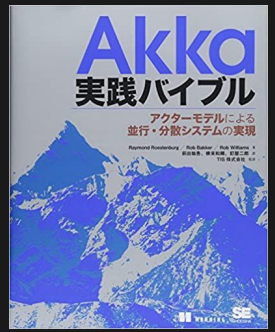
AWS Dev Day Online Japan 2021

I-4: AWSでスケーラビリティとレジリエンスを実現するアーキテクチャを考える

2021-09-30, Chatwork株式会社 テックリード 加藤潤一(@j5ik2o)

自己紹介

- プログラミングは10歳から
- レビュー
  - エリック・エヴァンスのドメイン駆動設計
  - Akka実践バイブル(Akka in Action)
  - ドメイン駆動設計入門
- 仕事でScala, 趣味でRust



このセッションの目的

クラウドを採用しても、スケーラビリティやレジリエンス(回復力)は自動的に得られるものではありません。スケーラビリティやレジリエンスはサービスの応答性の問題であり顧客の関心事です。あらゆるシステムでこれらの非機能要件は必ずしも高レベルではありません。

しかしシステムから応答性が失われた際は、ユーザから見限られる可能性が高くなります。最悪は代替サービスに乗り換えられてしまいます。そのような事態を招かないために私たちエンジニアにできることはないかアーキテクチャの側面から考えます。

このセッションの流れ

- Q1.なぜシステムに応答性が必要なのか
- Q2.なぜメッセージ駆動が必要なのか
- Q3.なぜリアクティブ原則の考え方が必要なのか
- Q4.どのようにCQRS/Event Sourcingを設計・実装すべきか

Q1

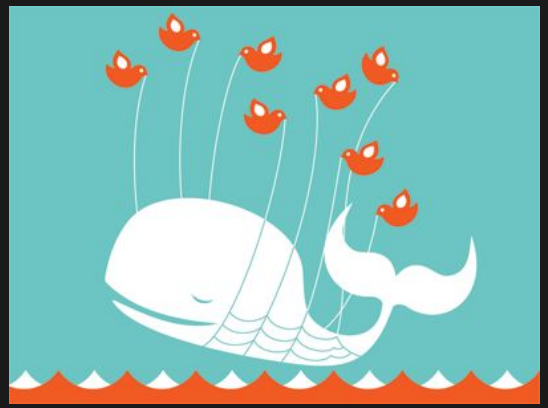
Q1.なぜシステムに応答性が必要なのか

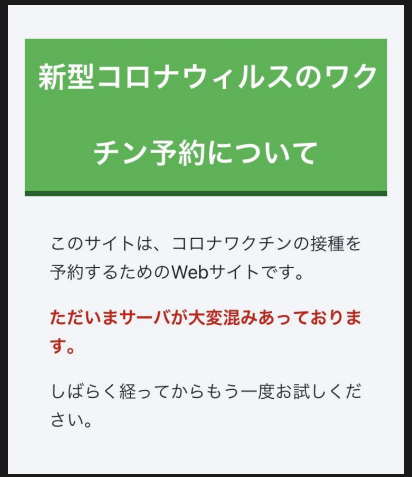
問い: サービス(システム)はいつでも使える？

Q1

社会に与える影響を考えると、システムに障害はつきものとなかなか言い切れない。

➡緊急度・重要度が高いが技術的な難易度が高いというギャップがある





Q1

どういう考え方が求められるのか

「止まらないシステム」ではなく「回復力があるシステム」が求められている

Q1

- 東京証券取引所, 株式売買システムで全銘柄で売買不能になった(2020/1)
  - NASのフェイルオーバーができなかったことが原因
- 恒久対策は回復力を向上させること。異常を起こした部分を切り離し障害から回復できるアーキテクチャに変更していくために、MSAに移行していく、とのこと

止まらないシステム(全く障害を起こさない完全なシステム)を目指すのではなく**【障害から回復する能力を設計すること】**に価値がある

Q1

システムが止まらなければよいのか？
多少 遅くても問題ない？

8秒ルールとは

Q1

8秒ルール - Wikipedia によると、「ウェブサイトを構築する際のガイドライン・経験則の1つ。利用者がそのサイトを訪れてから、ページ全体の内容が表示されるまでに8秒以上を要すると、利用者は待ちきれずに他のサイトに行ってしまい、再び戻ってくることが非常に少ないとされる。」

