Windows Azure 対応 SaaS アプリケーション設計の考慮点

マイクロソフト株式会社
デベロッパー＆プラットフォーム統括本部
アーキテクトエバンジェリスト
野村一行

アジェンダ

Windows Azureのロール、Tableの
活用方法

SaaS アプリケーション
アーキテクチャ設計の着眼点

マルチテナントにおける
データベース設計の選択肢

アジェンダ

Windows Azureのロール、Tableの
活用方法

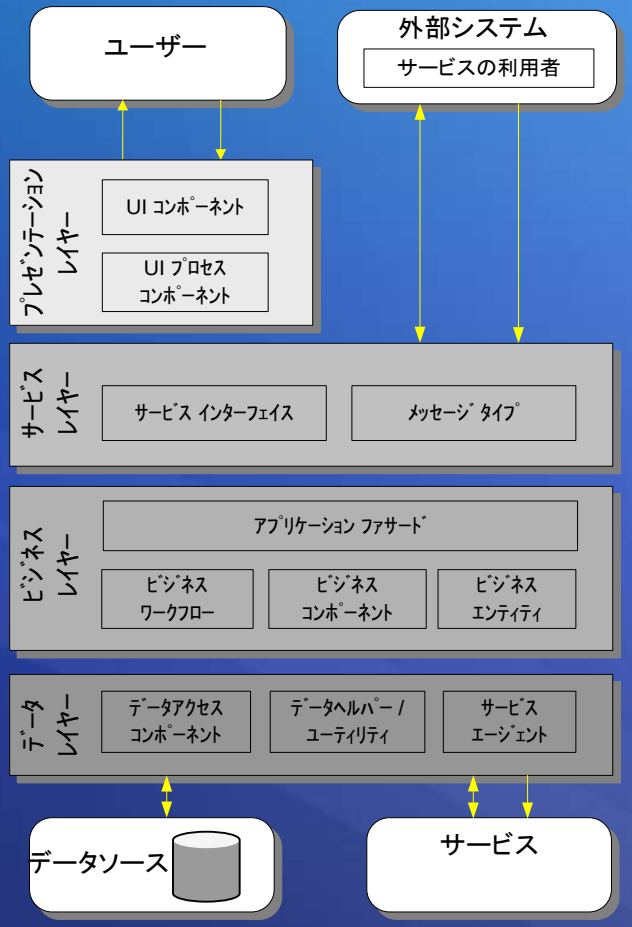
SaaS アプリケーション
アーキテクチャ設計の着眼点

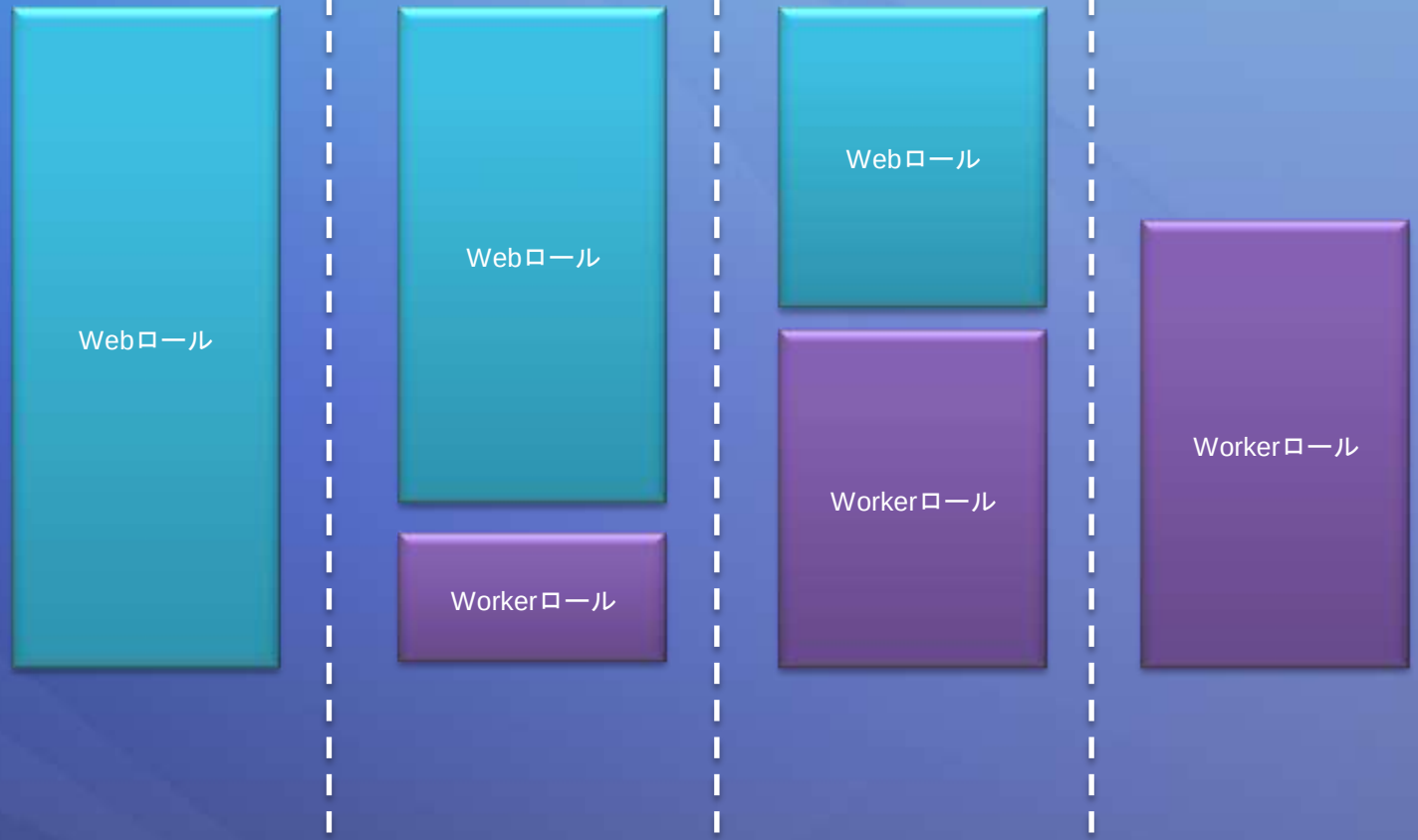
マルチテナントにおける
データベース設計の選択肢

WebロールとWorkerロールの役割の違い

- Webロール
  - ASP.NET 3.5 SP1 – 64ビット
  - Web プロジェクト
    - .ASPX
    - Web.config
    - 静的コンテンツ
  - Http(s)
- Workerロール
  - クラスライブラリ（マネージコード）
  - RoleEntryPoint を継承
  - Start() メソッド
    - 開始時にファブリックから呼ばれる
    - 本メソッドからリターンはしない
  - Stop() メソッド
    - ロールのシャットダウン時に呼ばれる
  - GetHealthStatus()
    - ヘルシーな状態かのハートビート

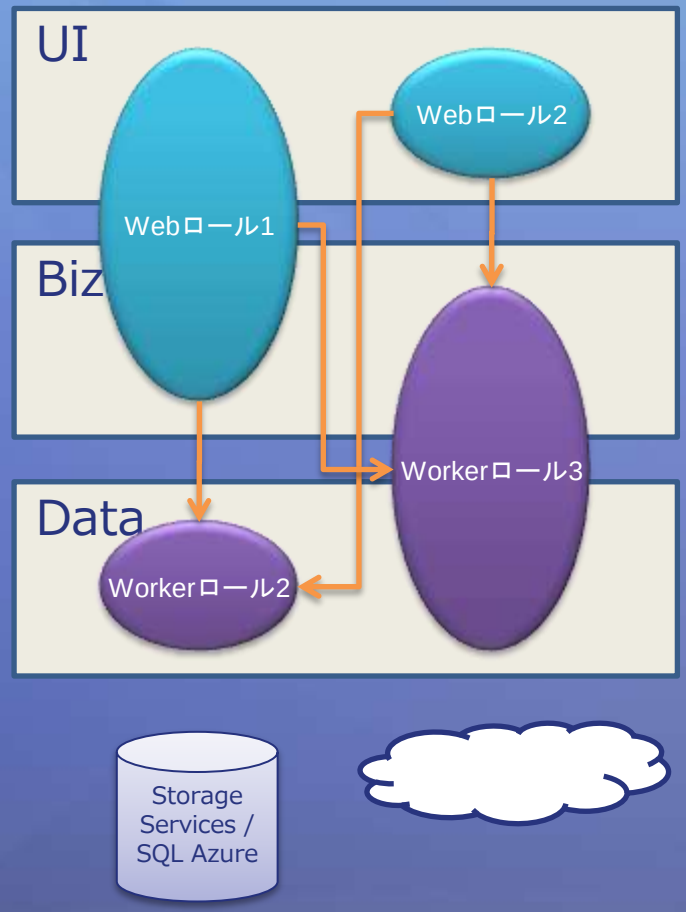
N階層アーキテクチャとAzureのロール





マルチロール

- マルチロール
  - 異なるタイプのWebロール、Workerロールが定義可能
  - 特長：異なる役割ごとにロールを定義するという設計アプローチが採用できる
  - 考慮点：ロール間の通信はキューを介する、ロールごとに別々のVMが作成される
  - したがって、アプリケーション管理のメリットを生かしつつ、ライフタイムが異なる場合（常時使用 or たまに使用）などにロール分割を検討する
    - Webロールはポート番号で区別され、サブドメインは作成されない

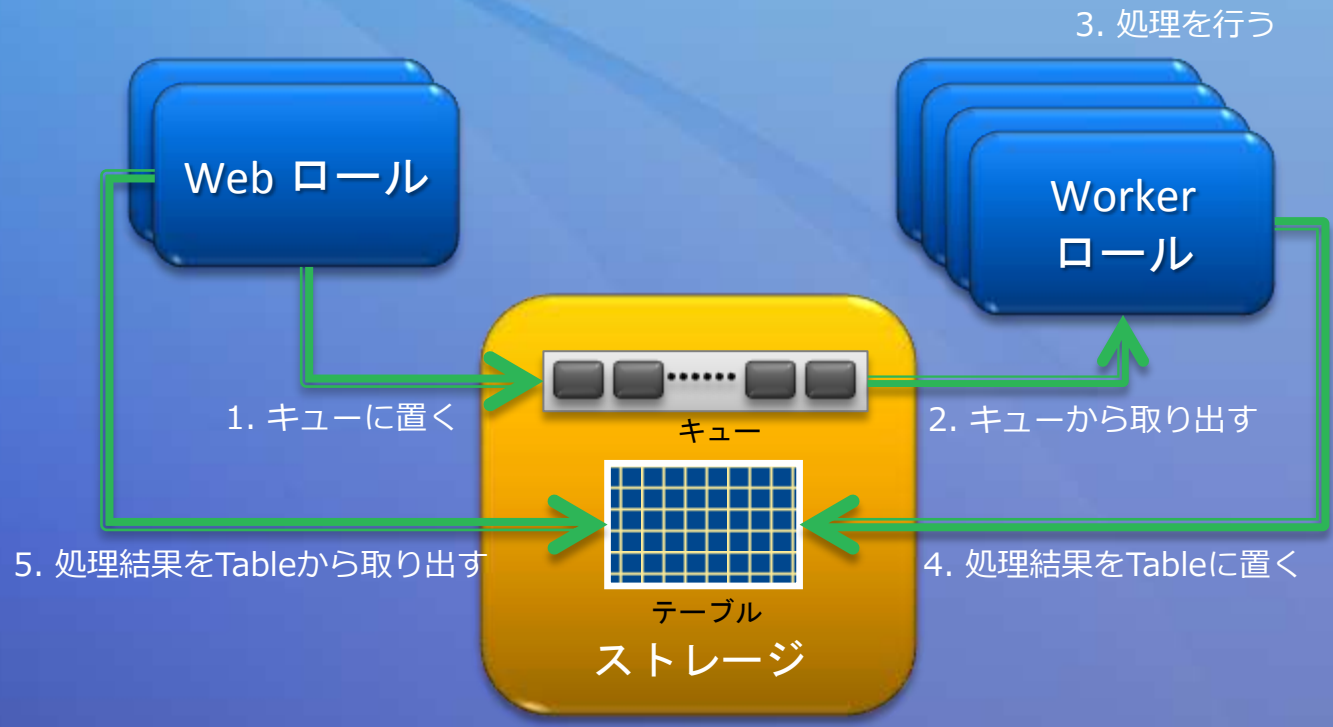


設計上の考慮点

- スケーラビリティと可用性を重視
- ステートレス
  - フロントエンドはステートレス、ステート保管はストレージに
- べき等（Idempotence）になるよう設計
- コンポーネント間をキューを使って疎結合化する
- アプリケーションの計測
- いったん動き始めたら、動き続けることを意識する
- パッチ、アップデートについて検討する

