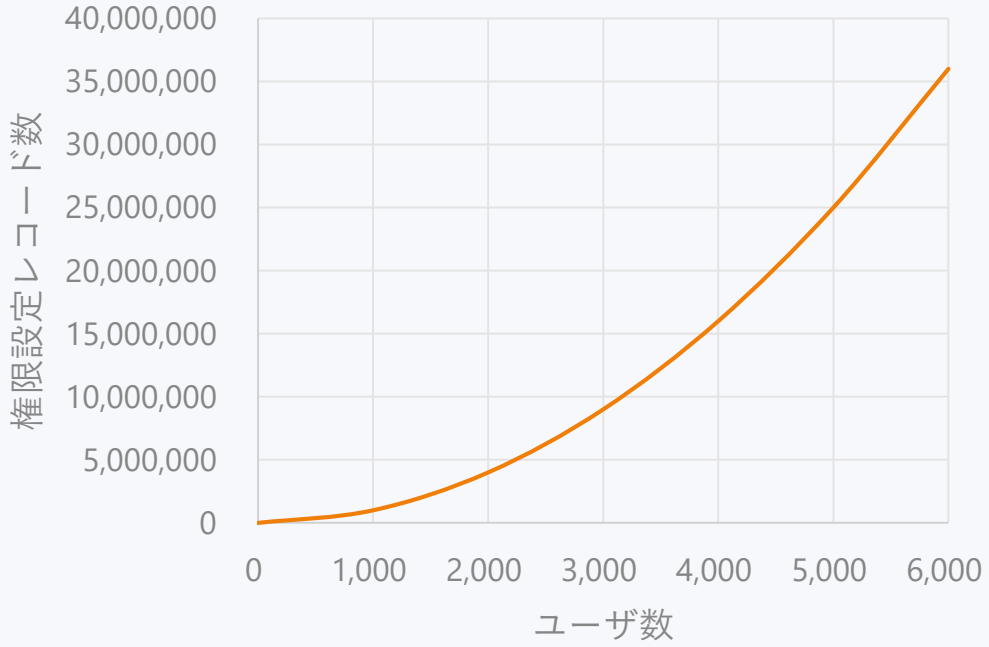
抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

巨大なトランザクション

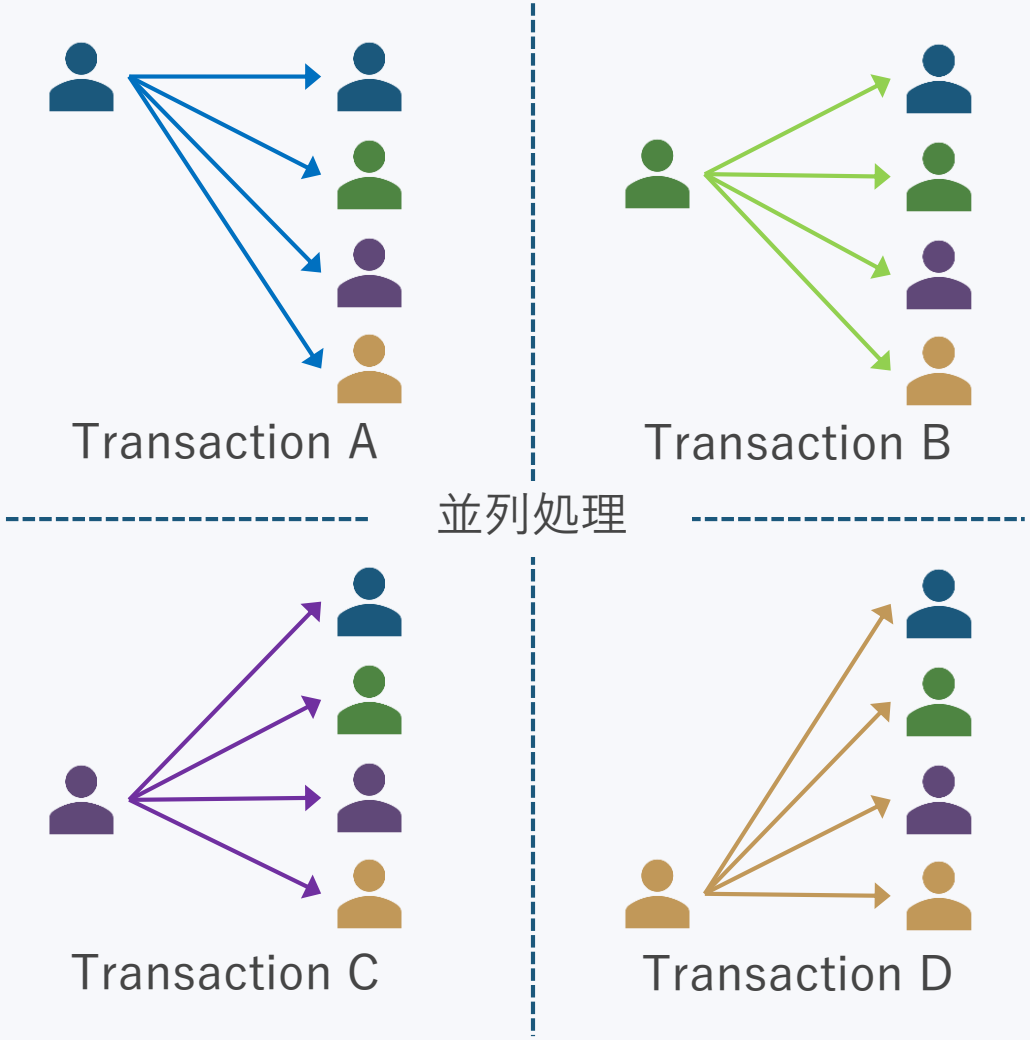


所有名刺の参照・更新可否を
ユーザ単位で設定可能



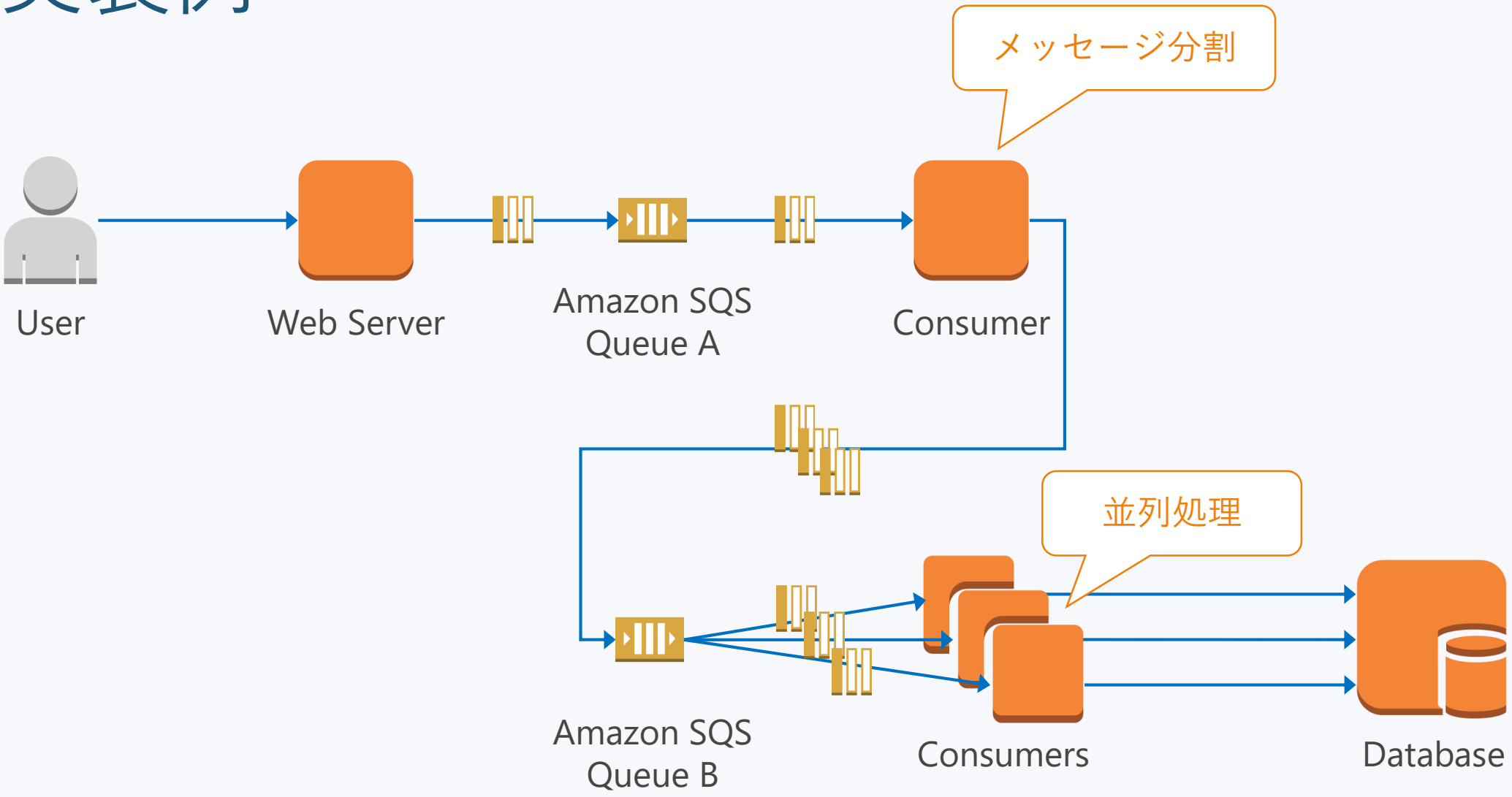
権限設定レコードの洗い替え処理を
1トランザクションで処理すると、
指数関数的にトランザクションが巨大化