別の事例では2秒遅いだけで直帰率50%増加という数字もある。今のユーザは3秒も待てないのではないか。つまり応答性が重要なファクタになっている

Q1

「システムの停止」「システムのレイテンシの悪化」を
言い換えると、**応答性の消失や低下**

課題は**システムの応答性**

Q1

システムに応答性がないとどうなるか

ユーザがそのサービスを見限り
代替に乗り換えることに…

ユーザや事業の立場からみると
「応答性の確保」は ほぼMUST要件

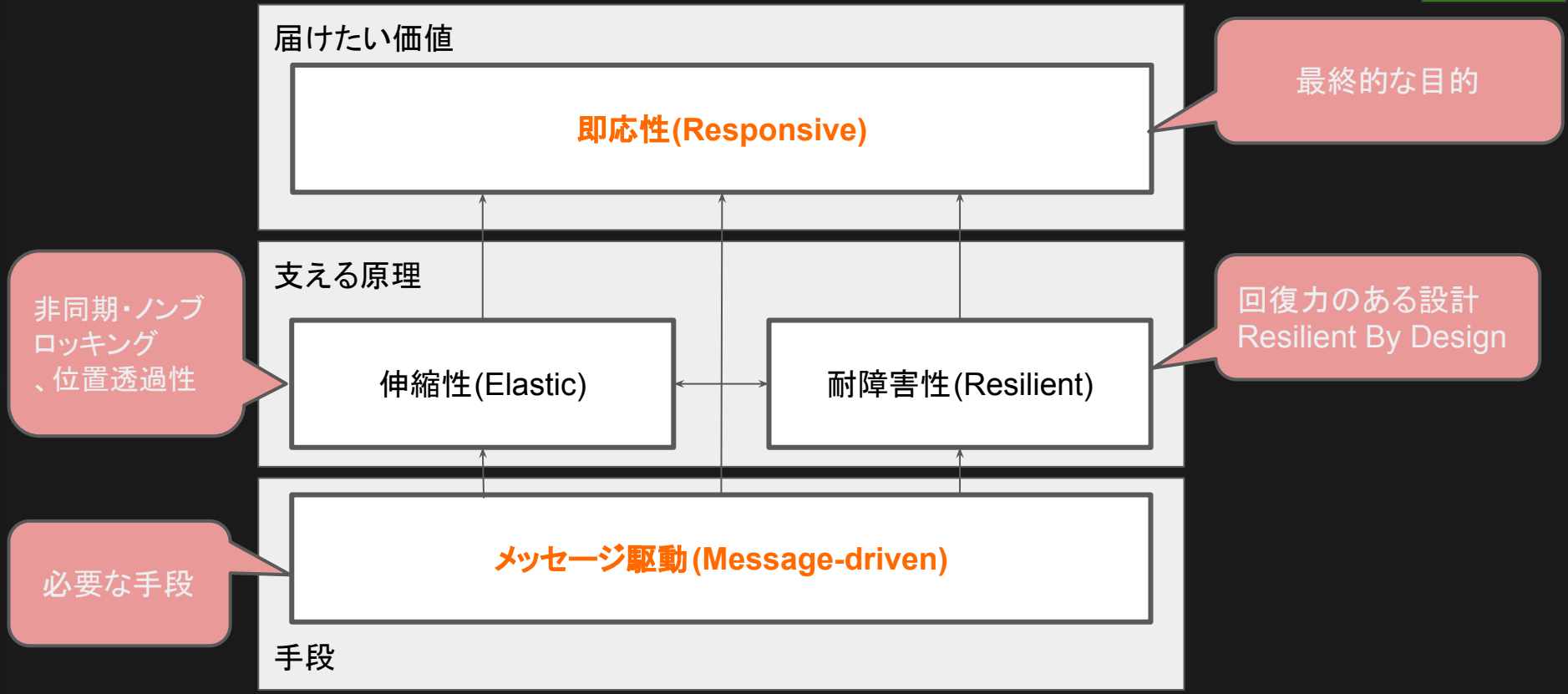
Q1

障害に強くて
ワークロードが急増しても
応答性があるシステムを
どのように実現がすればいいのか？

<<事業的にも技術的にも価値があること>>

リアクティブ宣言(2014)

Q1



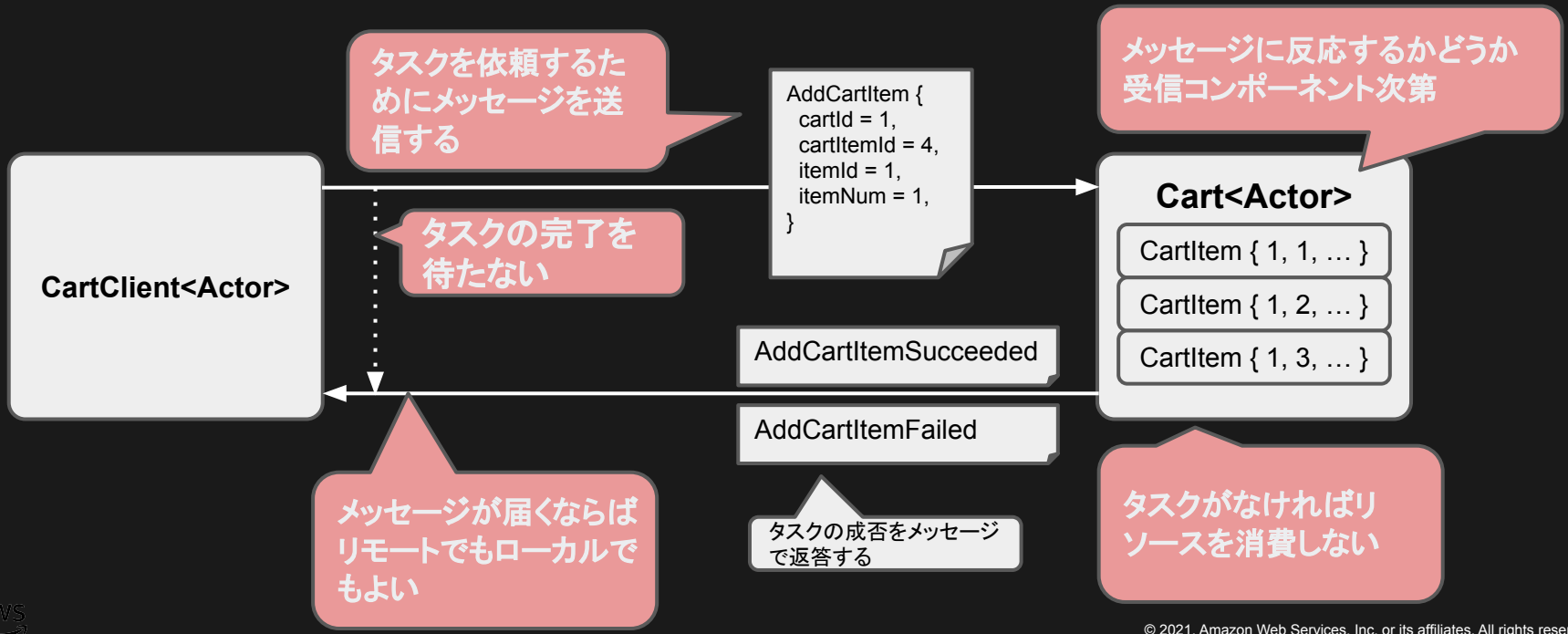
Q2

Q2.なぜ**メッセージ駆動**が必要なのか

メッセージ駆動とは

Q2

リアクティブシステムは【非同期・ノンブロッキング】なメッセージ・パッシングによってコンポート間の境界を確立する



なぜメッセージ駆動か

Q2

- マルチスレッドプログラミングの複雑さを隠蔽し、マルチコアをいかしたプログラミングが容易になる
- ノンブロッキングとイベントループによって、C10K問題の解決
- 信頼性の高いソフトウェアを実現できる。
  - Erlangのアクターモデル。電話交換機で99.9 999 999%の稼働率。論理的には20年間で1秒未満の停止時間