Workerロールからの応答の取得方法（例）

- Webロールから Workerロールへの一方向キュー
- Workerロールは処理の結果をTableに書き出し、Webロールは必要に応じて取り出す



Demo

- 開発用ファブリックから本番環境への移行

パーテションキーとパーテション

- すべてのTableはパーテションキーを持つ
  - 最初のプロパティ（カラム）
  - 同じパーテションキーを持つ、Table内のすべてのエンティティは同じパーテションに格納される
- つまり、データアクセスをスケーラブルにするにはパーテション化における基本方針を持つことが必要

Partition Key Document Name	Row Key Version	Property 3 Modification Time	…..	Property N Description
福利厚生Doc	V1.0	3/21/2007	…..	2007年度
福利厚生Doc	V1.0.6	9/28/2007		2008年度用 山田作成中
勤怠Doc	V1.0	3/28/2007		2007年度
勤怠Doc	V1.0.1	7/6/2007		2008年度用 千田作成中

同じパーテション

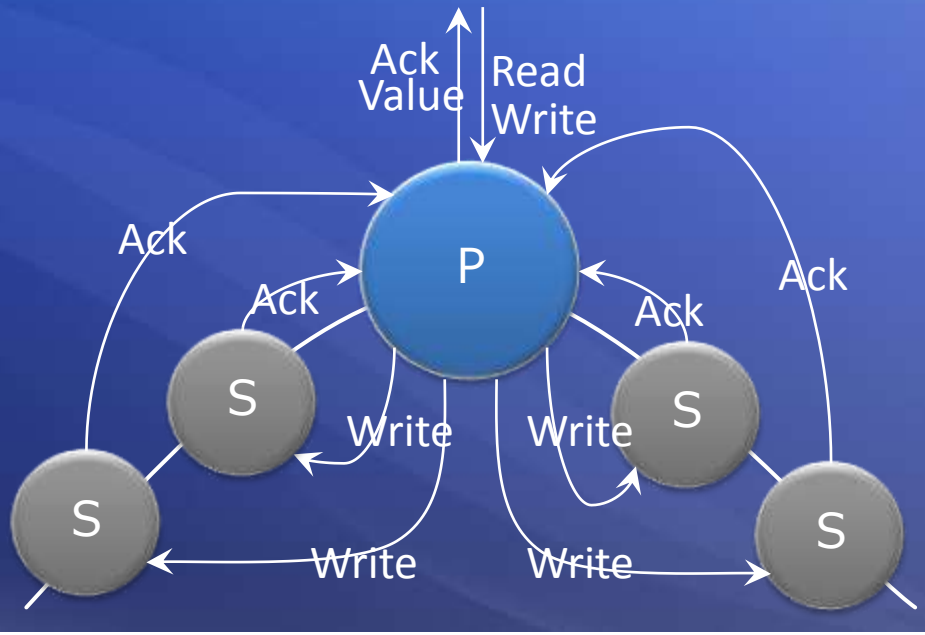
同じパーテション

Azureにみるクラウド技術：冗長ゆえの可用性

- 数万台もあればどれかが壊れて当然、という考え方
- 障害を前提としたフェイルオーバーの機構をAzure自体が持つ

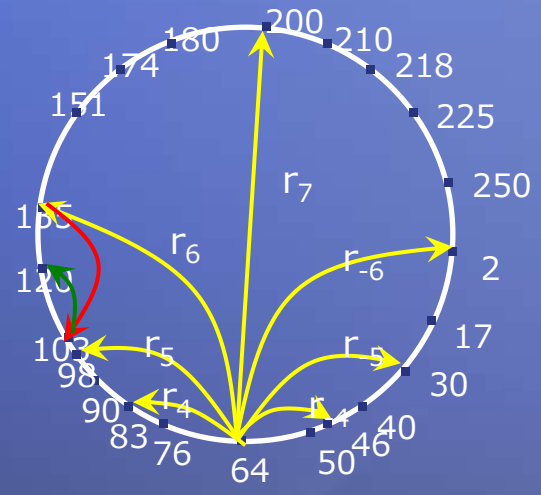
データ冗長化のイメージ

- データの読み出しはプライマリのみ
- セカンダリノードに非同期で書き込みを行う
- プライマリ障害時に昇格するセカンダリは多数決で決定



構造化オーバーレイ

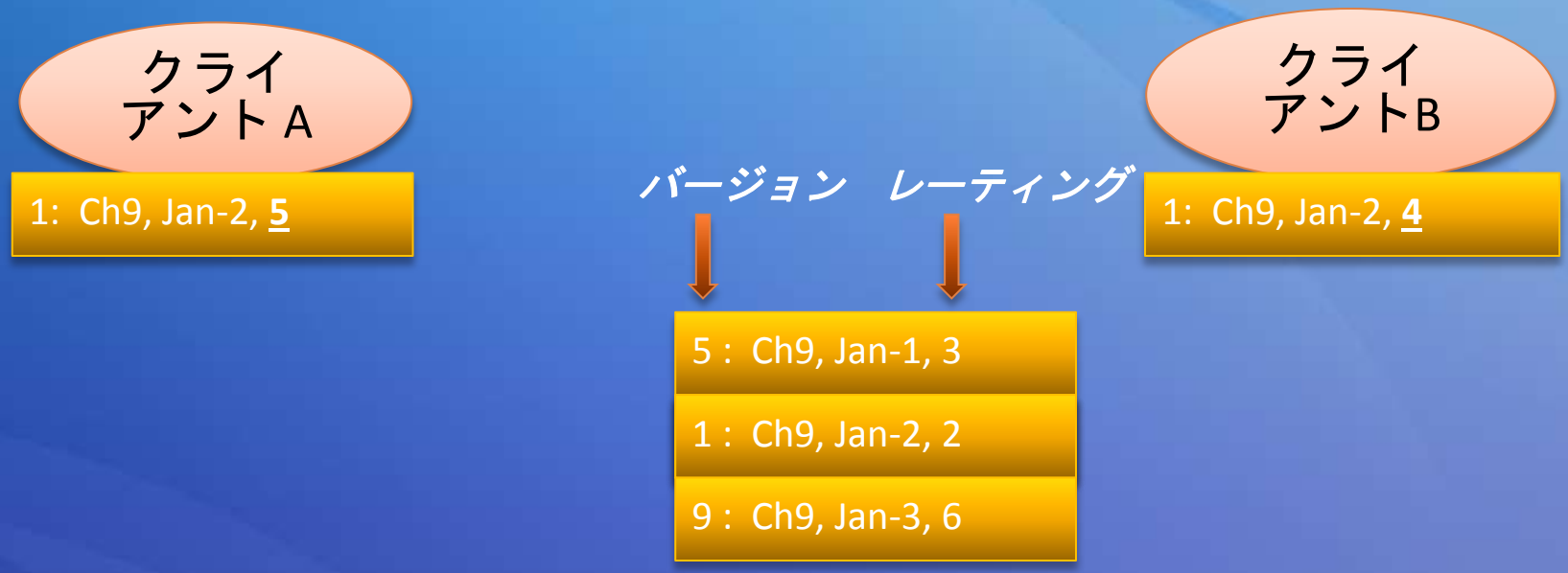
- 数万台のサーバーとなると、それぞれのサーバーは全ノードの管理領域を把握しにくい
- ノードの離脱、追加を行った際に、管理領域の変更の影響をおさえる仕組みが必要



パーテション化のガイドライン

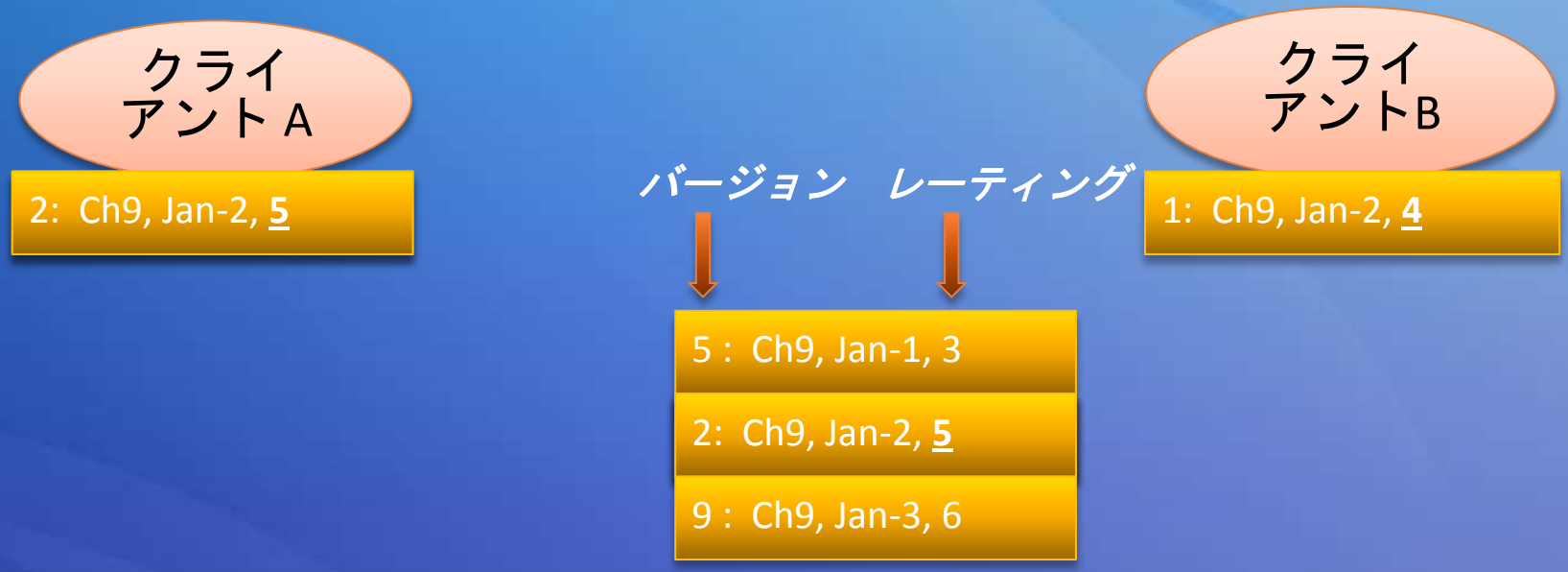
- パフォーマンス
  - クエリで共通に使われる値をパーテションキーとする
    - 同じパーテションキー値を持つエンティティはクラスタ化される
- スケーラビリティ
  - パーテションのトラフィックはモニタされている
  - 自動的にパーテションはロードバランスされる
    - それぞれのパーテションは異なるストレージノードに格納される可能性がある
    - アプリケーションのトラフィック要件に合うようにスケールさせるべき
  - パーテションが多ければロードバランスはされやすい

同時更新における考慮



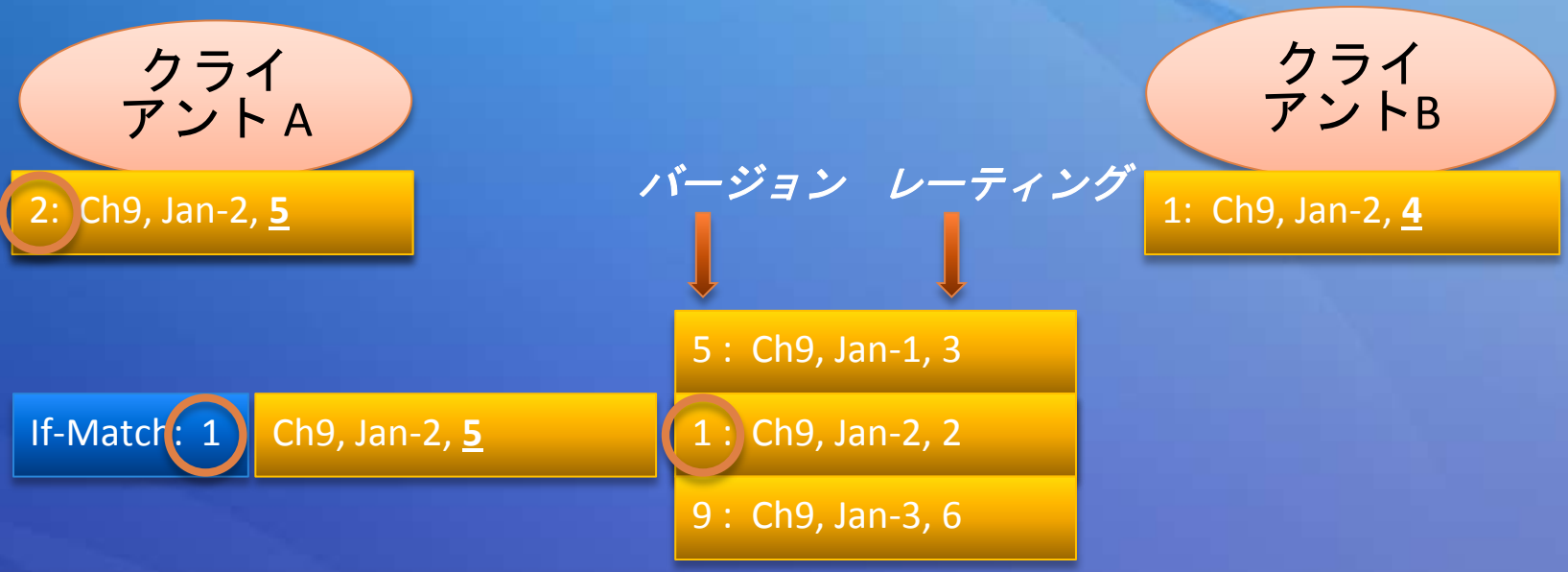
- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守