Scalable Design



トータルで処理するレコード数は変わらないが、**並列処理によりスループットが向上**

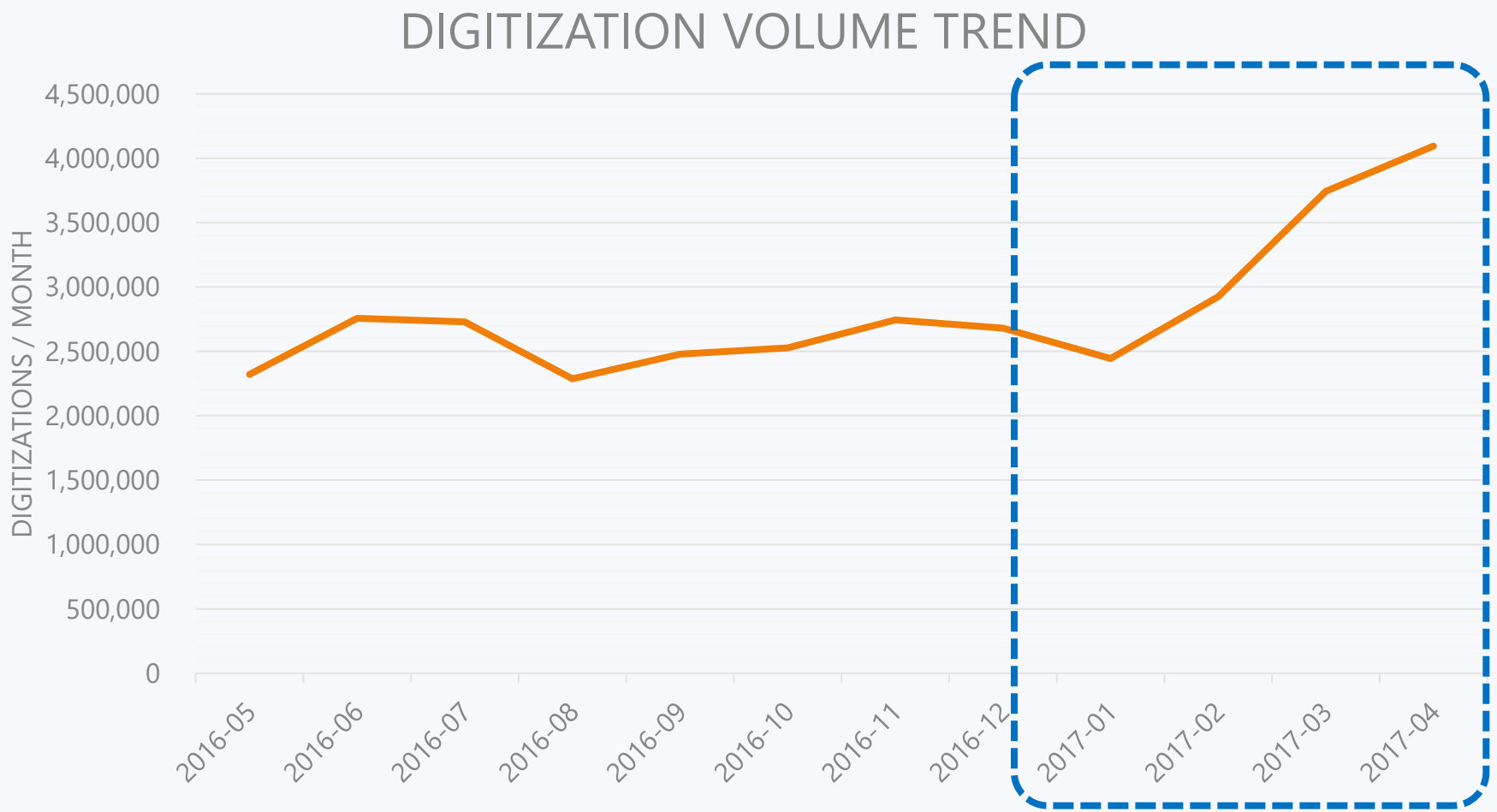
実装例


メッセージ分割
User
Web Server
Amazon SQS
Queue A
Consumer
並列処理
Amazon SQS
Queue B
Consumers
Database

抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

終わらないバッチ処理

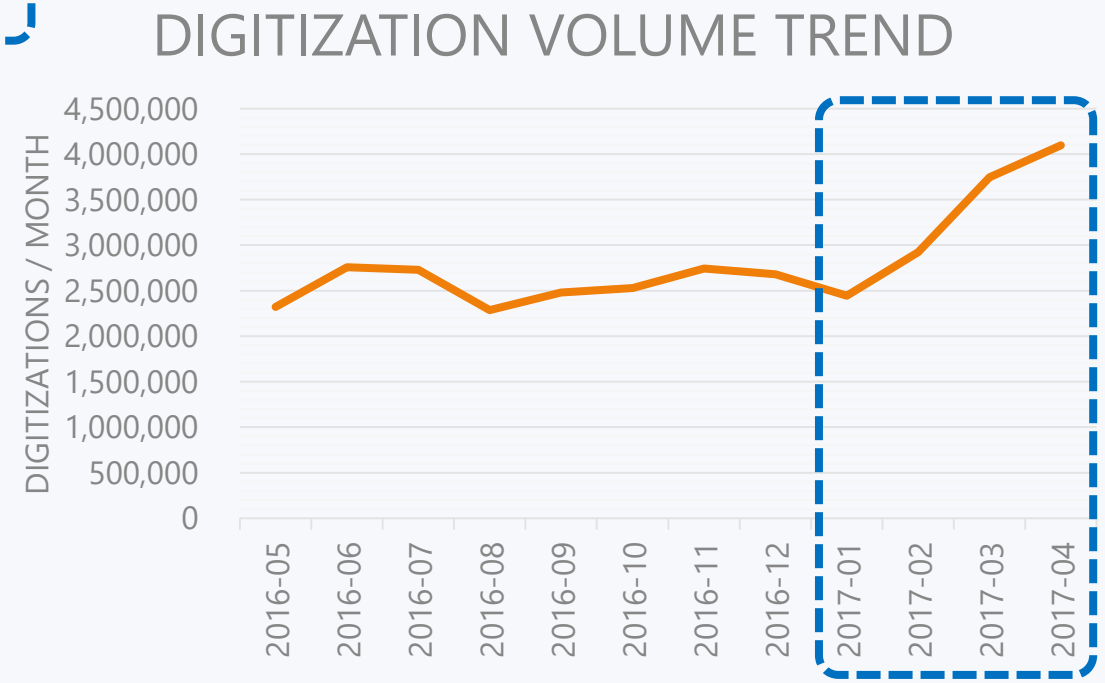

DIGITIZATION VOLUME TREND
DIGITIZATIONS / MONTH
4,500,000
4,000,000
3,500,000
3,000,000
2,500,000
2,000,000
1,500,000
1,000,000
500,000
0
2016-05
2016-06
2016-07
2016-08
2016-09
2016-10
2016-11
2016-12
2017-01
2017-02
2017-03
2017-04