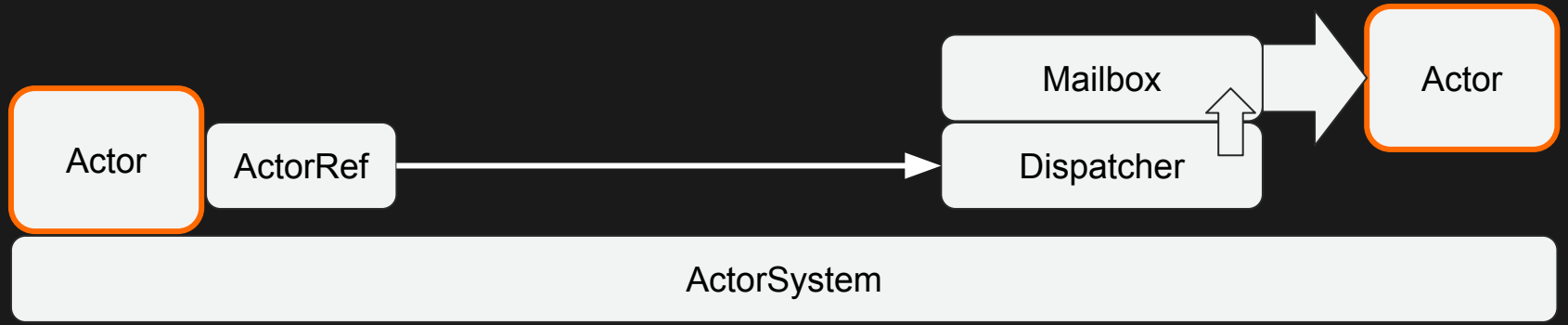
Q2

アクターモデルによるメッセージパッシング

メッセージパッシングはスループットを最適化できる

Q2

送信側アクターがアクター参照を使ってメッセージを送信する。メールボックスに溜まったメッセージを受信側アクターが処理する。**非同期・ノンブロッキング**が前提



メッセージの送信は返信を待たない。送信者は返信が来るまで別のタスクを処理できる

Q2

future/promise
async/await

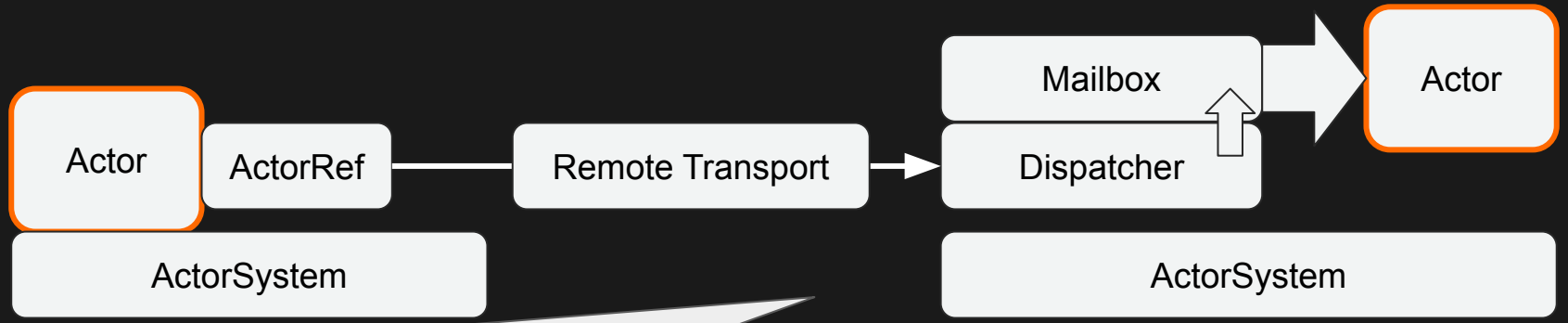
pub/sub
channel

actor
reactive streams

メッセージパッシングはローカル/リモートの区別がない

Q2

- アクター参照はローカルであってもリモートのように。メッセージ送信側は、すべての宛先がリモートに見える。
- リモートから呼出しであってもローカル呼出しのように。メッセージ受信側は、送信元がすべてのローカルに見える

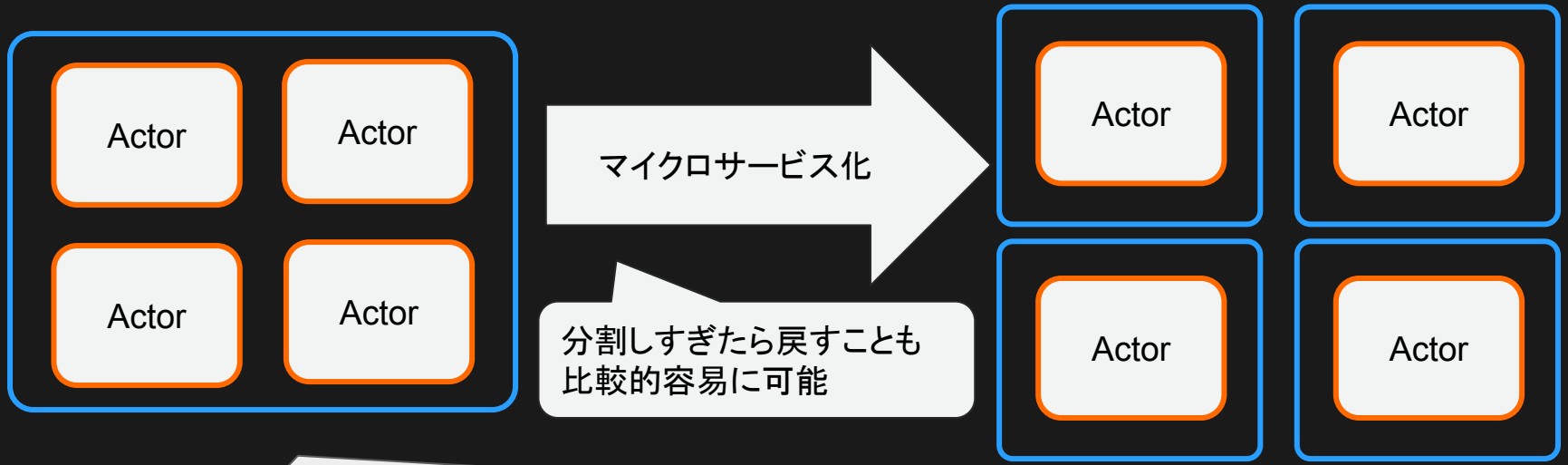


アクターはローカルでもリモートでも区別がない(位置透過性)。マルチスレッドとRPCをメッセージ駆動というプログラミングモデルで統一する

メッセージ駆動によるモジュラーモノリスからのMSA化

Q2

- アクターモデルからみた場合、右も左も違いがない
- 二つの位置関係が混在しても問題はない



同一トランザクションに様々な関心を巻き込む設計でモジュラーモノリスを構築すると、後の分割のハードルを上げることになりかねない。

Q2

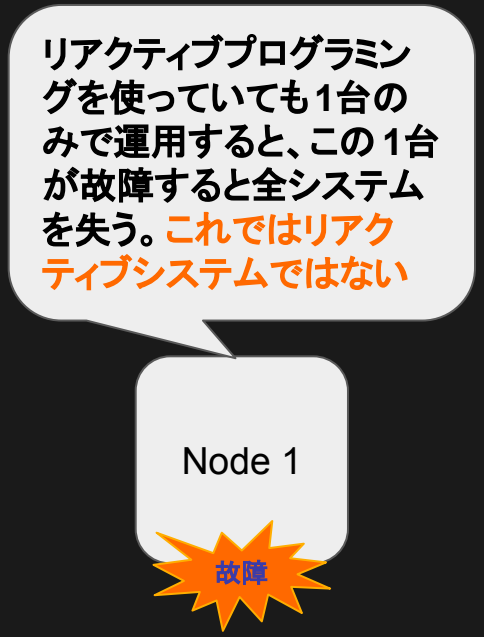
第1世代
Erlang/Elixir
Akka(Scala,Java)
第2世代
Dapr(Go)
Orleans(.NET)
proto.actor(Go,.NET,Kotlin)