同時更新における考慮



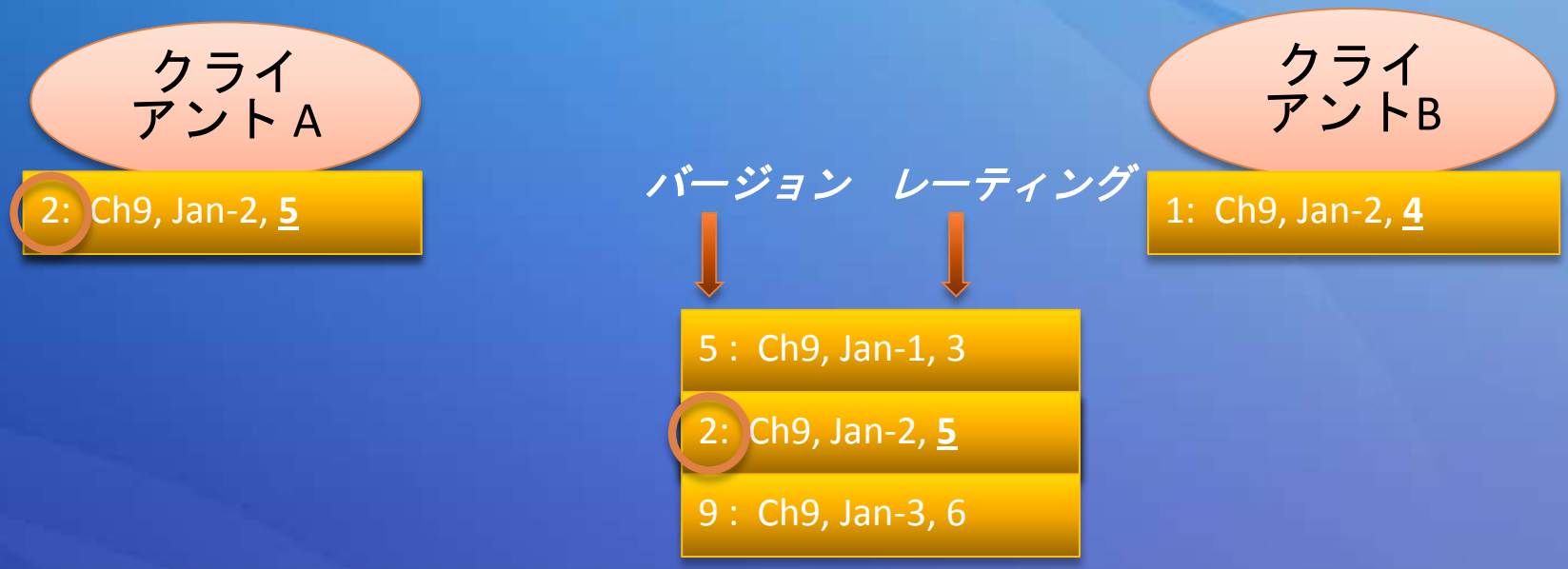
- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守
- Entityをローカルで更新 – レーティングの変更

同時更新における考慮



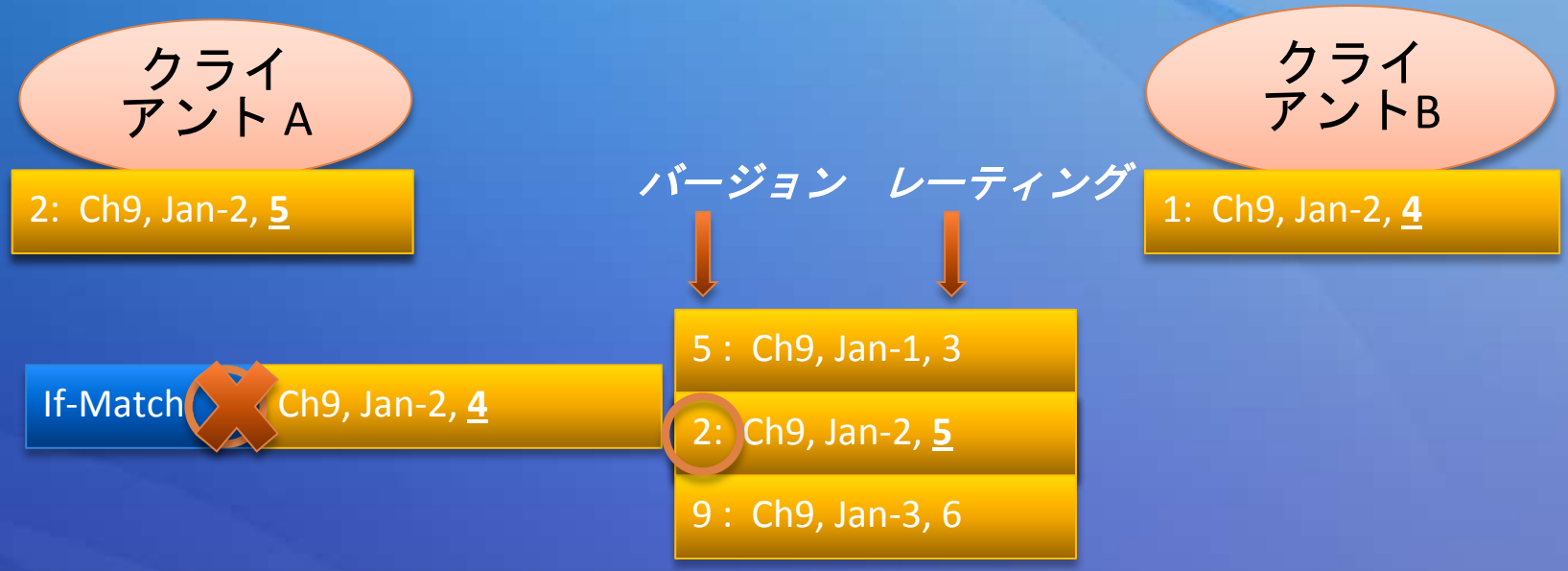
- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守
- Entityをローカルで更新 – レーティングの変更
- バージョンチェックとともに更新を送信 - Etag に対する IF-Match

同時更新における考慮



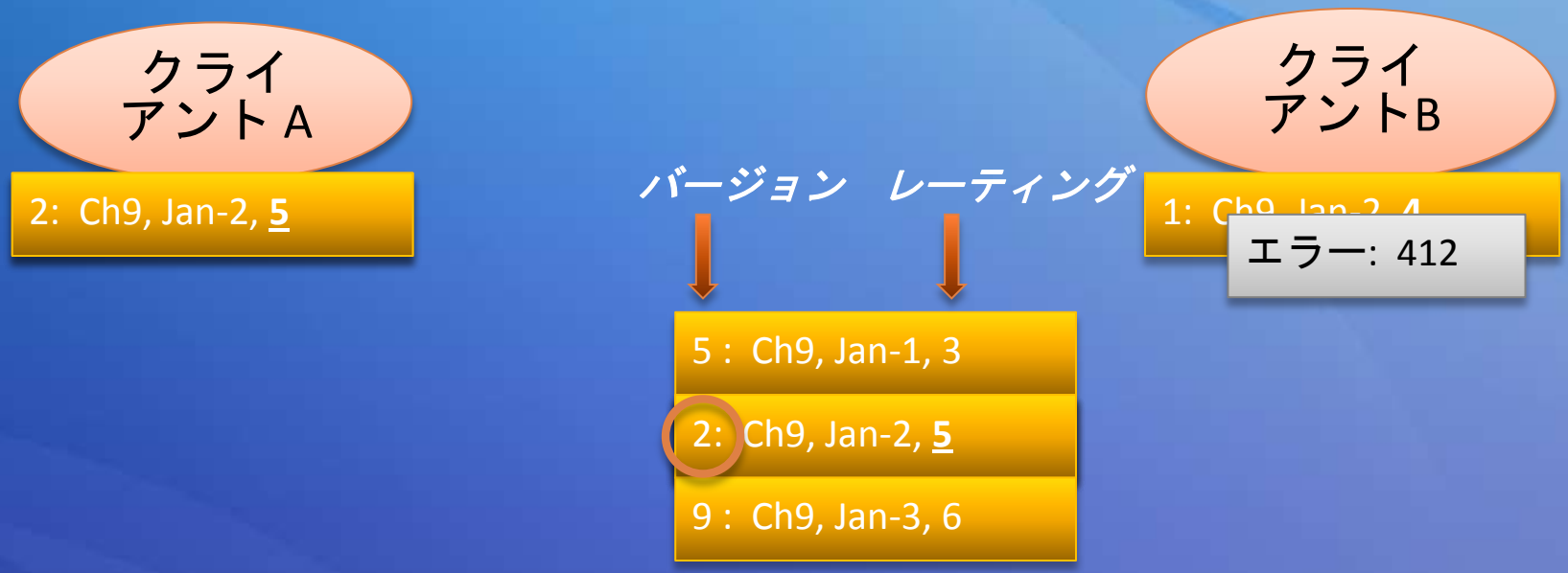
- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守
- Entityをローカルで更新 – レーティングの変更
- バージョンチェックとともに更新を送信 - Etag に対する IF-Match
- バージョンがマッチすれば成功、クライアントAのバージョンを上げる

同時更新における考慮



- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守
- Entityをローカルで更新 – レーティングの変更
- バージョンチェックとともに更新を送信 - Etag に対する IF-Match
- バージョンがマッチすれば成功、クライアントAのバージョンを上げる
- バージョンがマッチしなければ前提条件に合わないので失敗

同時更新における考慮



- 標準的な HTTP メカニズムを使用 – Etag と If-Match
- Entityを取得 – システムがバージョンを Etag として保守
- Entityをローカルで更新 – レーティングの変更
- バージョンチェックとともに更新を送信 - Etag に対する IF-Match
- バージョンがマッチすれば成功、クライアントAのバージョンを上げる
- バージョンがマッチしなければ前提条件に合わないので失敗

先頭からN個のEntityを取得

.NET: LINQ Take(N) 関数

```
serviceUri = new Uri("http://<account>.table.core.windows.net");
DataServiceContext context = new DataServiceContext(serviceUri);

var allMessages = context.CreateQuery<Message>("Messages");
foreach (Message message in allMessages.Take(100))
{
    Console.WriteLine(message.Name);
}
```

REST: $top=N クエリ文字列

```
GET http://<serviceUri>/Messages?$top=100
```

ページネーション－継続用トークン

要求の送信

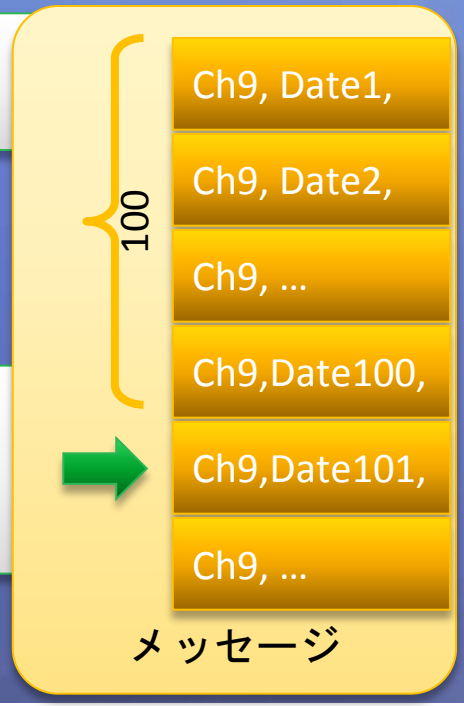
```
GET http://<serviceUri>/Messages?$filter=...&$top=100
```

返信ヘッダから継続用トークンを取得

```
x-ms-continuation-NextPartitionKey: Channel9
x-ms-continuation-NextRowKey: Date101
```

HTTP クエリパラメータをセット

```
GET http://<Uri>/Messages?$filter=...&$top=100
    &NextPartitionKey=Channel9
    &NextRowKey=Date101
```