Non-scalable Design

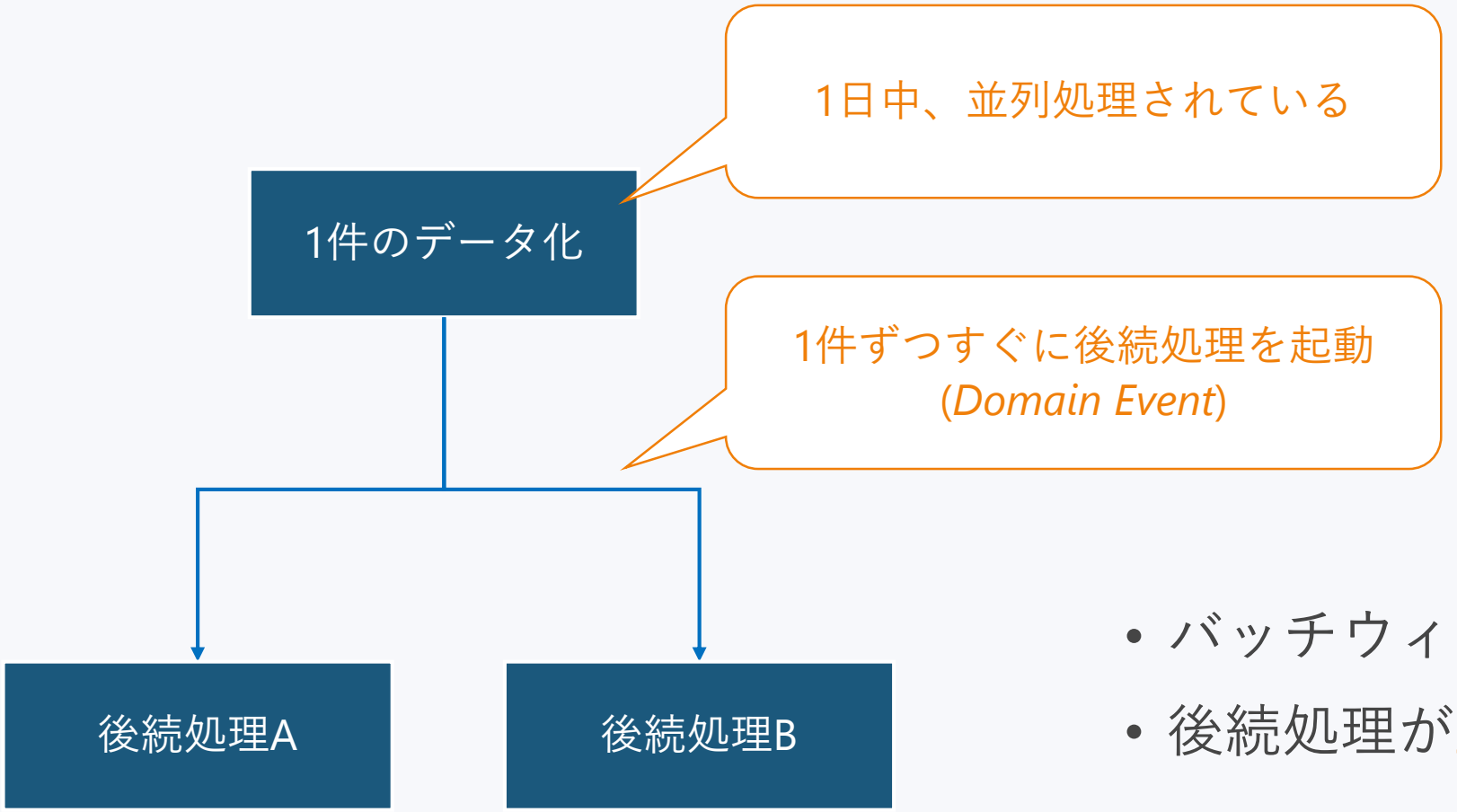
```
var targetBizCards = GetBizCardsByDigitizedTimestamp(from, to);
```

```
foreach (var targetBizCard in targetBizCards)
{
    DoSomething(targetBizCard);
}
```