Swift 5.5から言語組込でactor構文がサポートされた。distributed actor構文も提案されている

リアクティブシステム ≠ リアクティブプログラミング

Q2

- リアクティブシステムはアーキテクチャレベルでリアクティブ原則を適用する
- リアクティブシステムを実現する手段としてFuture/Promise, アクターモデルなどのリアクティブプログラミングが利用されます。だからといって、**自動的にリアクティブシステムになりません**



Q3

Q3.なぜリアクティブ原則の考え方が必要なのか

リアクティブ原則(2020)

Q3

- 応答性を維持する/Stay Responsive
- 不確実性を受入る/Accept Uncertainty
- 失敗を受け入れる/Embrace Failure
- 自律性を表明する/Assert Autonomy
- 一貫性を調整する/Tailor Consistency
- 時間を分離する/Decouple Time
- 空間を分離する/Decouple Space
- ダイナミクスを処理する/Handle Dynamics

今回はこの2点を解説

失敗を受け入れる/Embrace Failure

Q3

- **物事がうまくいかないことを期待し、回復力のために構築する**
- キーとなる考え方はBulkheading。Bulkheadingは船舶由来の用語。大型貨物船の船倉は隔壁によって多くの区画に分割される。船底が何らかの原因で破損した場合でも、影響を受けた区画だけが浸水し、他の区画は適切に密閉された状態を維持できるため浮力を維持できる
- 以下の図はReactive Design Patternsで紹介されている

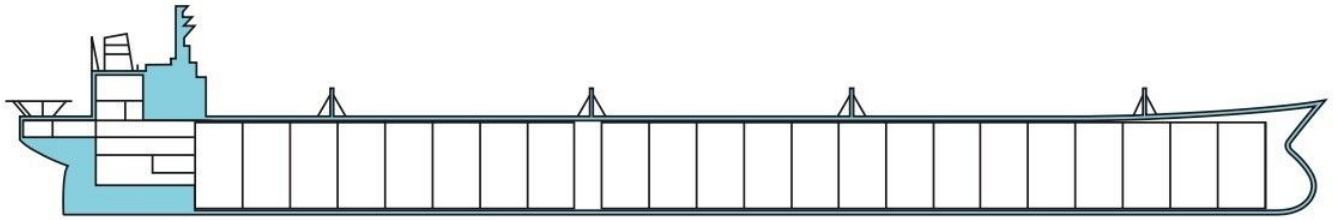
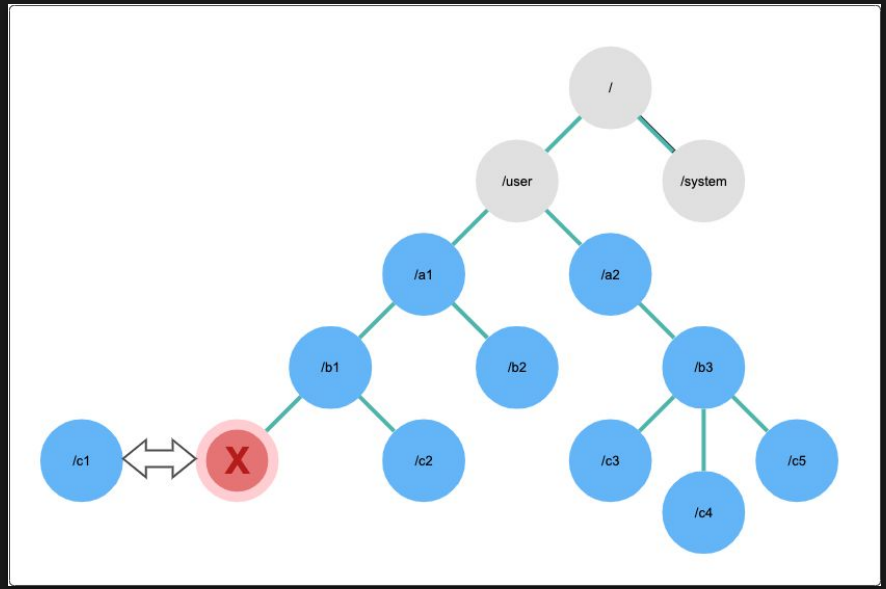


Figure 2.11 The term *bulkheading* comes from ship building and means the vessel is segmented into fully isolated compartments.

アクターモデルでどのようにBulkheadingするか

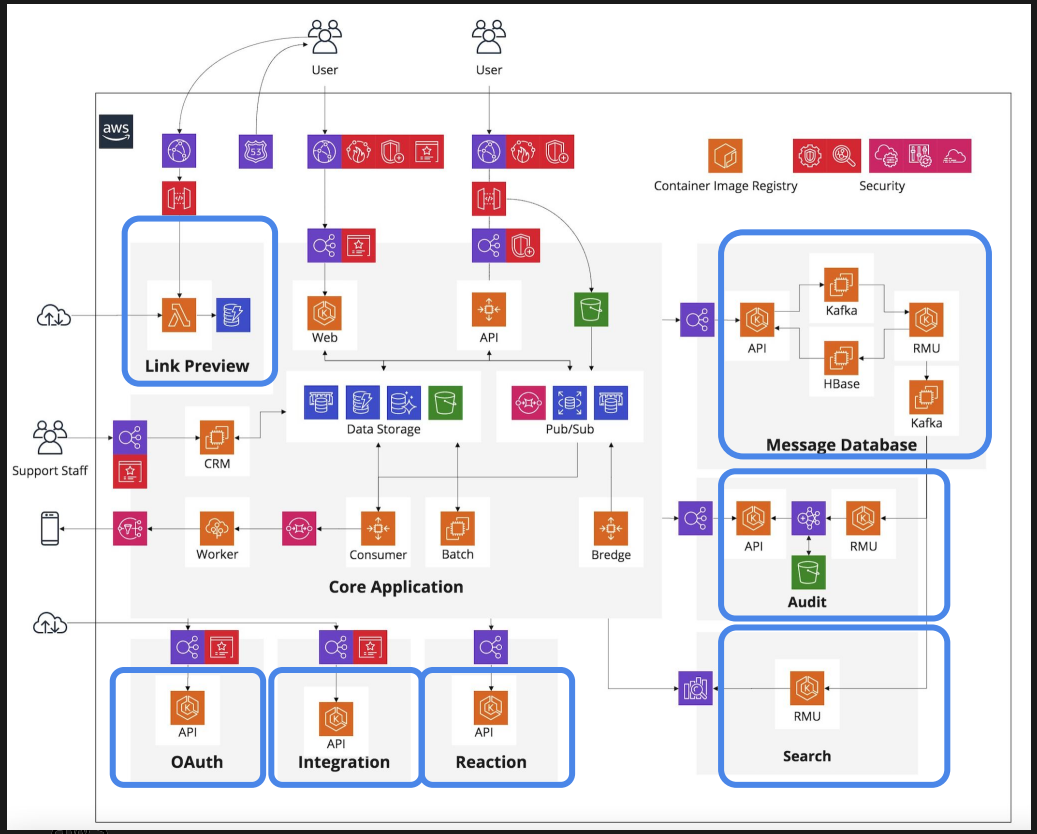
Q3

- スーパーバイザであるコンポーネントは簡単に故障するような仕事はせずに、失敗しやすい仕事はヒエラルキー下層の専門のコンポーネントに任せる
- このような構造を採用することで障害が発生しても、全体に障害が波及することを抑制する
- 障害発生時はスーパーバイザに判断を委任し、その指示に従ってコンポーネントを再起動して復旧する。このような階層的な再起動を用いる障害処理によって、障害モデルを大幅に簡素化でき、予期しない障害に直面しても生き残る可能性が高める