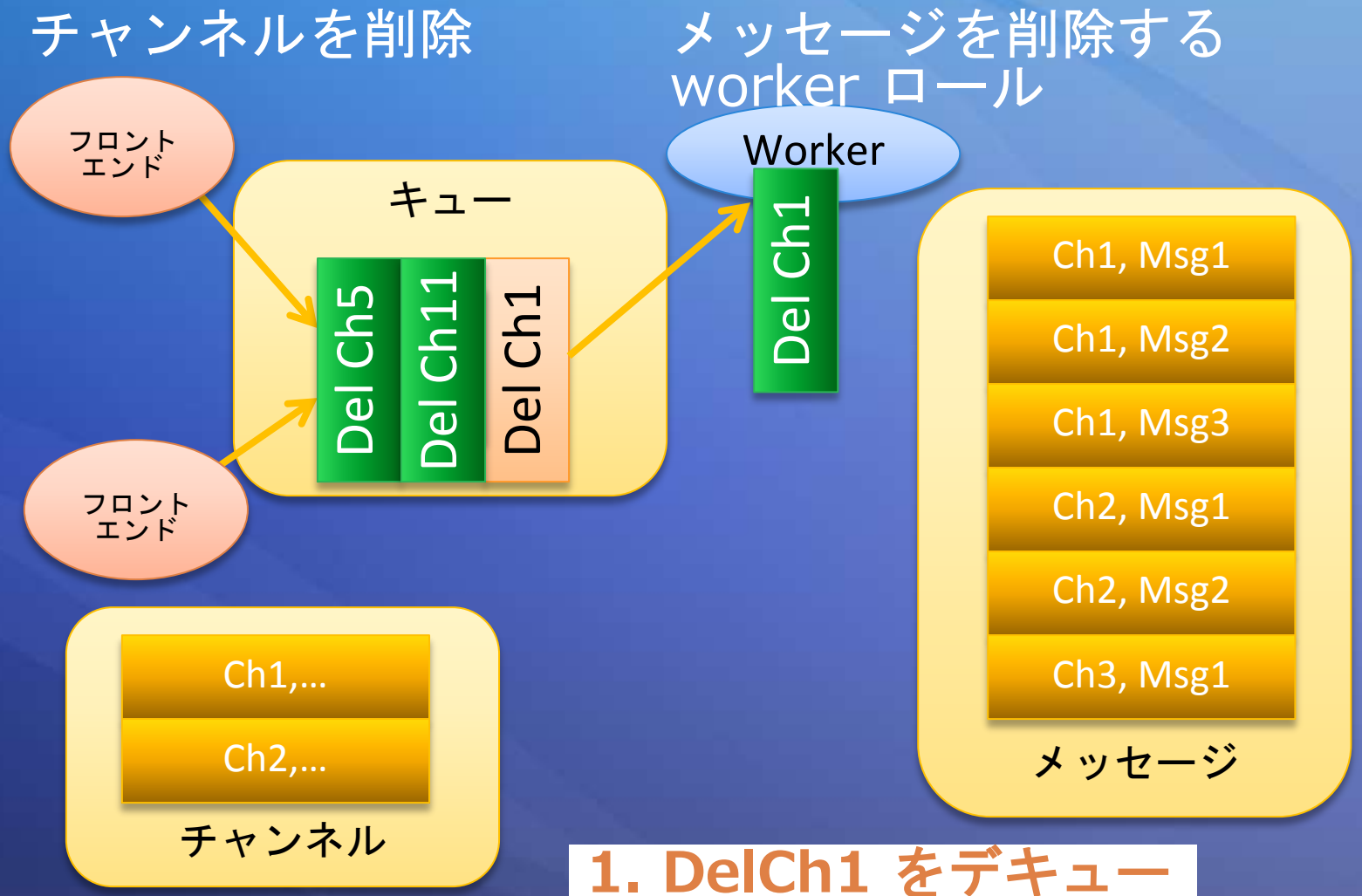
単一 Table の一貫性について

- 単一 Entity の CUD については ACID トランザクション
  - Insert/Update/Delete
- 単一パーテション内のクエリにはスナップショット分離
  - クエリの開始から一貫したビュー
  - ダーティリード（uncommitted）はない
  - 同時更新はブロックされない
- パーテション内かつ単一クエリの条件を満たす場合にのみスナップショット分離が有効

Table 間の一貫性について

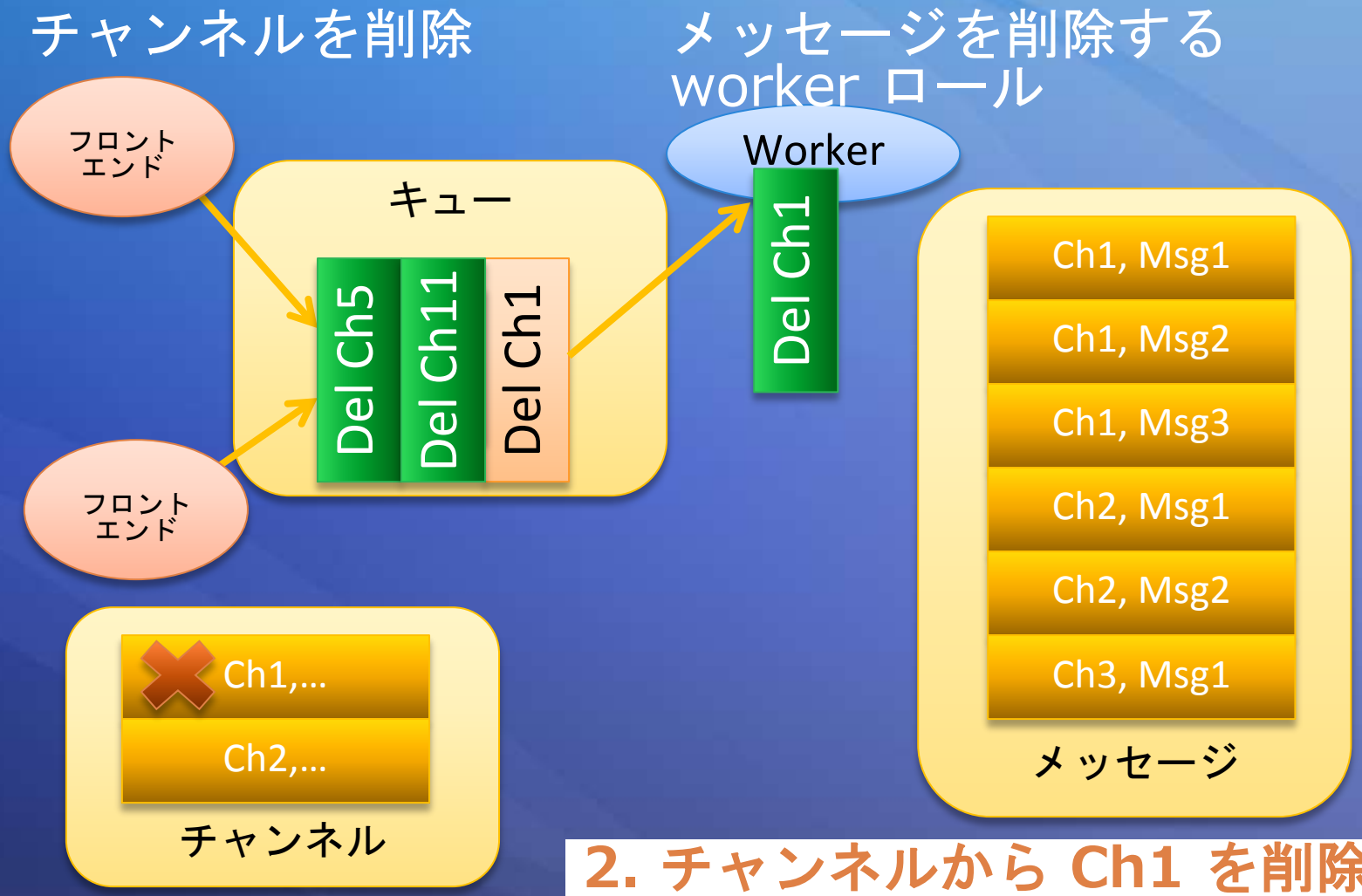
- アプリケーションが一貫性の保持の責任を持つ
  - 例：
    - ブログのあるチャンネルが削除されたら、そのチャンネルでのすべての発言を削除する
- 中間で失敗が起こりうる
  - 例：
    - メッセージ削除の途中でアプリケーションが失敗する
  - 操作の完了を正しく行えるよう Windows Azure キューを利用する