処理時間がデータ化の量に比例



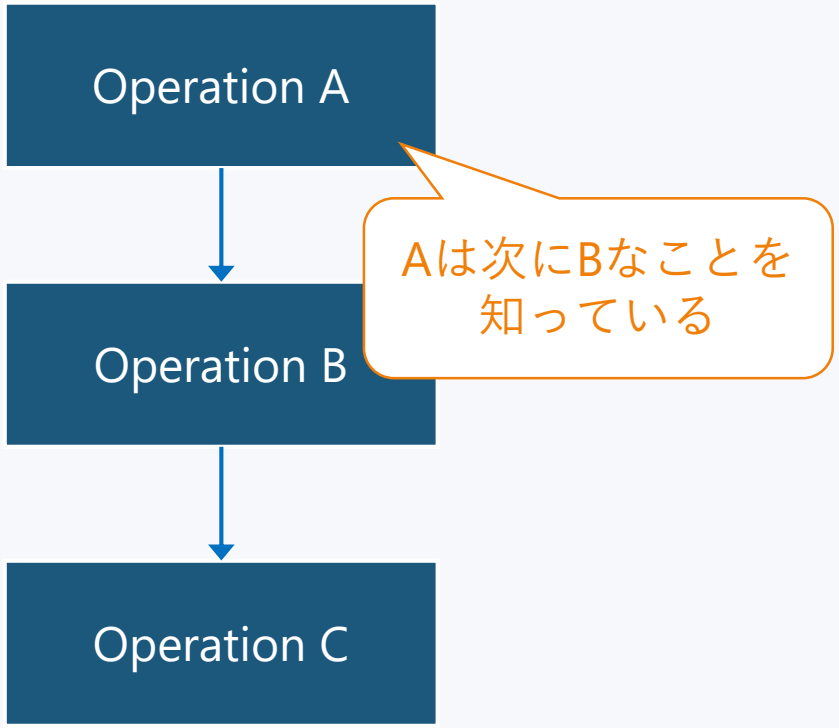
Scalable Design



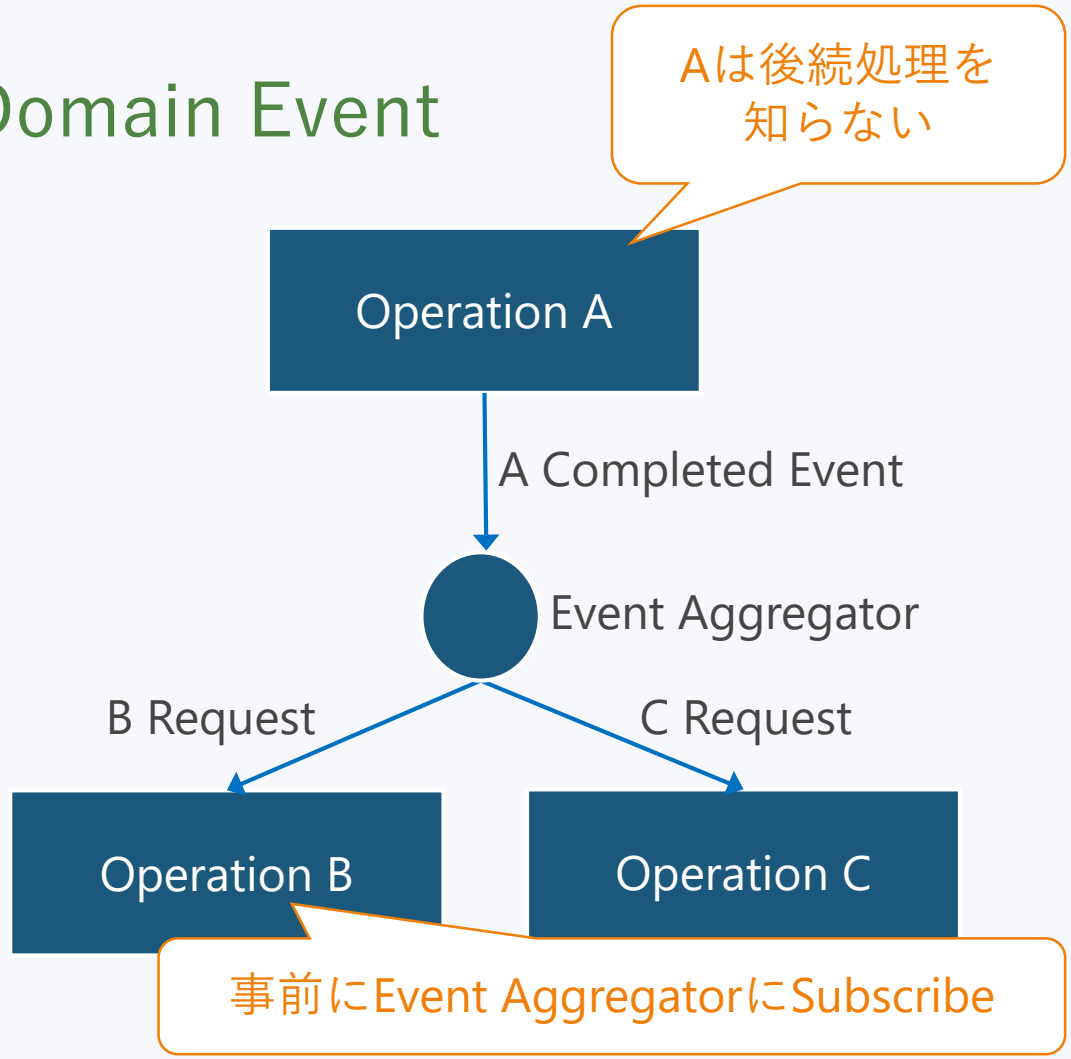
- バッチウィンドウが**1日中**に
- 後続処理が**並列化**

Domain Eventとは

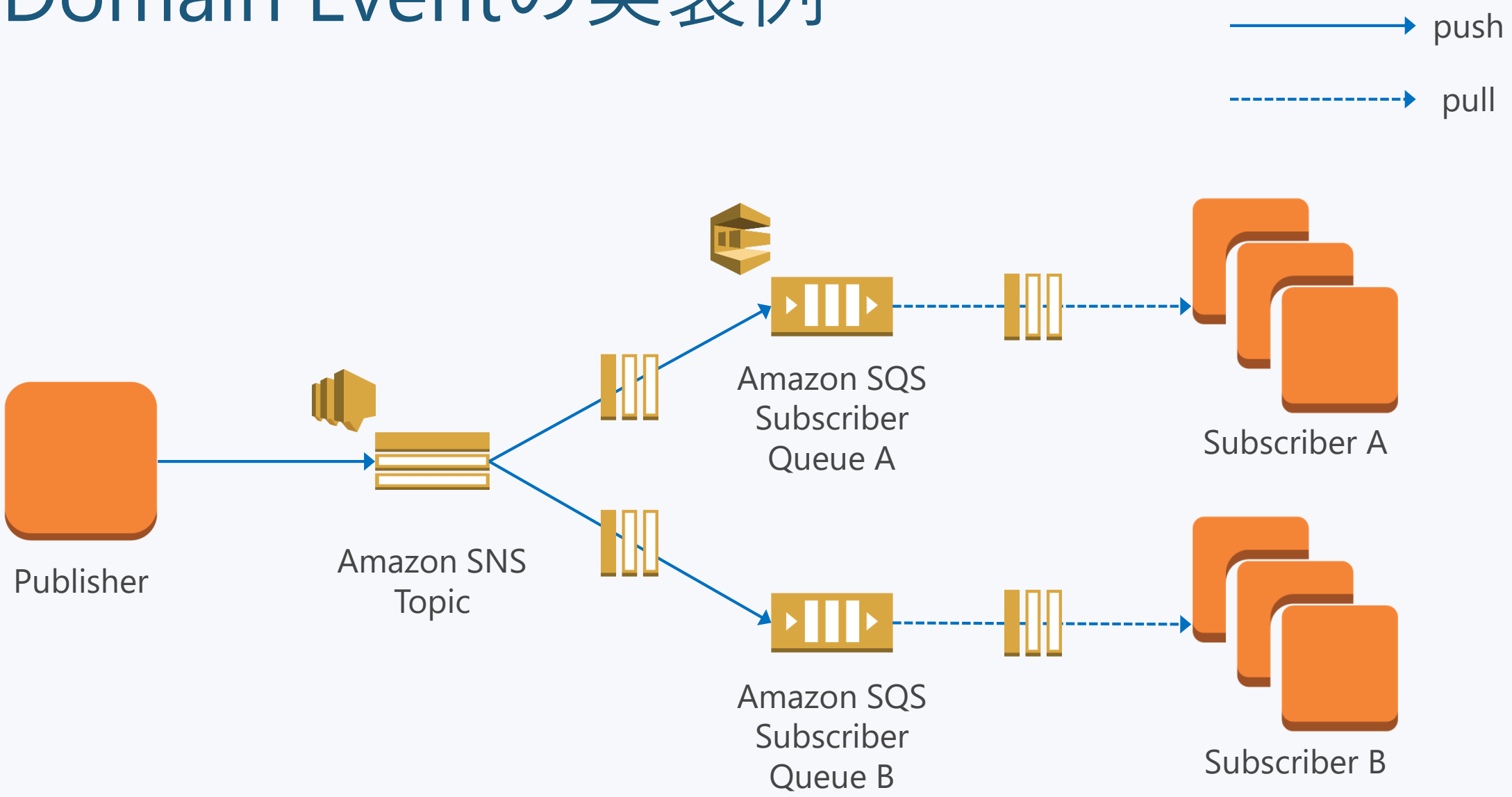