FYI: Chatworkでのアクターモデルの採用

Q3

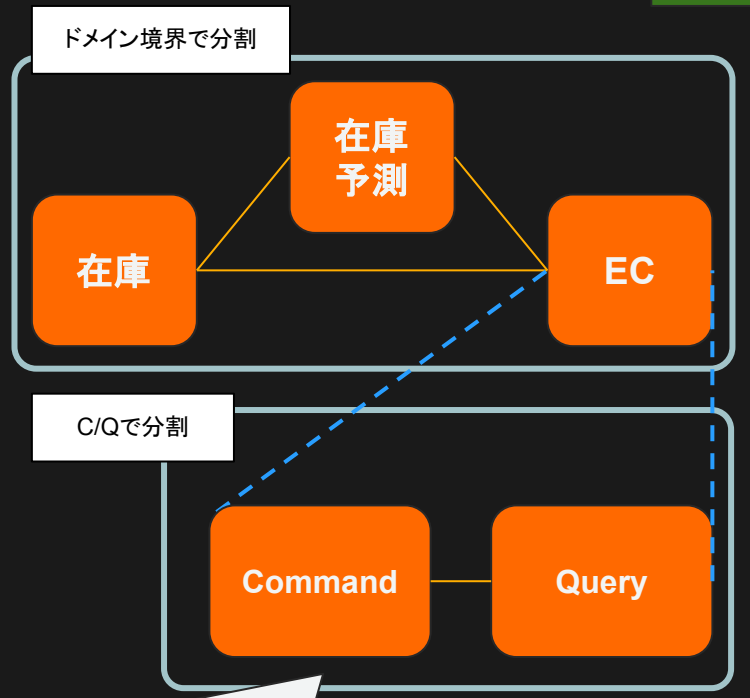


- 2016年末 メッセージング基盤にて、Akka,HBase,Kafkaを使って、CQRS+ESシステムを構築・運用開始
- 他のマイクロサービスでもAkkaを積極的に採用
- Core Applicationを刷新する計画を進行中

自律性を表明する/Assert Autonomy

Q3

- **独立して行動し、協調的に相互作用するコンポーネントを設計する。**自律性とは、各マイクロサービスが境界を維持し独立して運用できること
- **自律性を保つにはアプリケーションを分離する必要がある。**分離には主に以下の観点がある
  - DDDの**境界づけられたコンテキスト**単位で分離する
  - **CQRS/Event Sourcing**でのコマンドとクエリに分離する



Commandに障害が起きてもQueryできるようにするにはお互いに分離する必要がある

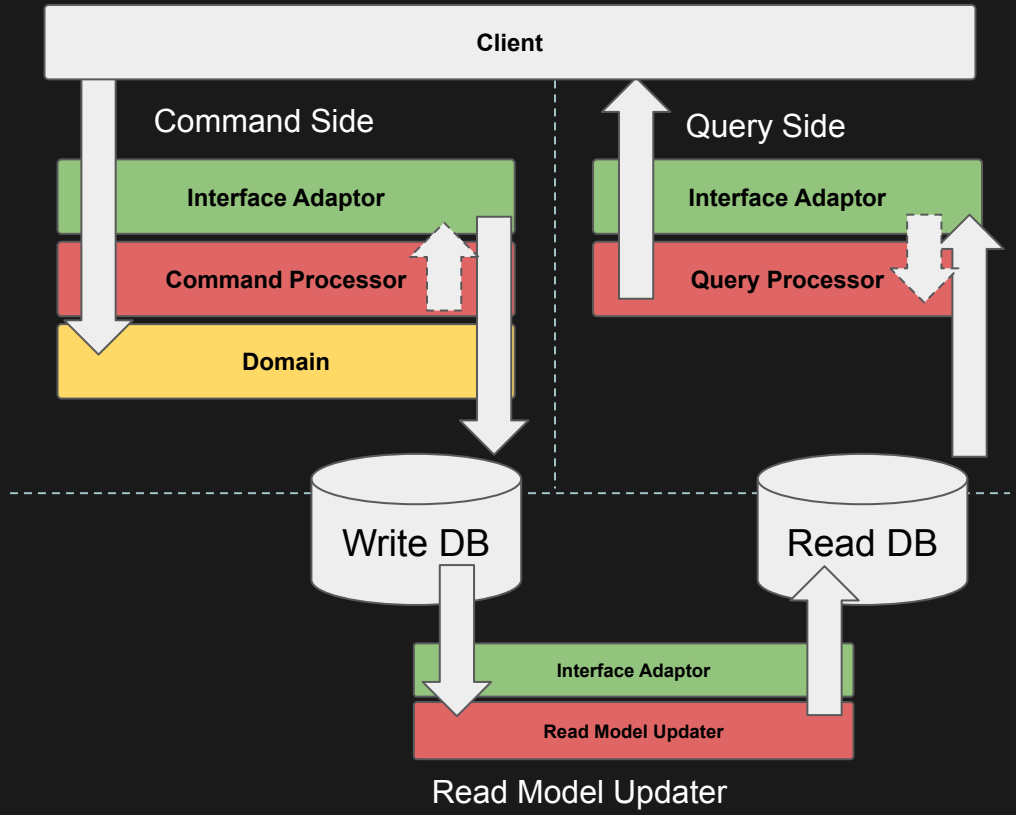
Q3

CQRS/Event Sourcing

CQRSとは

Q3

- **Command and Query Responsibility Segregation=コマンド・クエリ責務分離。分離というより隔離という解釈が正しい**
- **コマンド(書き込み)とクエリ(読み込み)をスタックごとにそれぞれに隔離することを意味する。単にドメインモデルをコマンド用・クエリ用に分割することではない**
- CQRSはDDDを前提としています(ドメインから本質的ではないクエリ責務を排除するための設計パターンです)。詳しくは CQRS Documents by Greg Young を参照のこと