Table 間の一貫性について



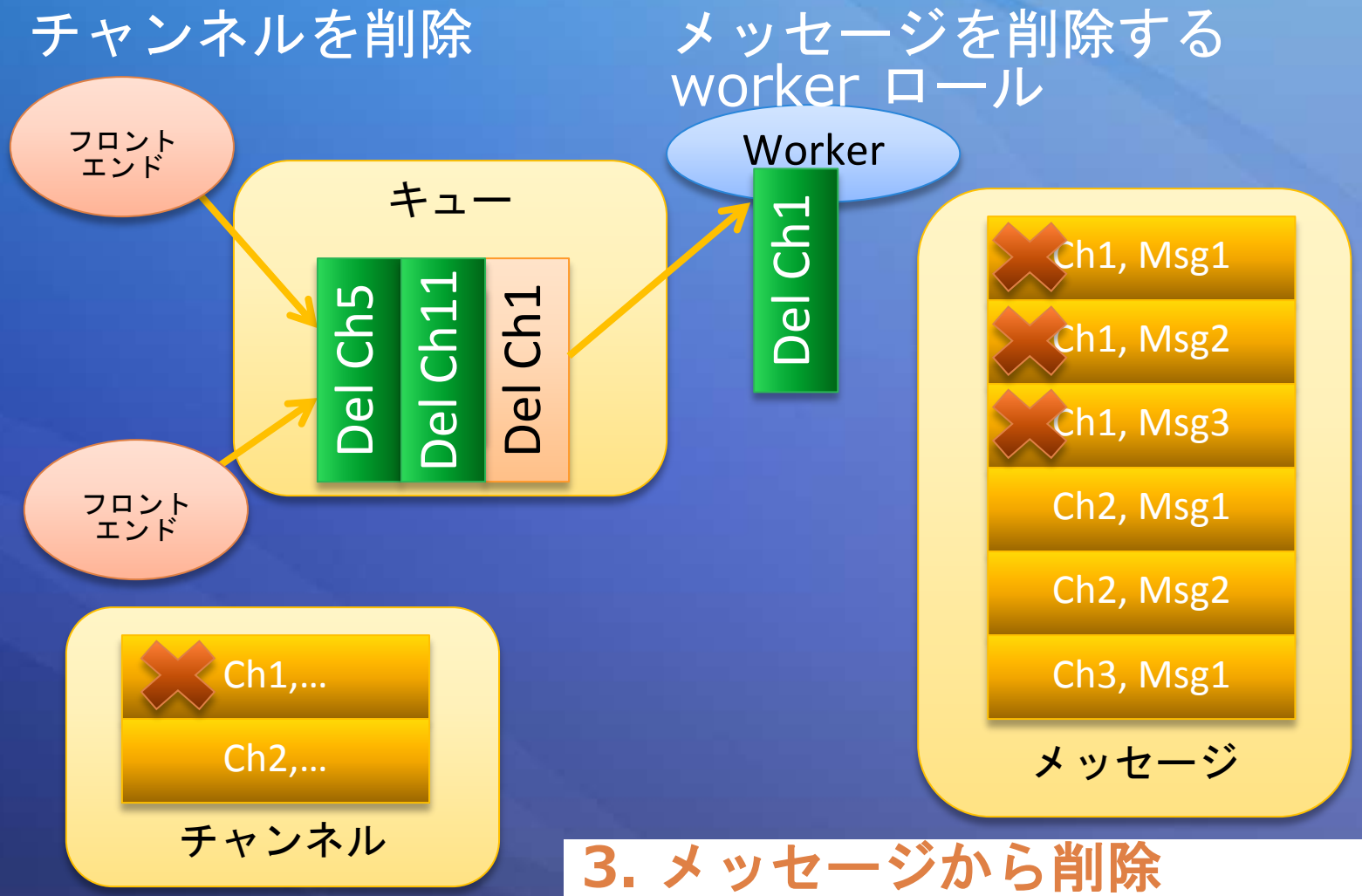
1. DelCh1 をデキュー

Table 間の一貫性について



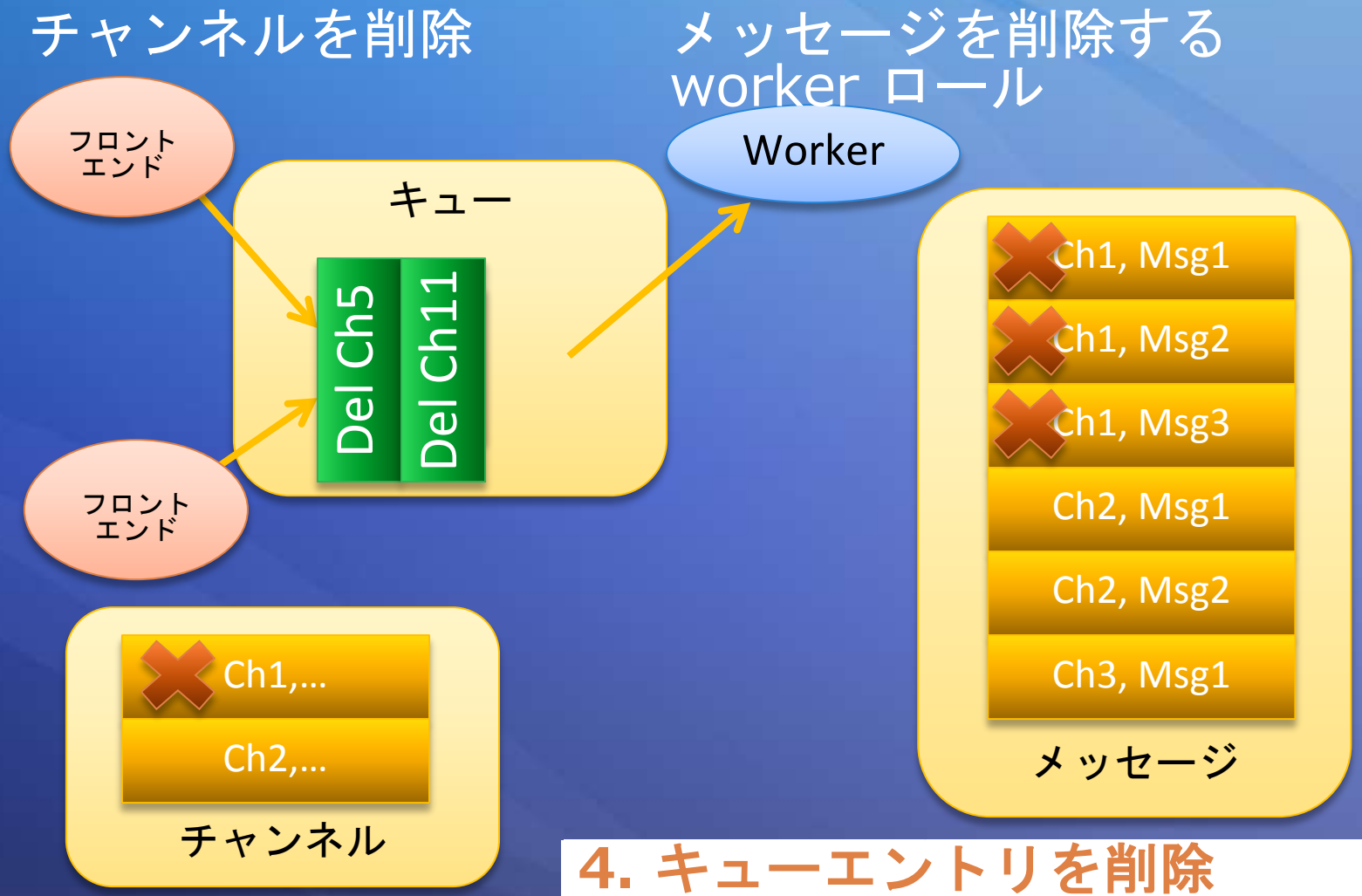
2. チャンネルから Ch1 を削除

Table 間の一貫性について



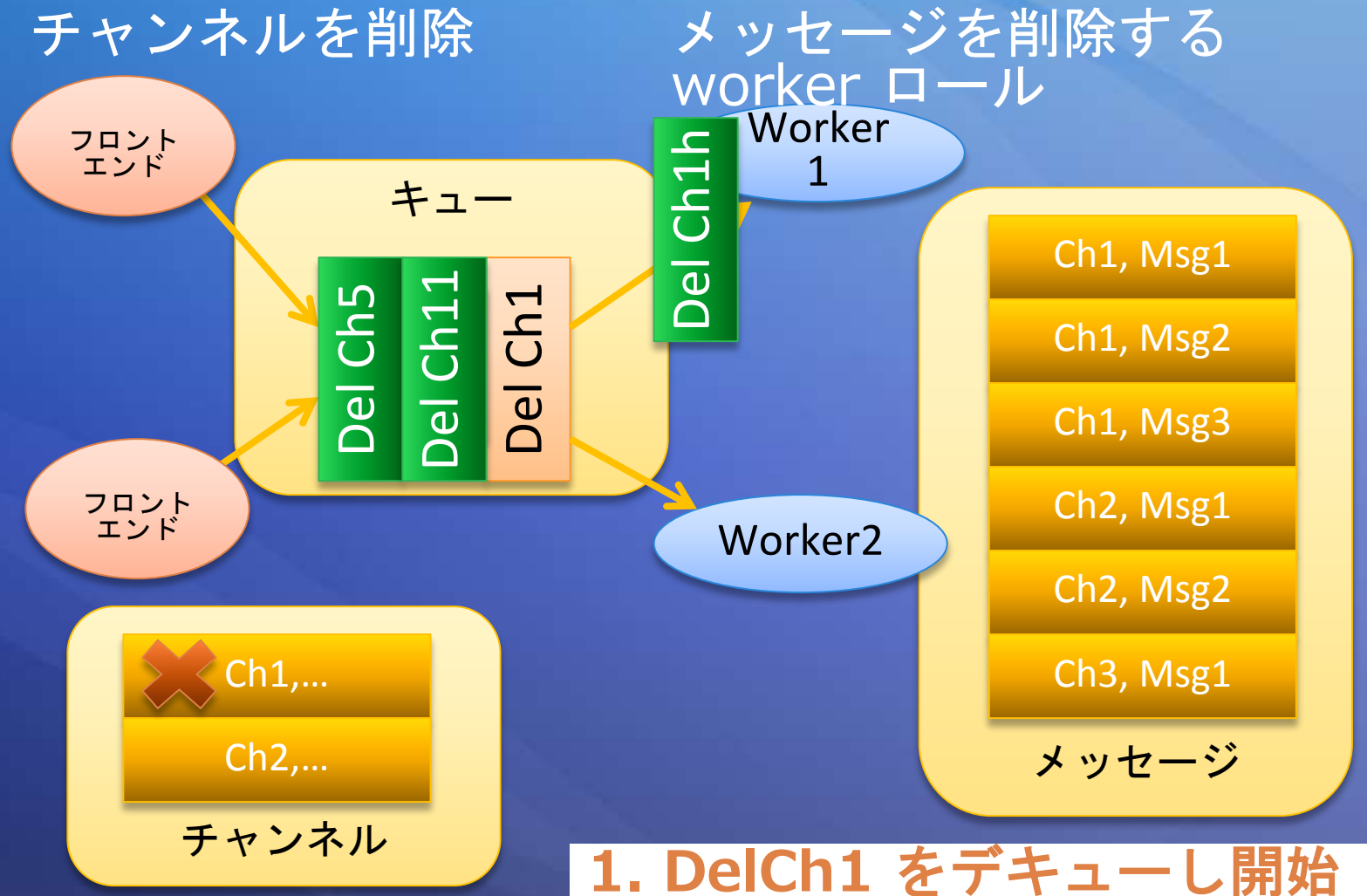
3. メッセージから削除

Table 間の一貫性について



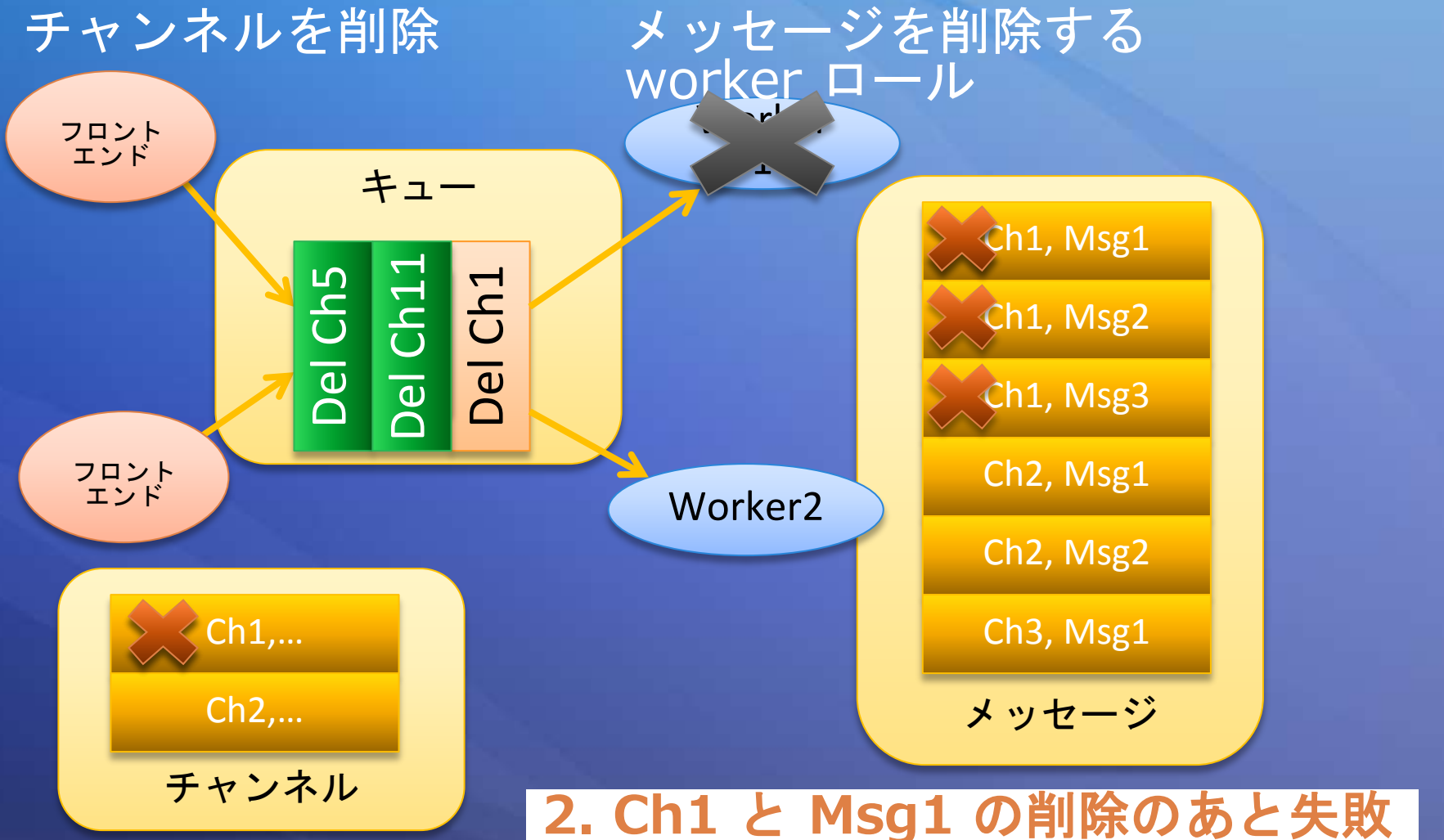
4. キューエントリを削除

削除の失敗時



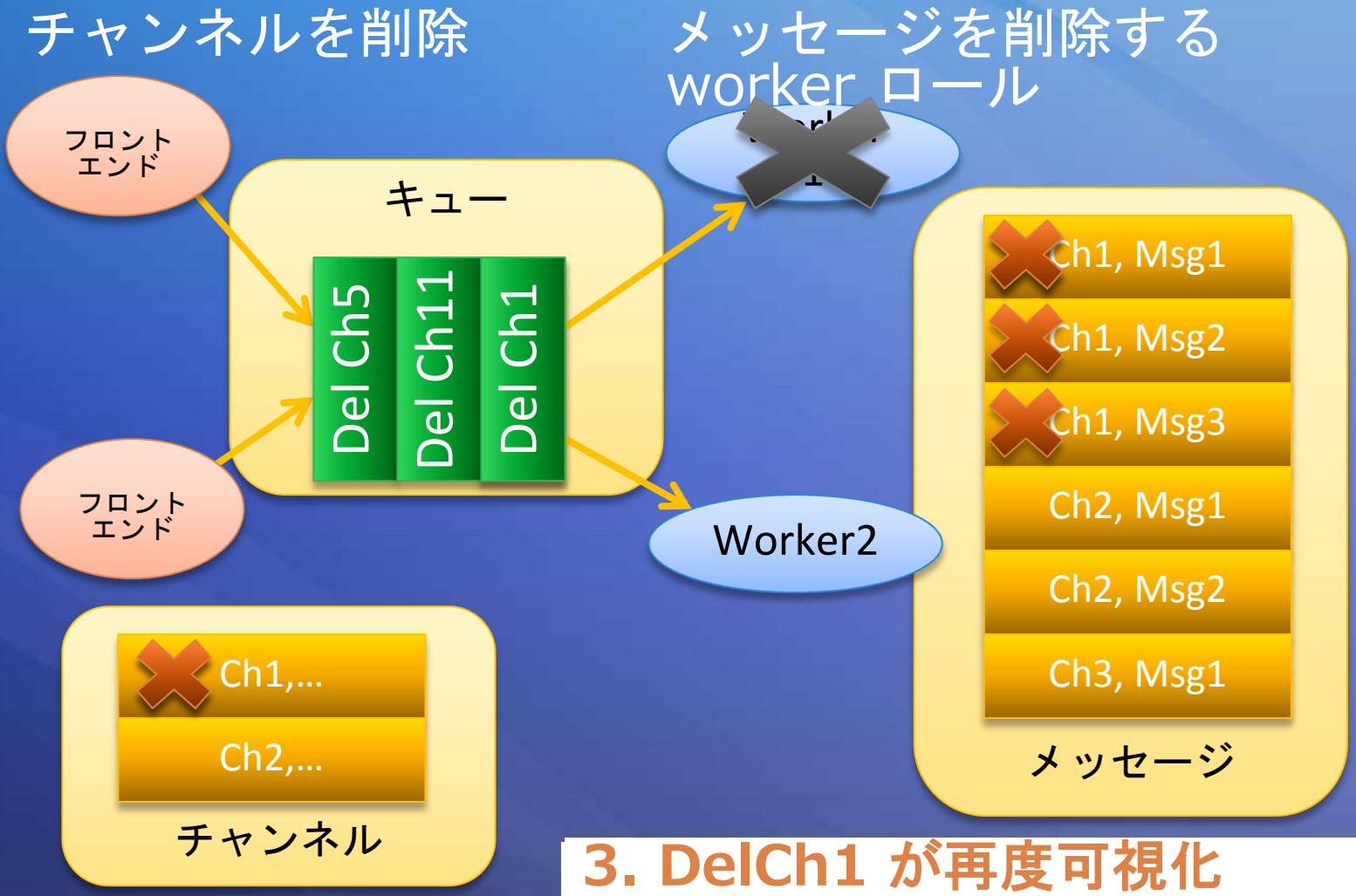
1. DelCh1 をデキューし開始

削除の失敗時