NOT Domain Event



Domain Event



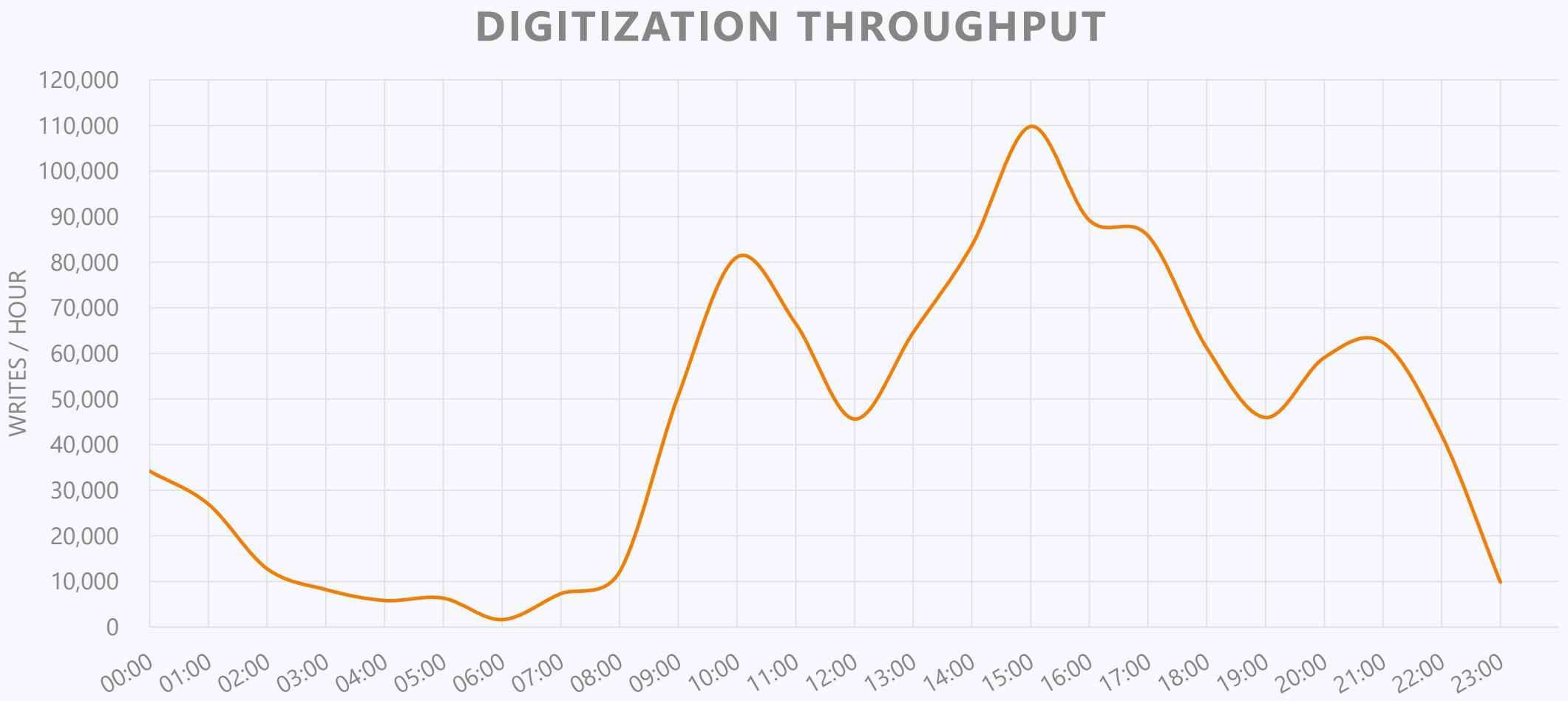
Domain Eventの実装例



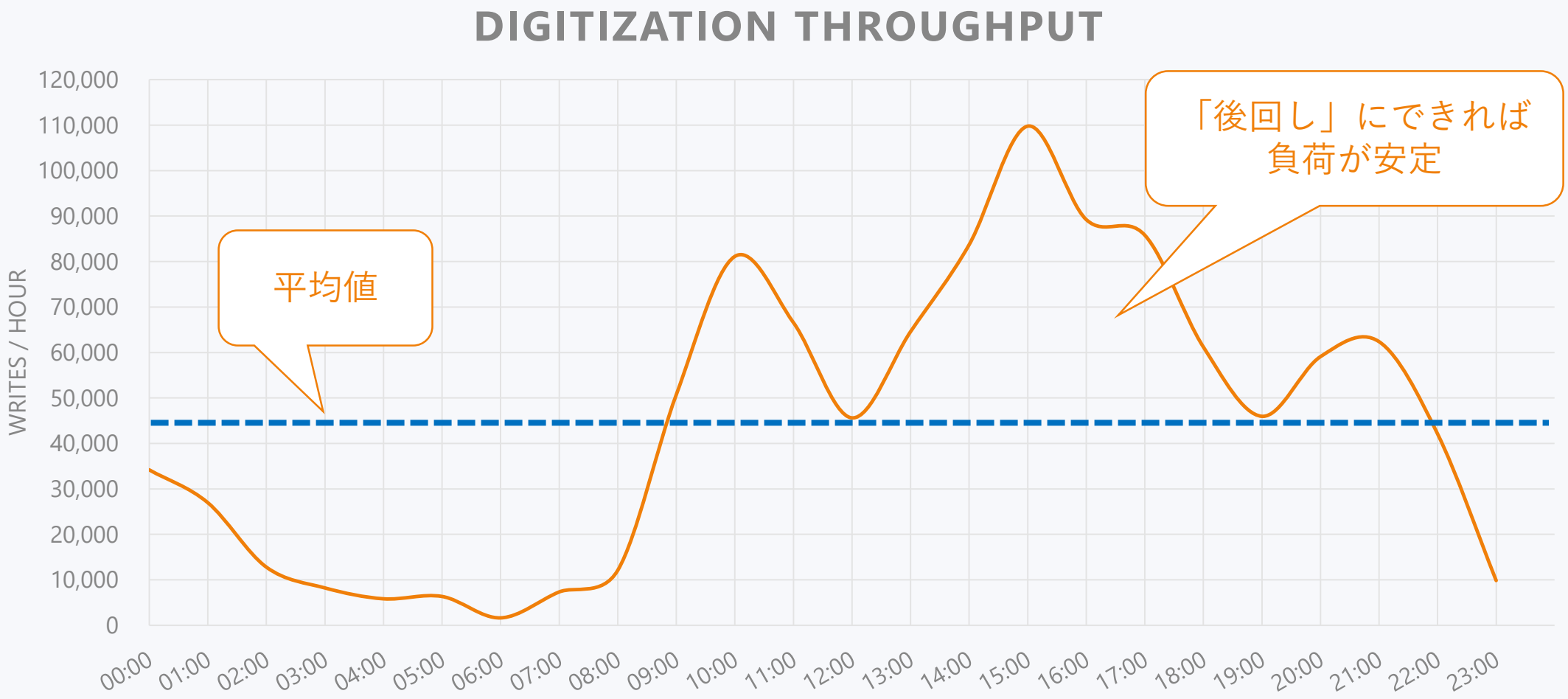
抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