CQRSの利点・欠点

Q3

- Pros
  - コマンドとクエリを別々に**デプロイ**できる、**耐障害性が高くなる**
  - コマンドとクエリを別々に**最適化**できるため、**スケーラビリティが高くなる**
- Cons
  - 分散システムとして、構成要素が多くなる
    - ネットワーク分断時に **一貫性** or **可用性** どちらを優先するか選択を迫られる

なぜCQRSなのか？

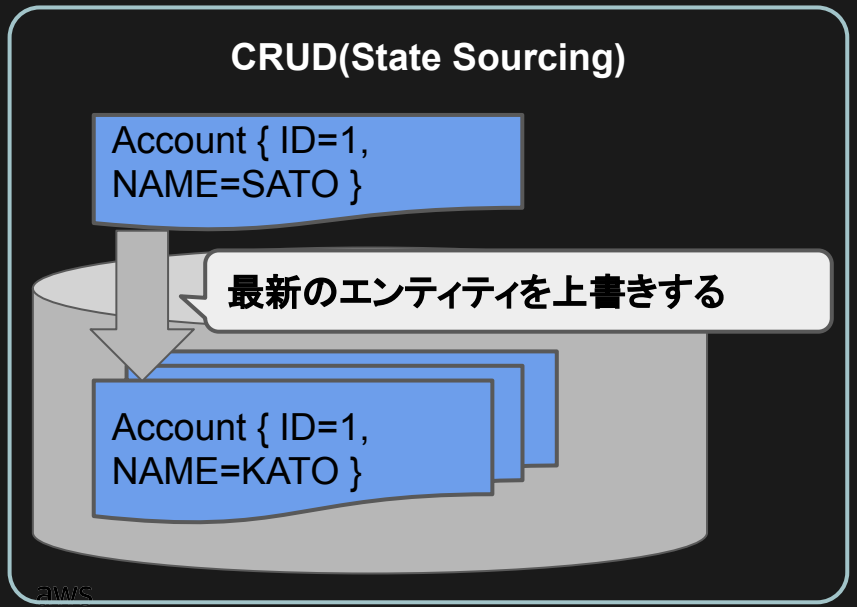
Q3

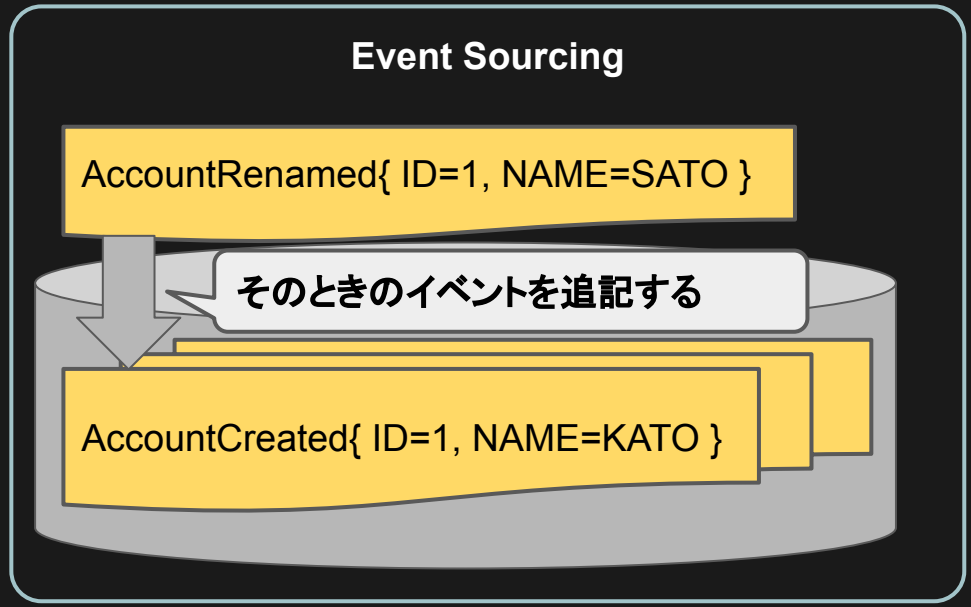
- そもそもC/Qで要件が異なるから
  - コマンド側は**書き込みの一貫性重視**、クエリ側は**読み込みの可用性重視**
  - コマンド側は**正規化されたデータ**、クエリ側は**非正規化されたデータ**を扱う
  - コマンド側より**クエリ側のほうがスケーラビリティ**が必要になる

Event Sourcingとは

Q3

- 唯一信頼できる情報源(Single Source Of Truth)は、状態(ステート)ではなく(ドメイン)イベントという考え方
- CRUDは、従来からの最新状態を常に上書きする
- コマンドとクエリを統合するために使う

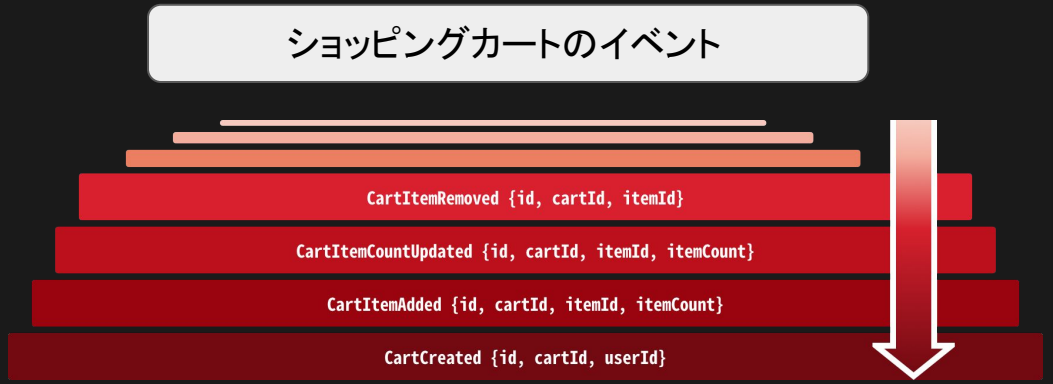




FYI: ドメインイベントとは

Q3

- イベントは過去に発生した出来事(事態)
- ドメイン上のイベント＝ドメインイベント
- 動詞の過去形で表現される
  - CustomerRelocated
- イベントからコマンド(命令)が想起可能
  - RelocateCustomer



なぜドメインイベントを使うのか

Q3

- CとQを連携させるため
  - C→Qに変更をイベントという形式で表現して通知する
- ドメイン分析で使える
  - イベント(コト)はリソース(モノ)やエージェント(ヒト)と関連するため、ドメイン分析で有益。
  - いくつかの分析・設計手法でもイベントが分析の中核として捉えられている
  - T字型ER
  - REA (Resource-Event-Agent)
  - Event Storming