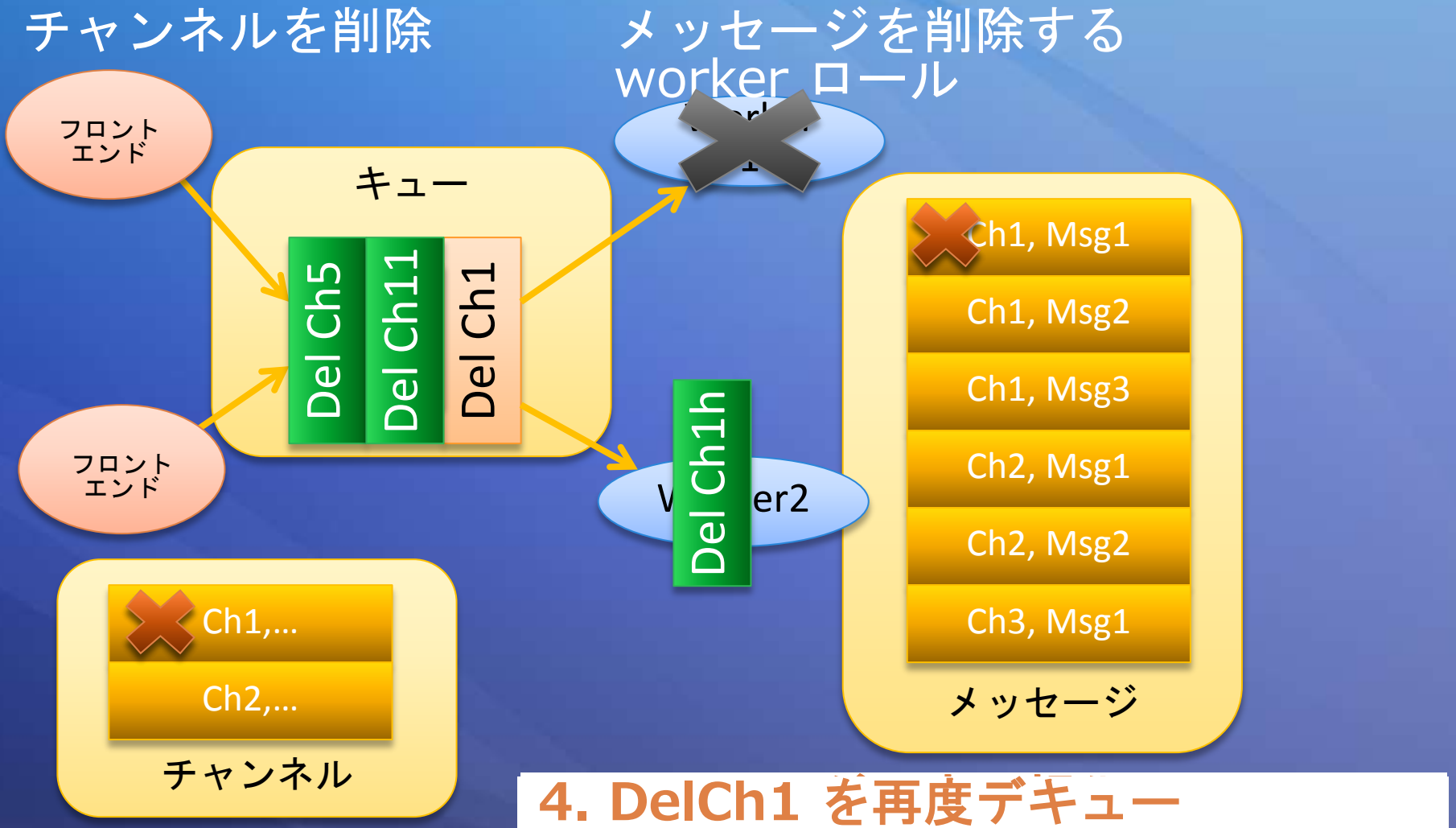
2. Ch1 と Msg1 の削除のあと失敗

削除の失敗時

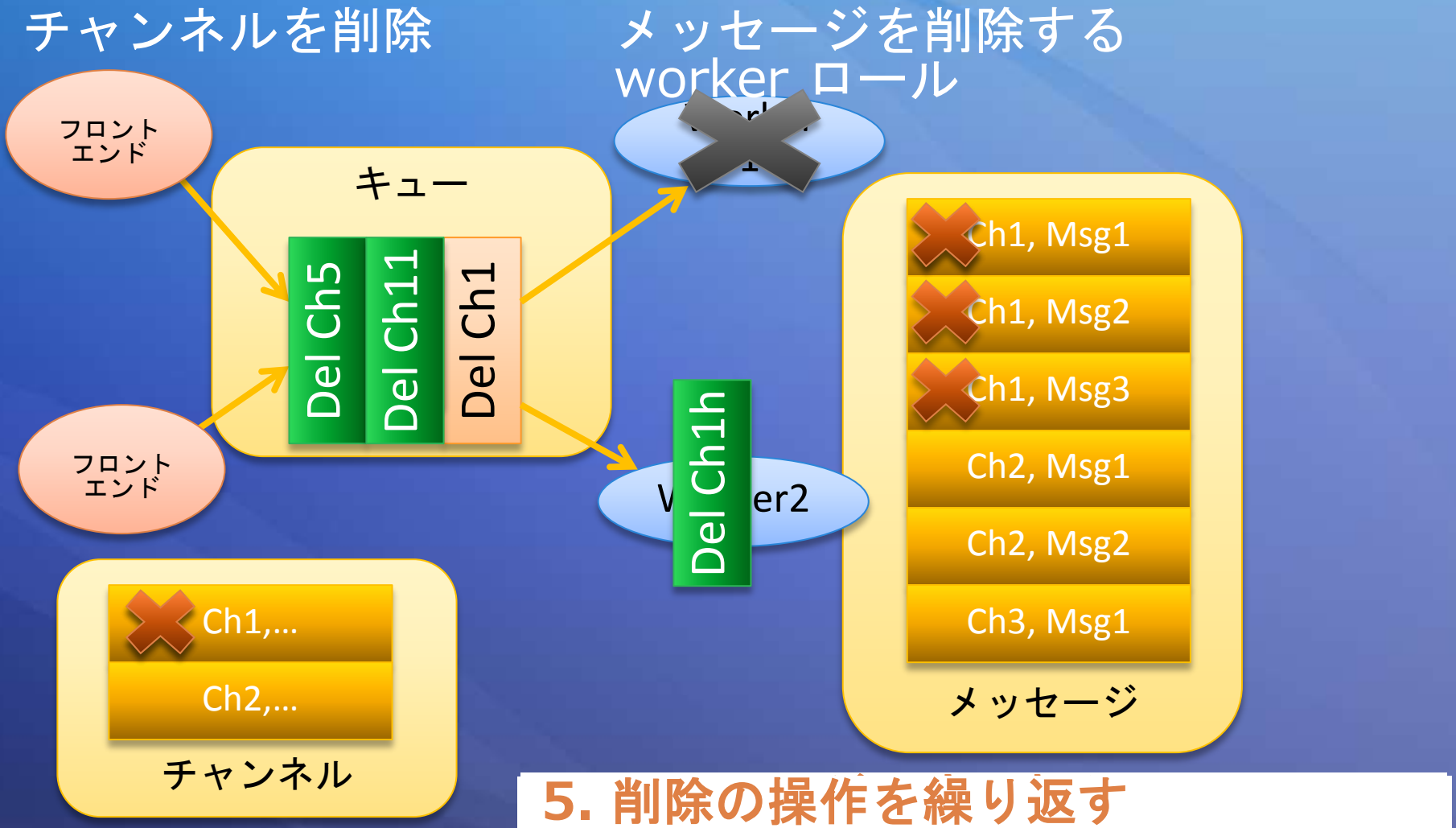


3. DelCh1 が再度可視化

削除の失敗時

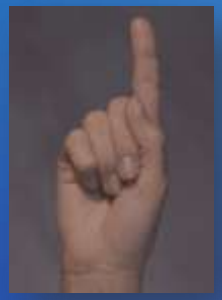

チャンネルを削除
メッセージを削除する worker ロール
フロント エンド
フロント エンド
キュー
Del Ch5
Del Ch11
Del Ch1
Del Ch1h
Ch1,...
Ch2,...
チャンネル
Ch1, Msg1
Ch1, Msg2
Ch1, Msg3
Ch2, Msg1
Ch2, Msg2
Ch3, Msg1
メッセージ
4. DelCh1 を再度デキュー