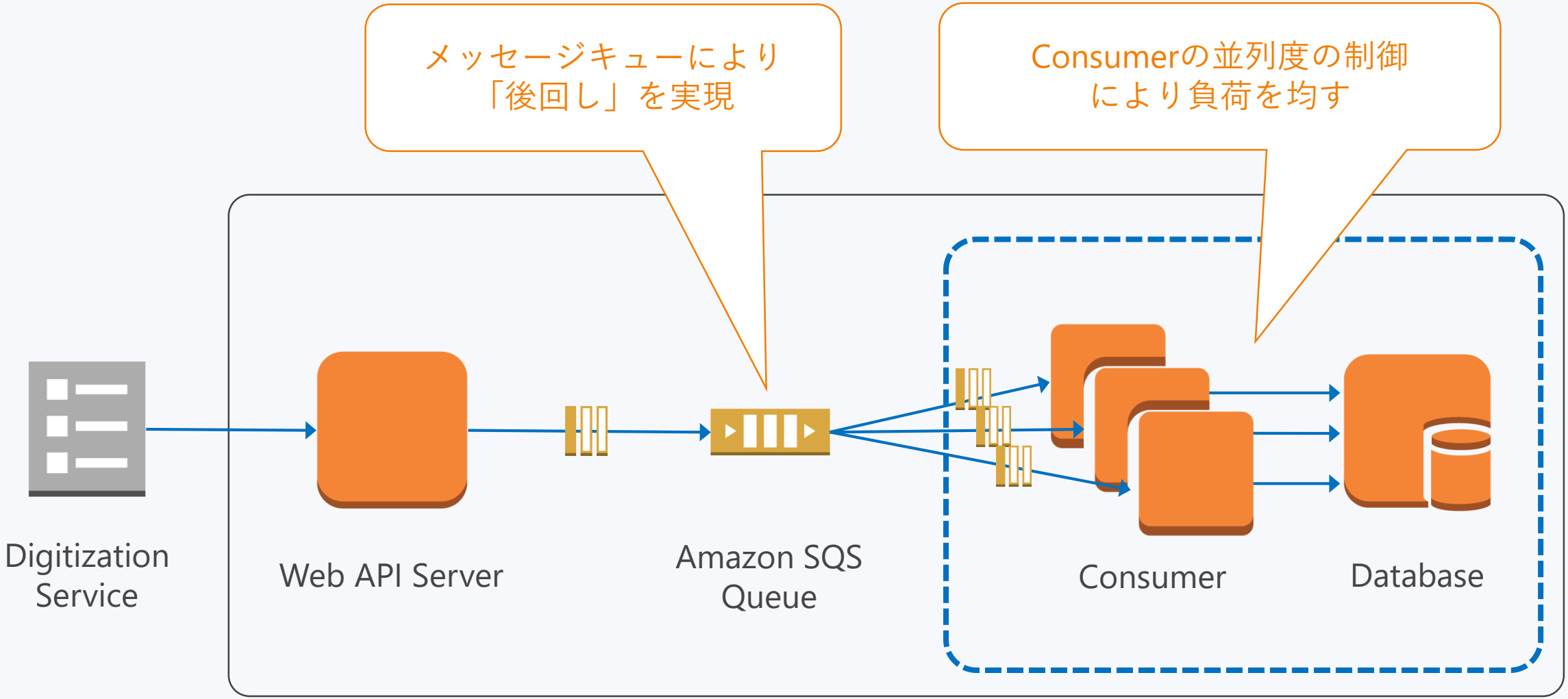
急激に変化するデータベース負荷



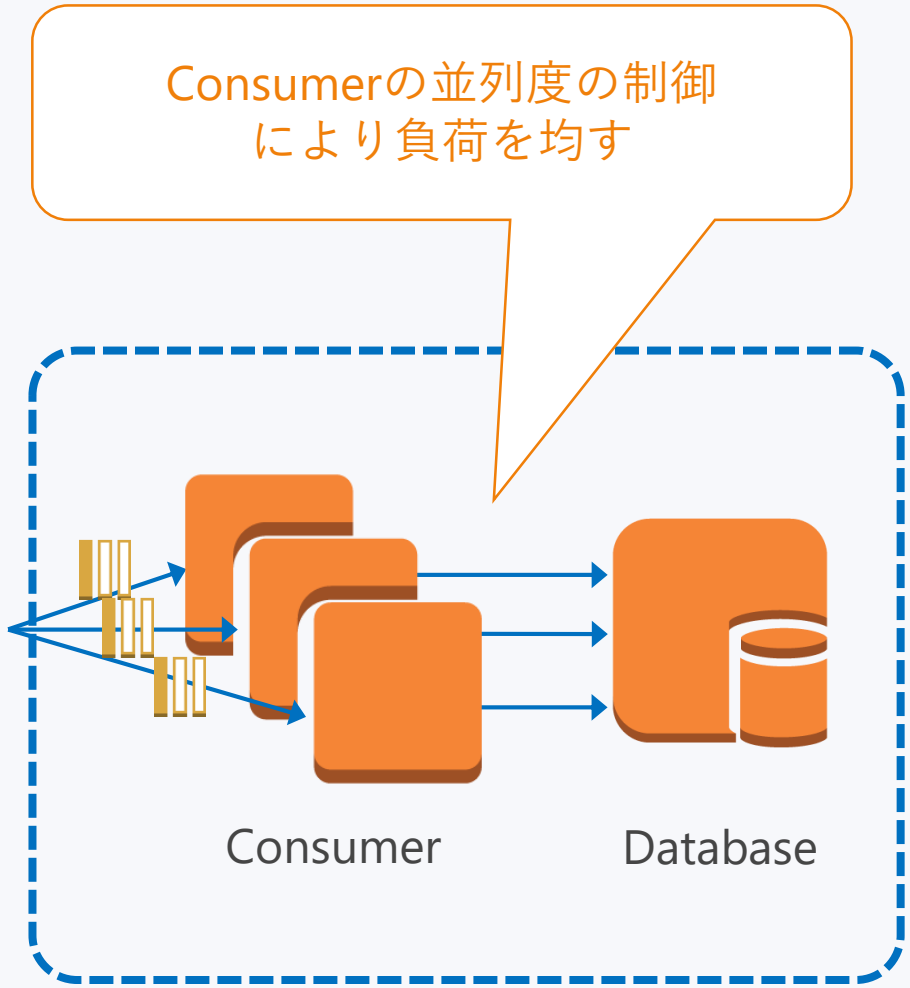
データベース負荷を安定させる



データ化処理のアーキテクチャ



並列度の制御



- インスタンス並列度
  - **Auto Scaling**等を利用すれば手軽
- スレッド並列度
  - .NETであれば**SemaphoreSlim**クラス等を利用して自前実装

```
while (true)
{
    IDisposable throttlingToken = null;

    try
    {
        throttlingToken = await ThrottlingPolicy.EnterAsync();

        var message = await MessageChannel.ReceiveAsync();

        if (message == null)
        {
            throttlingToken.Dispose();

            var interval = IdleRetryPolicy.GetRetryInterval(++idleCount);
            await Task.Delay(interval, cancellationToken);

            continue;
        }

        idleCount = 0;

        Task.Run(async () =>
```