Event Sourcing の利点・欠点

Q3

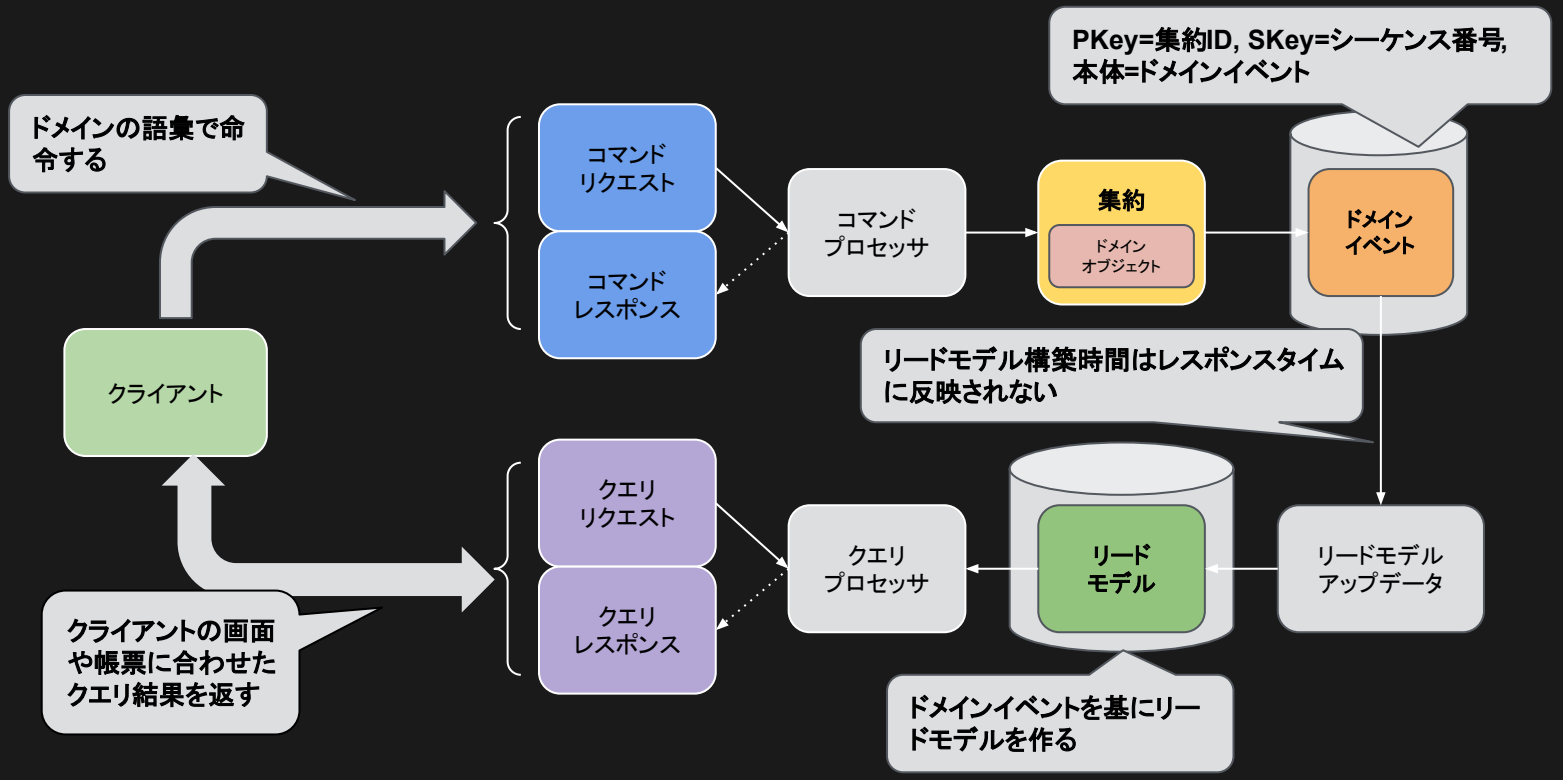
- Pros
  - **追記のみのイベント**はスケーラビリティが確保しやすい
  - イベントを基に**クライアント都合のリードモデルを構築**できる(再設計も自由)
  - イベントを使って**他のマイクロサービスの連携がしやすい**
  - 監査ログや行動履歴の分析に利用することができる
- Cons
  - 長大なイベントから状態をリプレイする際に時間がかかる
    - 最新状態を保存したスナップショットを使うとリプレイ時間を短縮できる
  - すべてのイベントをストレージに保存する必要がある
    - スナップショット保存時に、古いイベントを消すことも可能

Q4

Q4.どのようにCQRS/Event Sourcingを設計・実装すべきか

CQRS/ESのアプリケーションアーキテクチャ例(1)

Q4



コマンドプロセッサの実装イメージ(1)

Q4

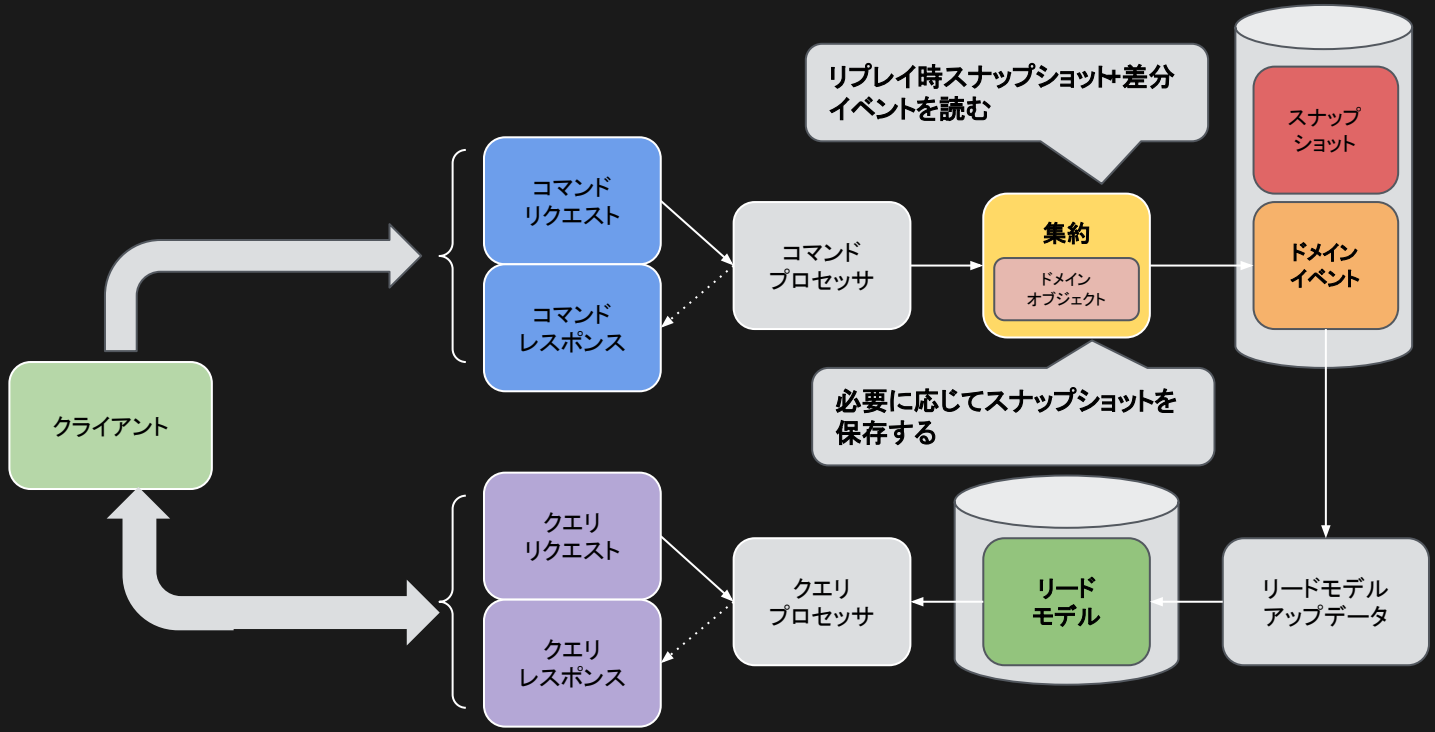
- DBにはイベントの追記しかしない前提

```
1  class AddCartItemUseCase(cartEventService: CartEventService) {
2
3    def execute(cartId: CartId, itemId: ItemId, num: ItemNum): Unit = {
4      // すべてのイベントを取得する
5      val allEvents = cartEventService.getAllEventsById(cartId)
6      // イベントから最新のカート状態を作り出す
7      val cart = Cart.fromEvents(allEvents)
8      // ドメインロジックの実行
9      val itemAdded = cart.addItem(itemId, num)
10     // イベントの永続化
11     cartEventService.store(itemAdded)
12   }
13
14 }
```