削除の失敗時



5. 削除の操作を繰り返す

アジェンダ



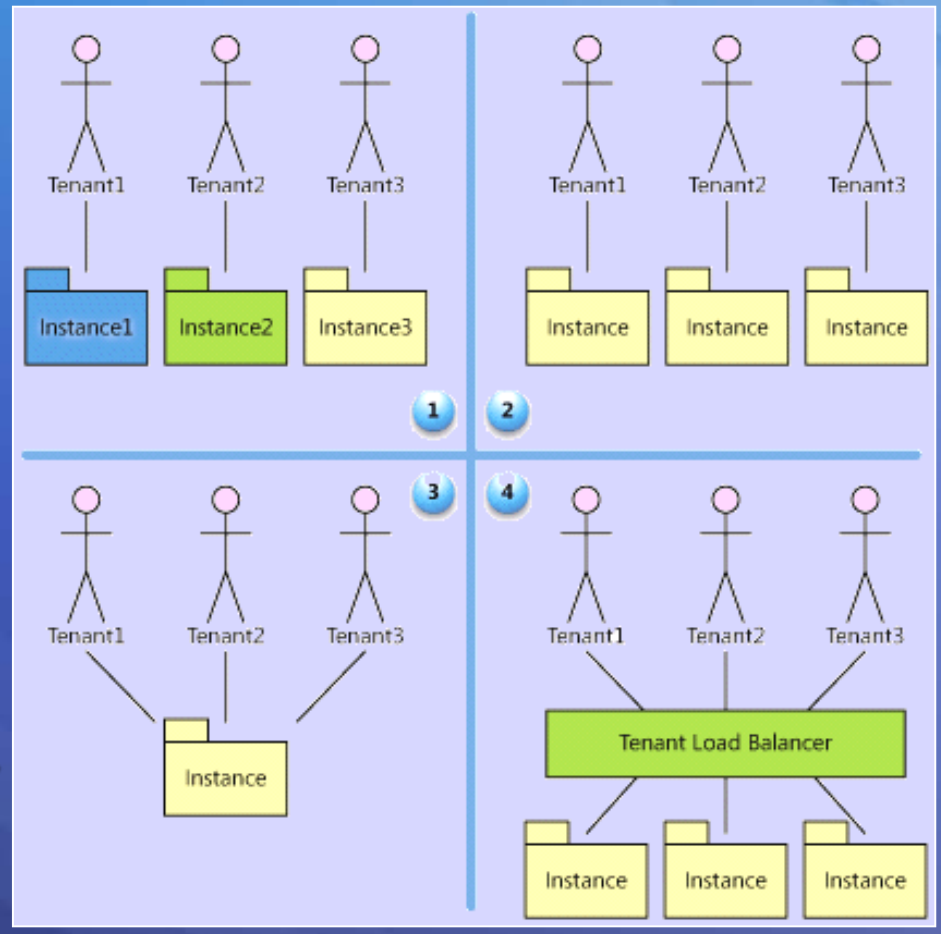
Windows Azureのロール、Tableの
活用方法

SaaS アプリケーション
アーキテクチャ設計の着眼点

マルチテナントにおける
データベース設計の選択肢

SaaS プロバイダの成熟度モデル

1.
アドホック / カスタム

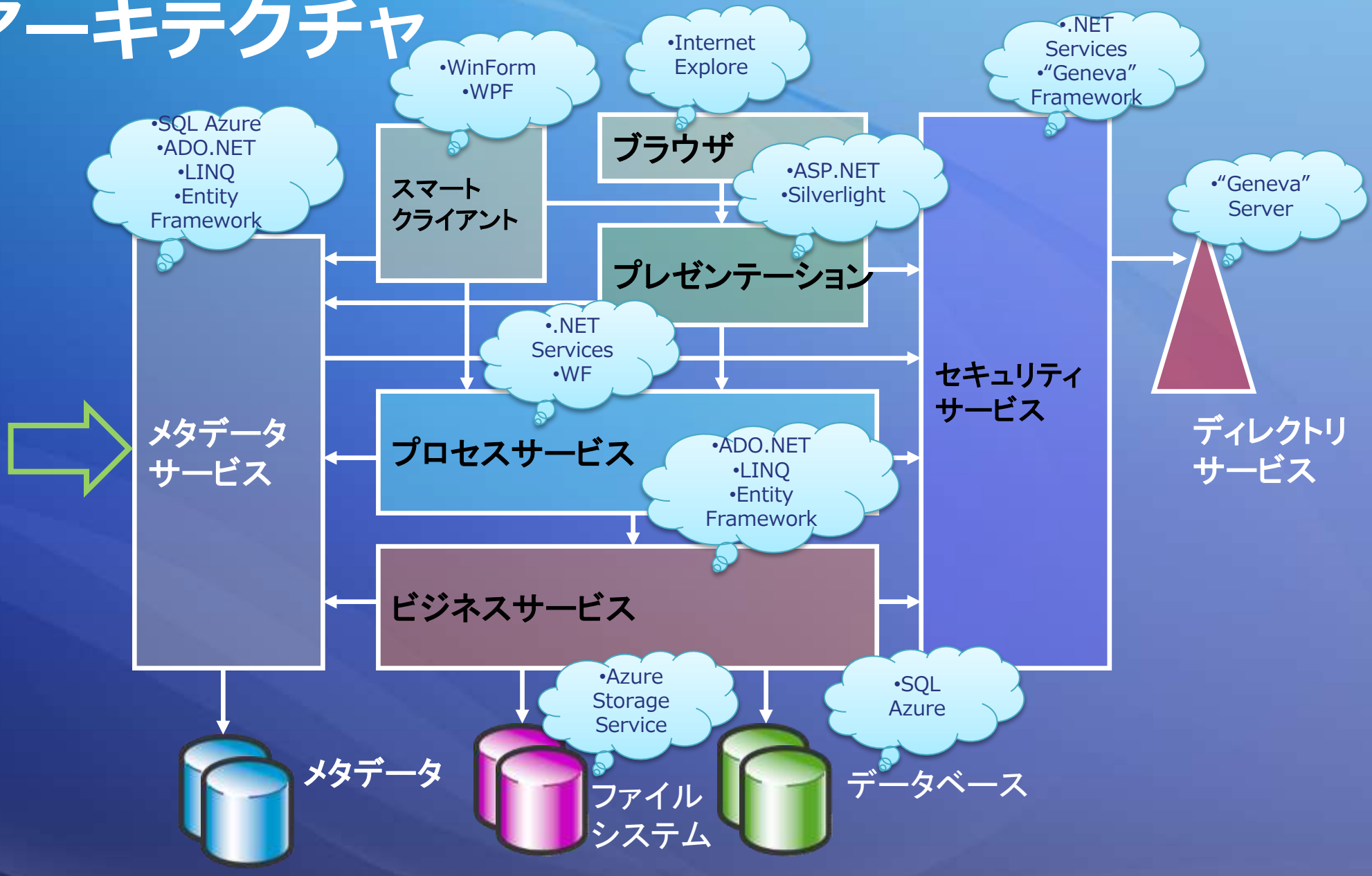


2.
構成可能
（シングルテナント）

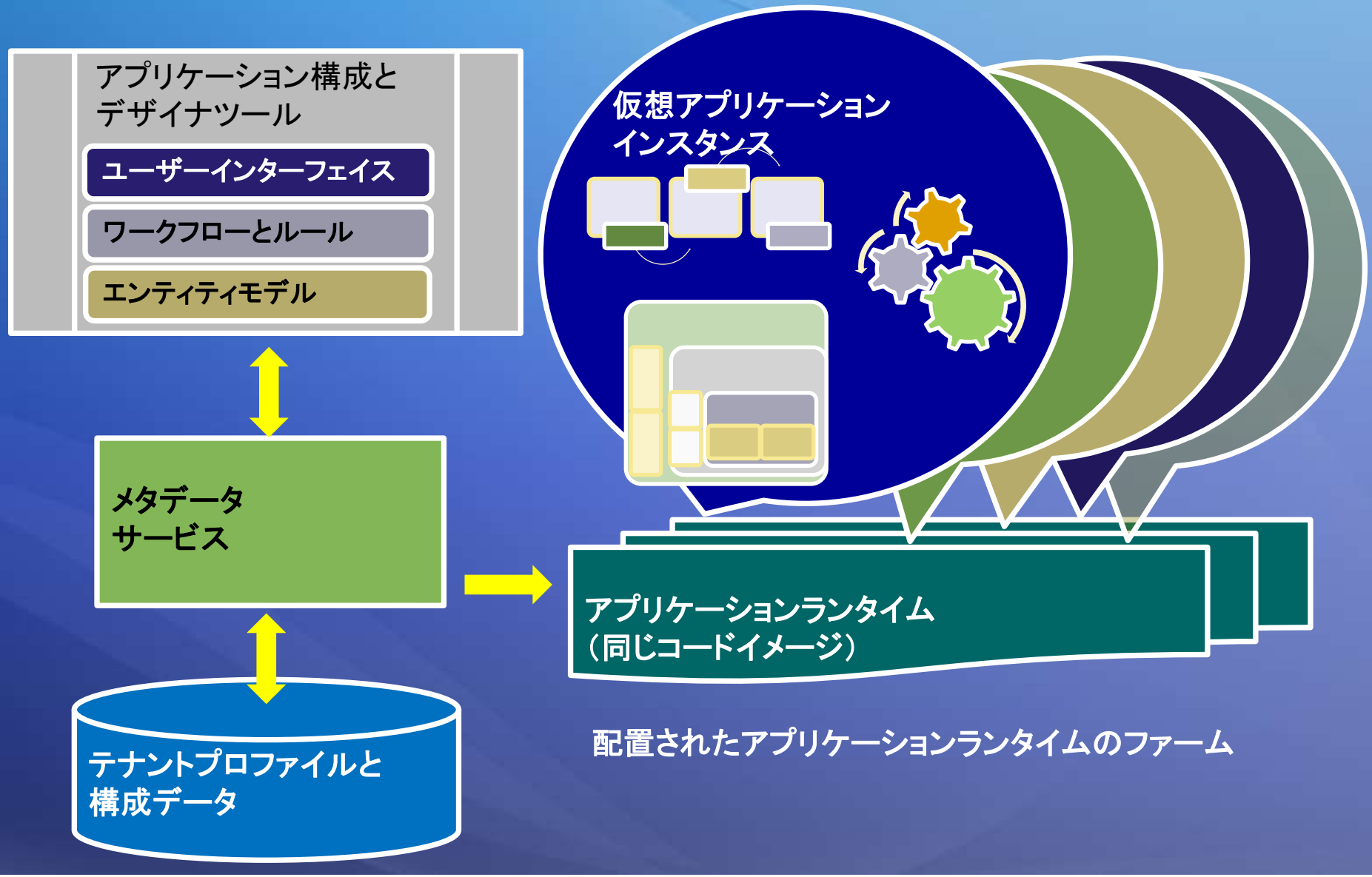
3.
構成可能
マルチテナント

4.
構成可能
マルチテナント
スケーラブル

ハイレベル SaaS アプリケーション アーキテクチャ



メタデータ駆動アーキテクチャ

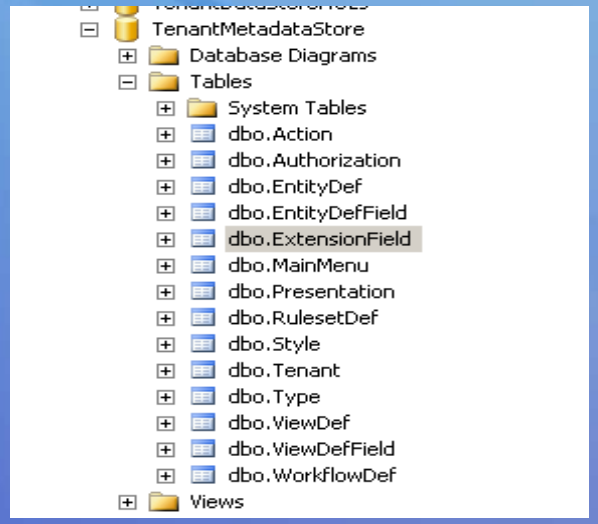


メタデータ駆動アーキテクチャ

- メタデータストア

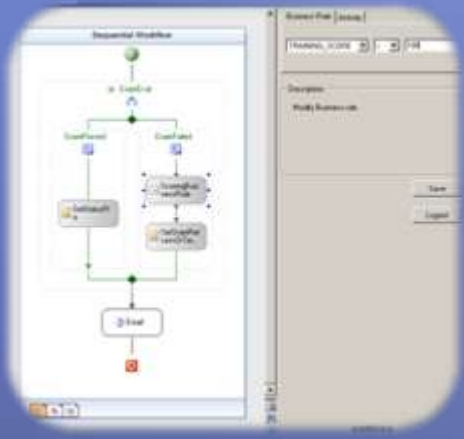
Table - dbo.ExtensionField | Summary

	id	tenantId	entityDefId	typeId	name
▶	000-3b4d72c6f69b	ed5586d5-7588-...	750438de-31a2-...	fa564e7c-f59b-...	location
	fb1a2963-2b43-...	a1eaebad-7e7c-...	750438de-31a2-...	8863152f-f10e-...	jobLevel
	3c93e4f4-1af1-...	a1eaebad-7e7c-...	750438de-31a2-...	fa564e7c-f59b-...	location
	3b78b04e-d9c7-...	765279a3-75c9-...	750438de-31a2-...	fa564e7c-f59b-...	location
	23da50c6-f37b-...	46f83df3-21c5-...	750438de-31a2-...	fa564e7c-f59b-...	location
*	NULL	NULL	NULL	NULL	NULL



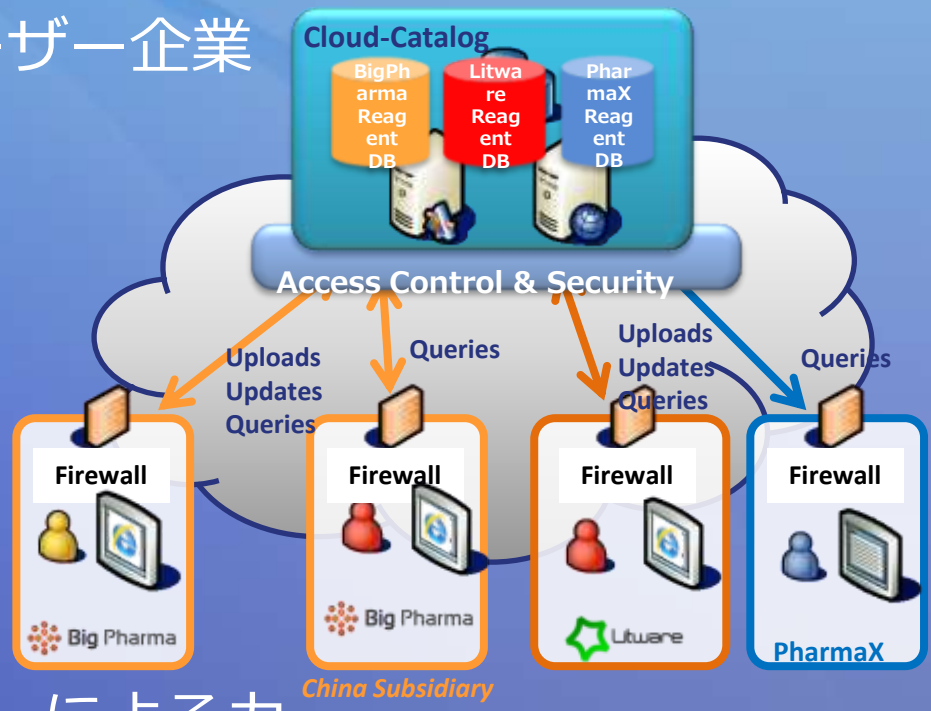
- ポリフォーミックな実行環境

```
static public EntityField[] GetEntityDefFields(string tenantAlias, string entityDefName)
{
    Database db = DatabaseFactory.CreateDatabase(Constants.DB.TenantMetadataStore);
    List<EntityField> list = new List<EntityField>();
    using (DbCommand command = db.GetStoredProcCommand("dbo.GetEntityDefFields"))
    {
        db.AddInParameter(command, "tenantAlias", DbType.String, tenantAlias);
        db.AddInParameter(command, "entityDefName", DbType.String, entityDefName);
        using (IDataReader reader = db.ExecuteReader(command))
        {
            while (reader.Read())
            {
                EntityField field = new EntityField();
                field.Id = reader.GetGuid(0);
                field.Name = reader.GetString(1);
```