SemaphoreSlimクラス等を
利用してスレッド並列度を制御

Channelからメッセージを取得

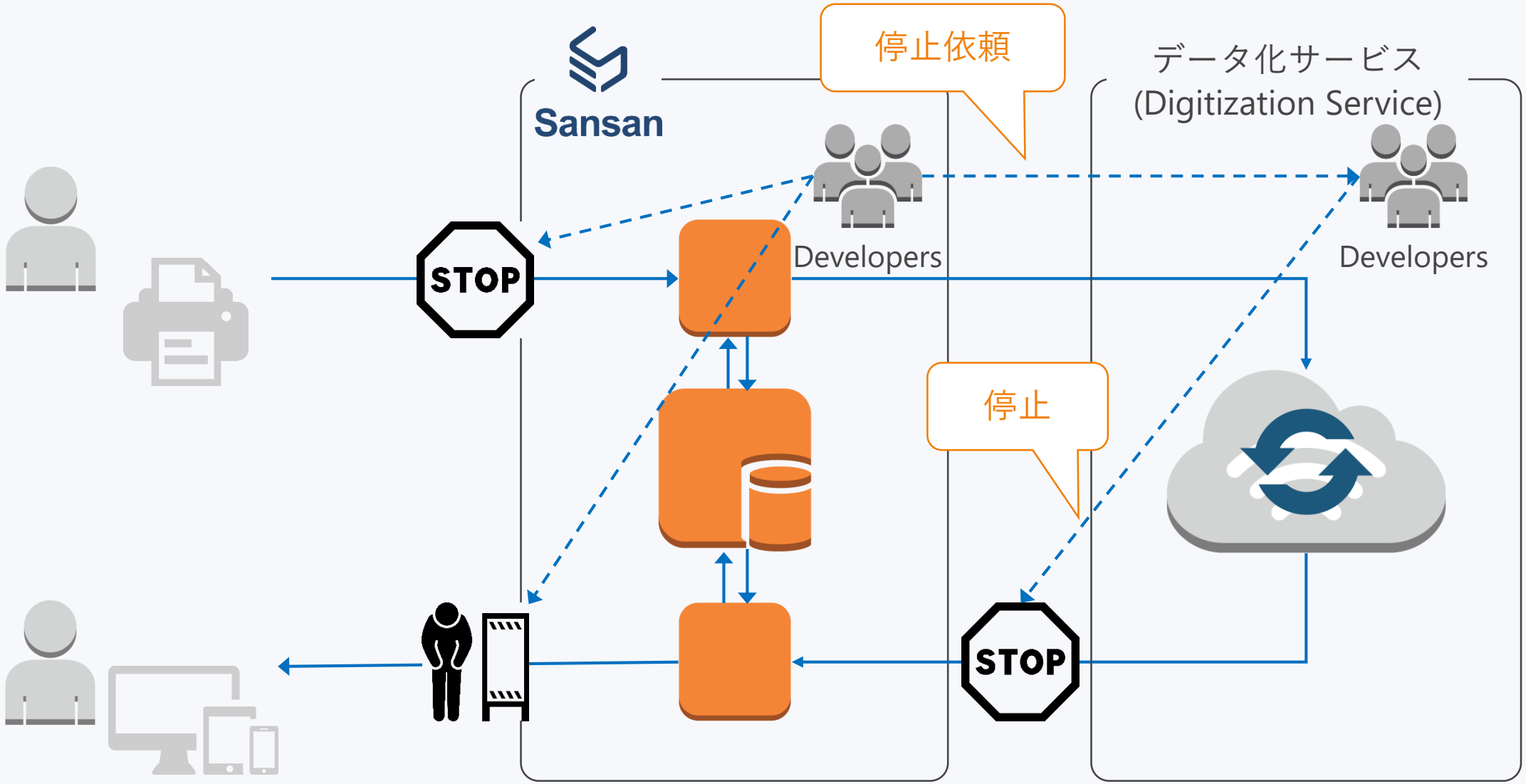
ワーカースレッドを起動

※イメージです

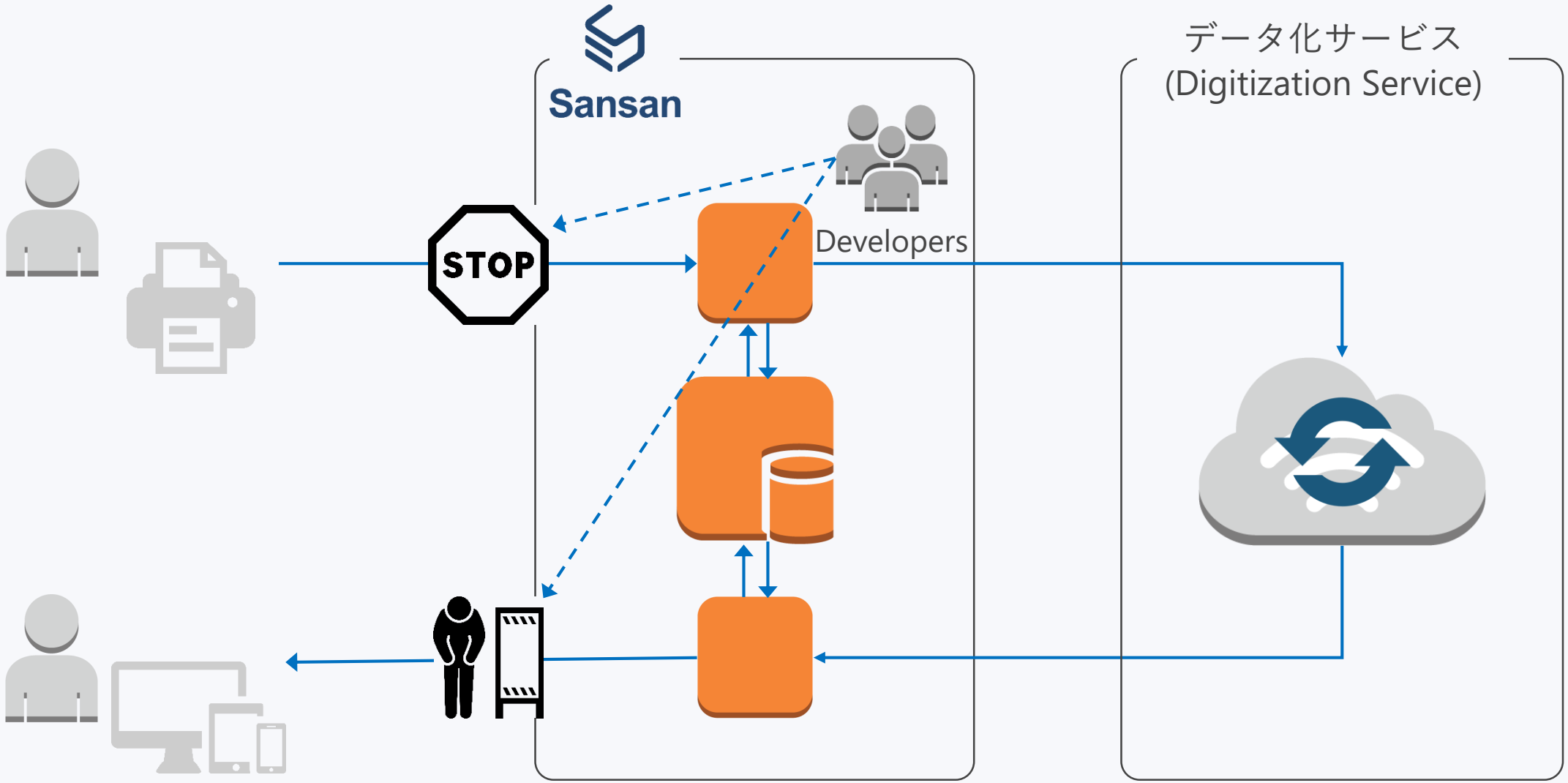
抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

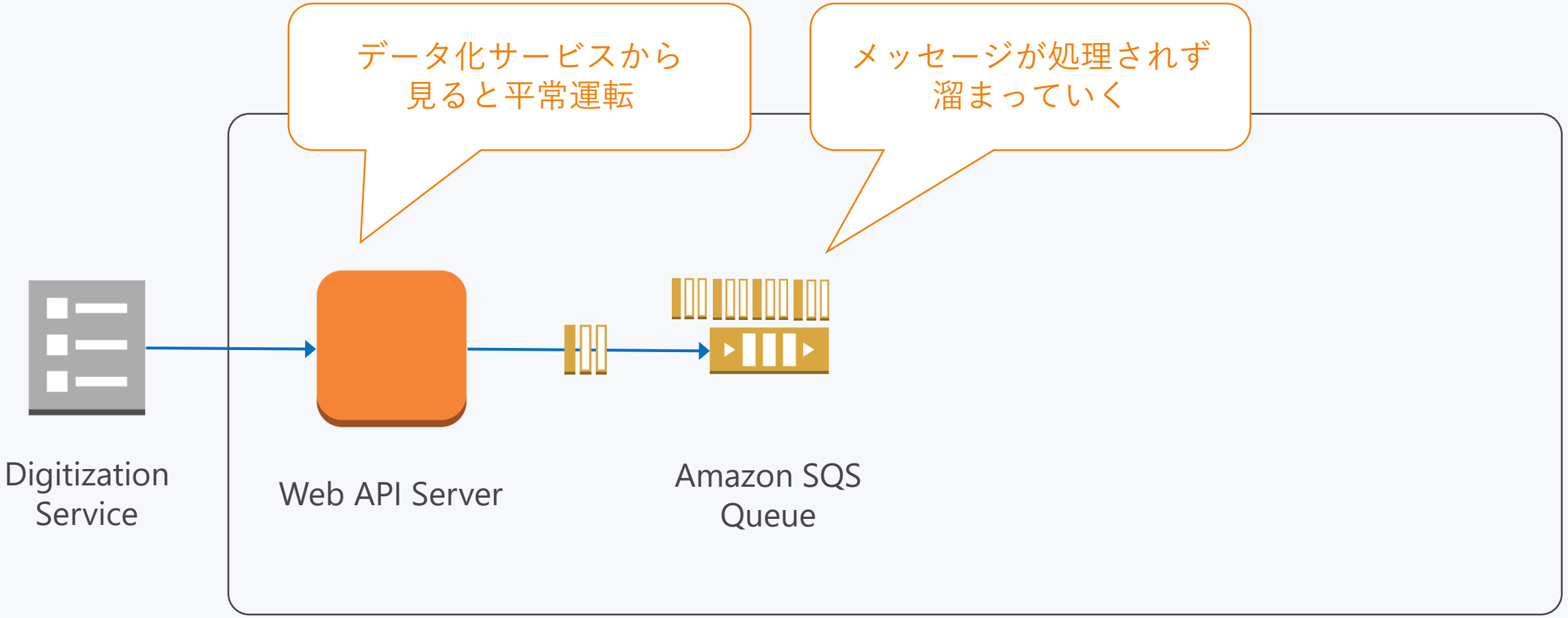
メンテナンス前 (従来)