・データ競合を防ぐためのロックができないのでは..
・イベントが長大な場合、集約の再生に時間かかるのでは..

CQRS/ESのアプリケーションアーキテクチャ例(2)

Q4



コマンドプロセッサの実装イメージ(2)

Q4

```
class AddCartItemUseCase(cartPersistenceService: CartPersistenceService) {

  def execute(cartId: CartId, cartVersion: Version, itemId: ItemId, num: ItemNum): Unit = {
    // 最新のスナップショットを取得する
    val snapshot = cartPersistenceService.getLatestSnapshotById(cartId)
    // 最新のスナップショット以降のイベントだけを取得する
    val events = cartPersistenceService.getEventsById(cartId, snapshot.sequenceNumber)
    // イベントから最新のカート状態を作り出す
    val cart = Cart.fromSnapshotWithEvents(snapshot, events)
    // ドメインロジックの実行
    val (newCart, itemAdded) = cart.addItem(itemId, num)
    // 新しいスナップショットとイベントの永続化
    // クライアントから指定されたcartVersionで楽観的ロックが適用される
    cartPersistenceService.store(newCart, itemAdded, cartVersion)
  }

}
```

リクエスト毎に、リプレイやスナップショット保存のオーバヘッドがかかる.

CQRS/ESのために使えるツール

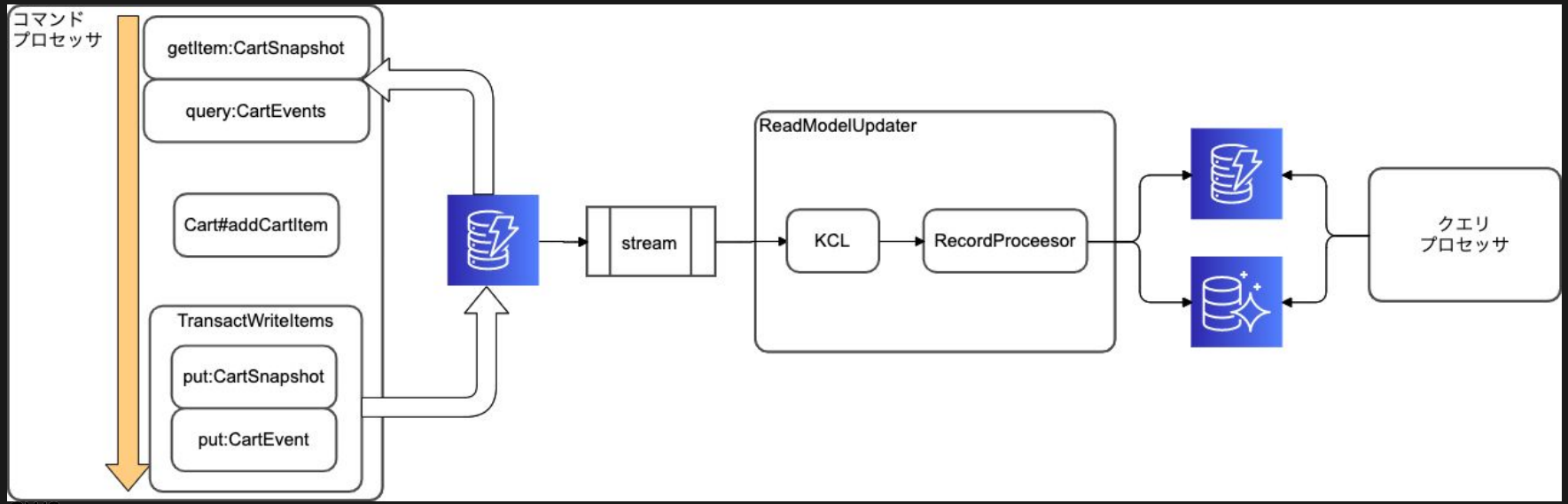
Q4

- これらのツールでアクターモデルを使う

AWSでどうやるかの例

Q4

- CartSnapshot, CartEventsを同じトランザクションで書き込む
- CartEventsのNewImageをStreamからコンシュームし後段につなげる
- 後段はKCLで使うなどが考えられる
- リードモデルは必要に応じてNoSQL, RDBMSを選択する



まとめ

- **分散システムの問題はそもそも難しい。真のクラウドネイティブを実現する上でも、リアクティブシステムから学ぶべきことが多いし、世界はこの方向に向かっている**と思われる
- 需要が見込める分野だが、まだまだ対応できるエンジニアは少ないので、投資が必要なフェーズ。海外でも関心は高まっている。日本でも知見や実績を積み上げていきたい。
- だからこそ、クラウドベンダさんのより強いサポートが必要になる。**技術として基礎理解をしつつも、難しいところはマネージドで実現でき、クラウドユーザがコアドメインに集中できるように！**

本日はご静聴ありがとうございました