Sansan
停止依頼
データ化サービス
(Digitization Service)
Developers
Developers
STOP
停止
STOP

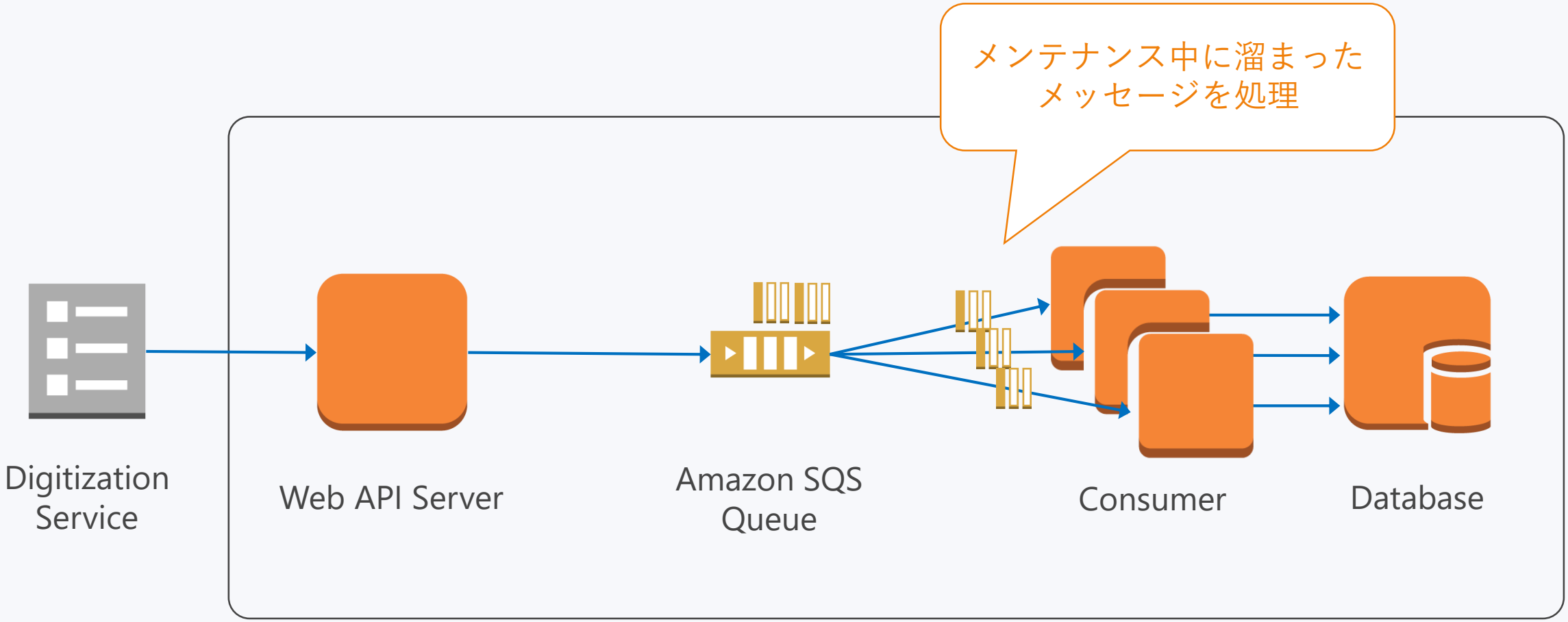
メンテナンス前 (現在)



メンテナンス中


データ化サービスから
見ると平常運転
メッセージが処理されず
溜まっていく
Digitization
Service
Web API Server
Amazon SQS
Queue

メンテナンス完了時


メンテナンス中に溜まった
メッセージを処理
Digitization
Service
Web API Server
Amazon SQS
Queue
Consumer
Database

抱えていた課題

- 巨大なトランザクション
- 終わらないバッチ処理
- 急激に変化するデータベース負荷
- 低いメンテナンス自由度
- リカバリ不可能な処理

リカバリ不可能な処理

タイムアウト、外部サービス障害等による処理失敗時

メッセージングなし

- リトライは自前実装
- 落ちた箇所によってはリカバリ不可能

メッセージングあり

- リトライは標準装備
- リトライしても失敗し続けた処理は**Dead Letter Queue**へ
- **処理内容がテキストで表現**されているので、必ずリカバリ可能

メッセージングの注意点

銀の弾丸はない

注意点

- 冪等性を保証する必要がある
- 順序は保証されない (Amazon SQS Standard Queue)
- 結果整合性モデルになることが多い

注意点

- 冪等性を保証する必要がある
- 順序は保証されない (Amazon SQS Standard Queue)
- 結果整合性モデルになることが多い

冪等性 (べきとうせい) とは

ある操作を**何回行っても結果が同じ**こと

REST API設計の議論で

HTTP POST (冪等でない) とHTTP PUT (冪等) の意味の違い

インフラ界隈で

ChefやAnsibleで実現されている

メッセージングと冪等性

- Amazon SQSは基本、**At-Least-Once Delivery** (Standard Queue)

  Exactly-Once ProcessingモデルのFIFO Queueもあるが、遅い & Tokyoに来ていない

- 例外発生等によりメッセージの処理に失敗した場合、該当処理はリトライされる

冪等性を保証する必要がある

強い一貫性モデル (ACID) を持つRDB等で保証するのが基本

注意点

- 冪等性を保証する必要がある
- 順序は保証されない (Amazon SQS Standard Queue)
- 結果整合性モデルになることが多い

順序は保証されない

- Amazon SQSは基本、順序保証なし (Standard Queue)
  FIFOを保証するFIFO Queueもあるが、遅い & Tokyoに来ていない

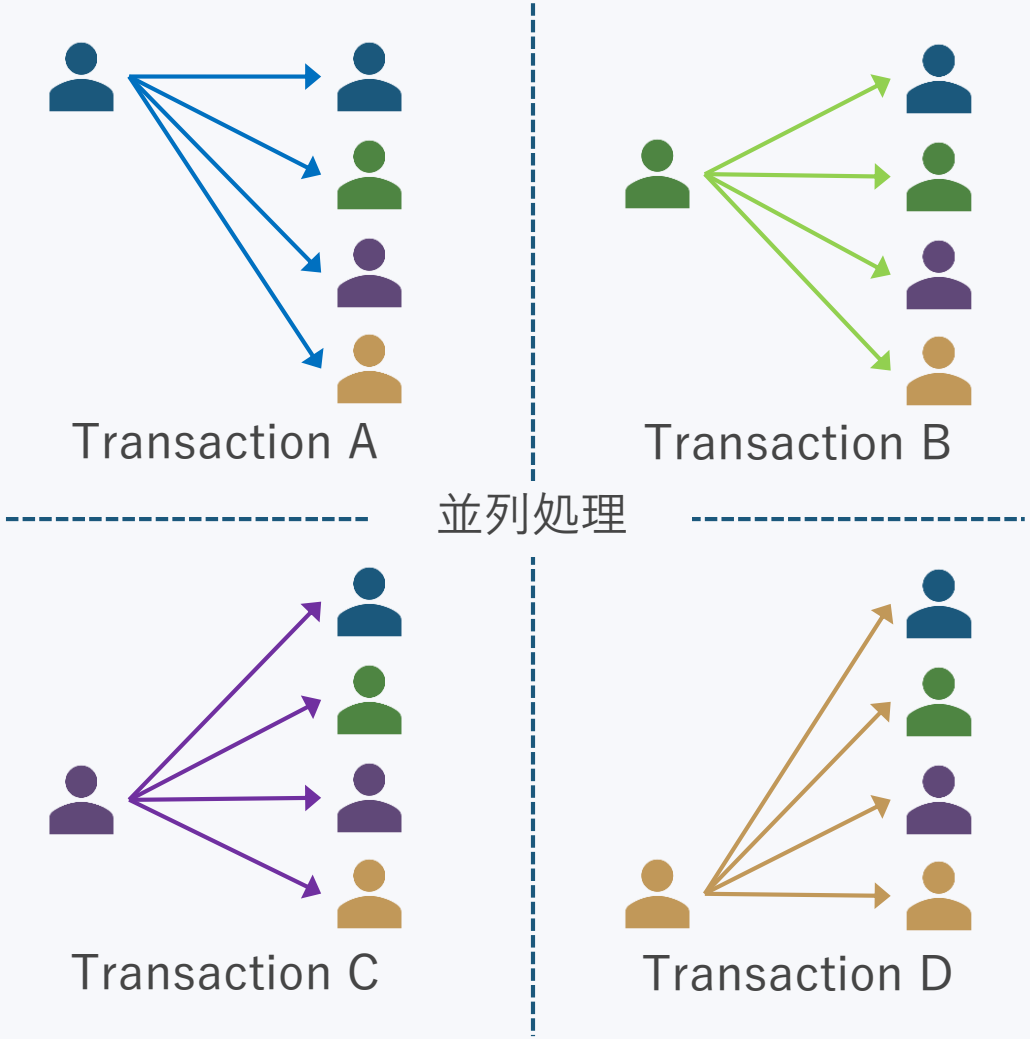
順序保証は自前で行う必要がある

メッセージの処理の最後に次のメッセージを送信する等

注意点

- 冪等性を保証する必要がある
- 順序は保証されない (Amazon SQS Standard Queue)
- 結果整合性モデルになることが多い

結果整合性モデルになる例



Transaction AとBしか終わっていない時点では、ビジネストランザクションとして整合性が取れていない

許容できない場合、分割後の全てのメッセージ処理が完了するまで処理結果を見せない等の制御が必要

まとめ

伝えたいことを改めて

まとめ

- メッセージングを使うことで以下を向上できる
  - スケーラビリティ
  - 堅牢性
  - 運用性
- 反面、設計難易度・複雑度は上がる傾向にある

特性を知り、適切な場面・方法で使用すべし