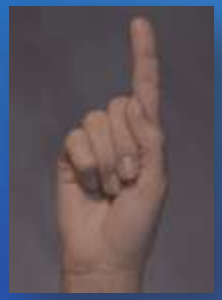


マルチテナントアーキテクチャ

- テナント
  - SaaSアプリケーションを使用するユーザー企業（サービスを提供する対象）
    - シングルインスタンス－シングルテナント
    - マルチインスタンス－マルチテナント
    - シングルインスタンス－マルチテナント
- 効率的
  - リソースの共有
- カスタマイズ可能
  - コンフィギュレーション（メタデータ）によるカスタマイズ
- スケーラブル
  - アプリケーションがインターネット規模で動作



アジェンダ



Windows Azureのロール、Tableの
活用方法

SaaS アプリケーション
アーキテクチャ設計の着眼点

マルチテナントにおける
データベース設計の選択肢

データの保管場所について

- アプローチ：
  - テナントごとに固有のデータベース
  - 共有データベース、テナント固有スキーマ
  - 共有データベース、テナント共通スキーマ
- これらのアプローチが意味すること：
  - データの安全性と分離性
  - データモデルの拡張性
  - データのスケーリングとパーテショニング

マルチテナントデータのスペクトル

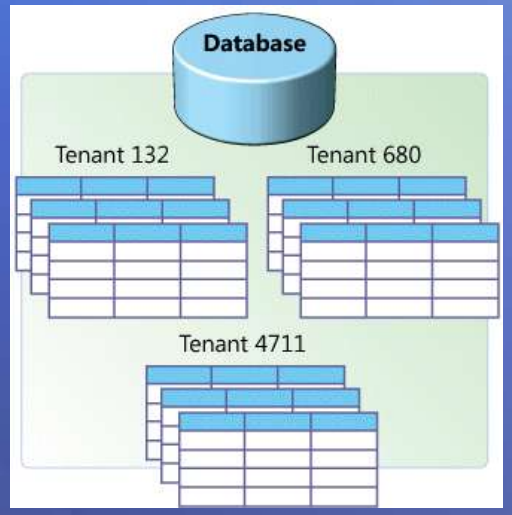
分離

共有

別々のDB

別々のスキーマ

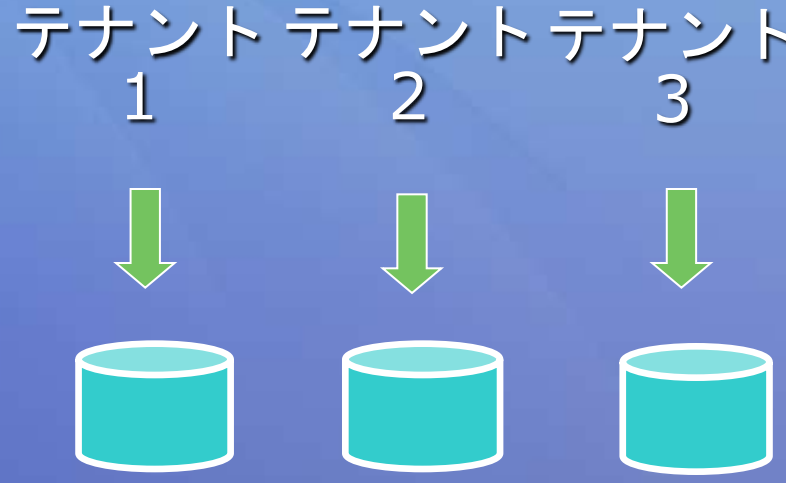
共有スキーマ



TenantID	CustName	Address
4	TenantID	ProductID	ProductNam
1	4	TenantID	Shipment	Date
6	1	4711	324965	2006-02-21
4	6	132	115468	2006-04-08
	4	680	654109	2006-03-27
		4711	324956	2006-02-23

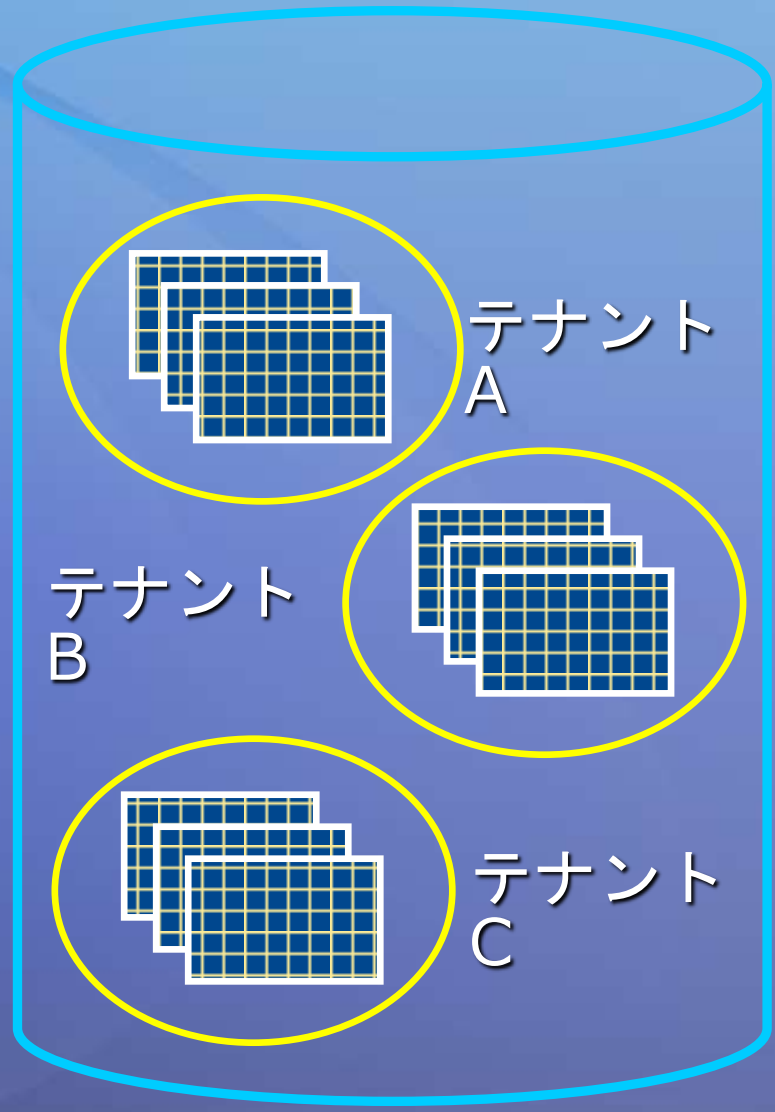
テナントごとに固有のデータベース

- アプローチ：
  - 各テナントごとのデータベースインスタンスをメタデータで識別
- 特長：
  - データモデル拡張を実装しやすい
  - テナントデータをリストアしやすい
  - よりセキュアな分離
- トレードオフ：
  - データベースサーバーごとのテナント数が小さい
  - 管理、バックアップコスト、データベースのインフラコストが高価
- 採用する条件：
  - テナントが特別なデータベース分離要件を持つ場合



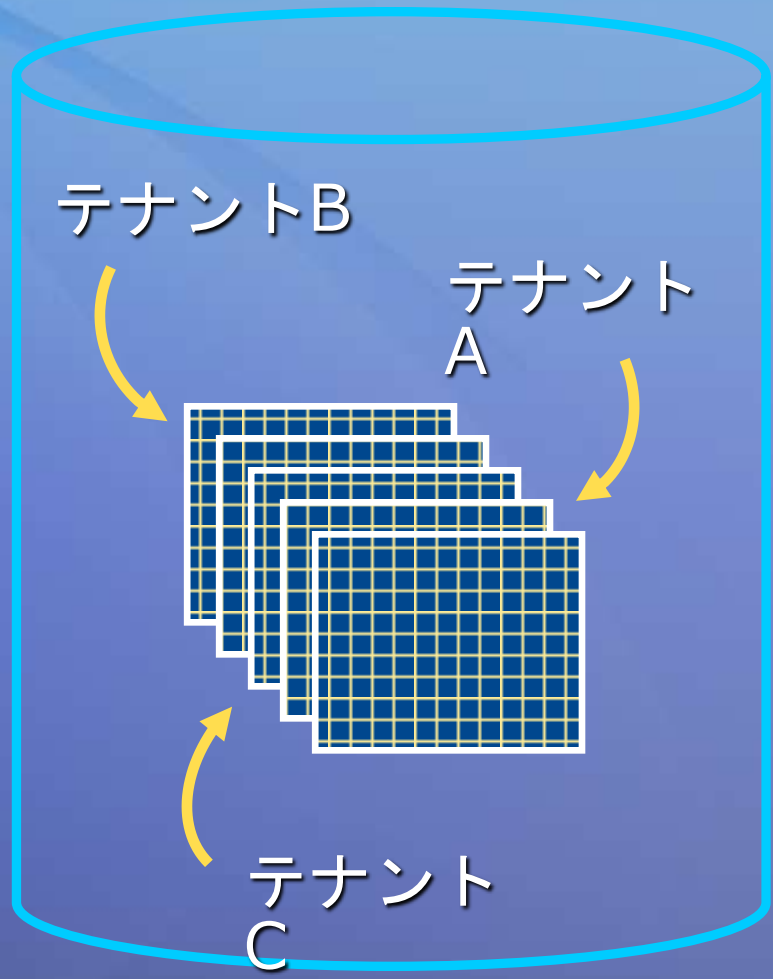
共有データベース、テナント固有スキーマ

- アプローチ：
  - それぞれのテナントが同一のデータベース内に独自のテーブルを持つ
- 特長：
  - データモデルの拡張がやさしい
  - セキュリティ分離は中程度
  - サーバーごとのテナントのスケーリングは中程度
- トレードオフ：
  - テナントデータのリストアがやや難しい
- 採用する条件：
  - アプリケーションごとのテーブルの数が少なめ（100程度）
  - サーバーごとのスケーリングが重要
  - 同一データベースにテナントデータが入ってもOK



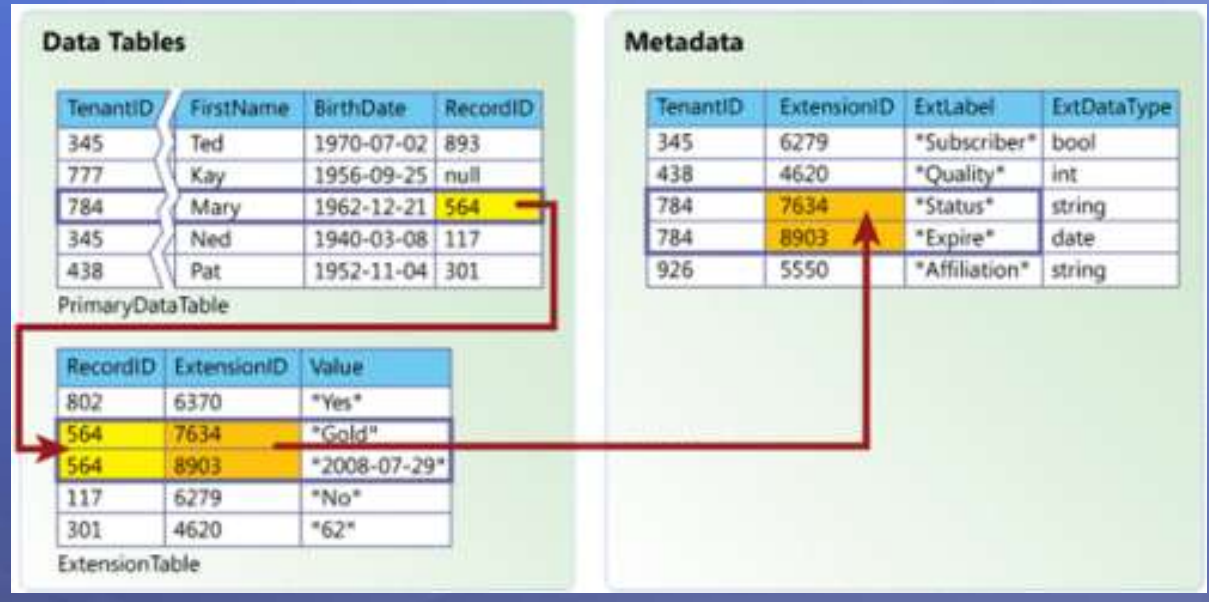
共有データベース、テナント共通スキーマ

- アプローチ：
  - すべてのテナントが同一データベースの同じテーブルを使う
- 特長：
  - サーバーごとのテナントのスケーリングが高い
  - 管理・バックアップのコストが小さい
- トレードオフ：
  - テナントデータのリストアが難しい
  - データモデル拡張がやや難しい
- 採用する条件：
  - サーバーごとのスケーリングが重要
  - 同一データベースにテナントデータが混在してもOK



テーブルのカスタム化

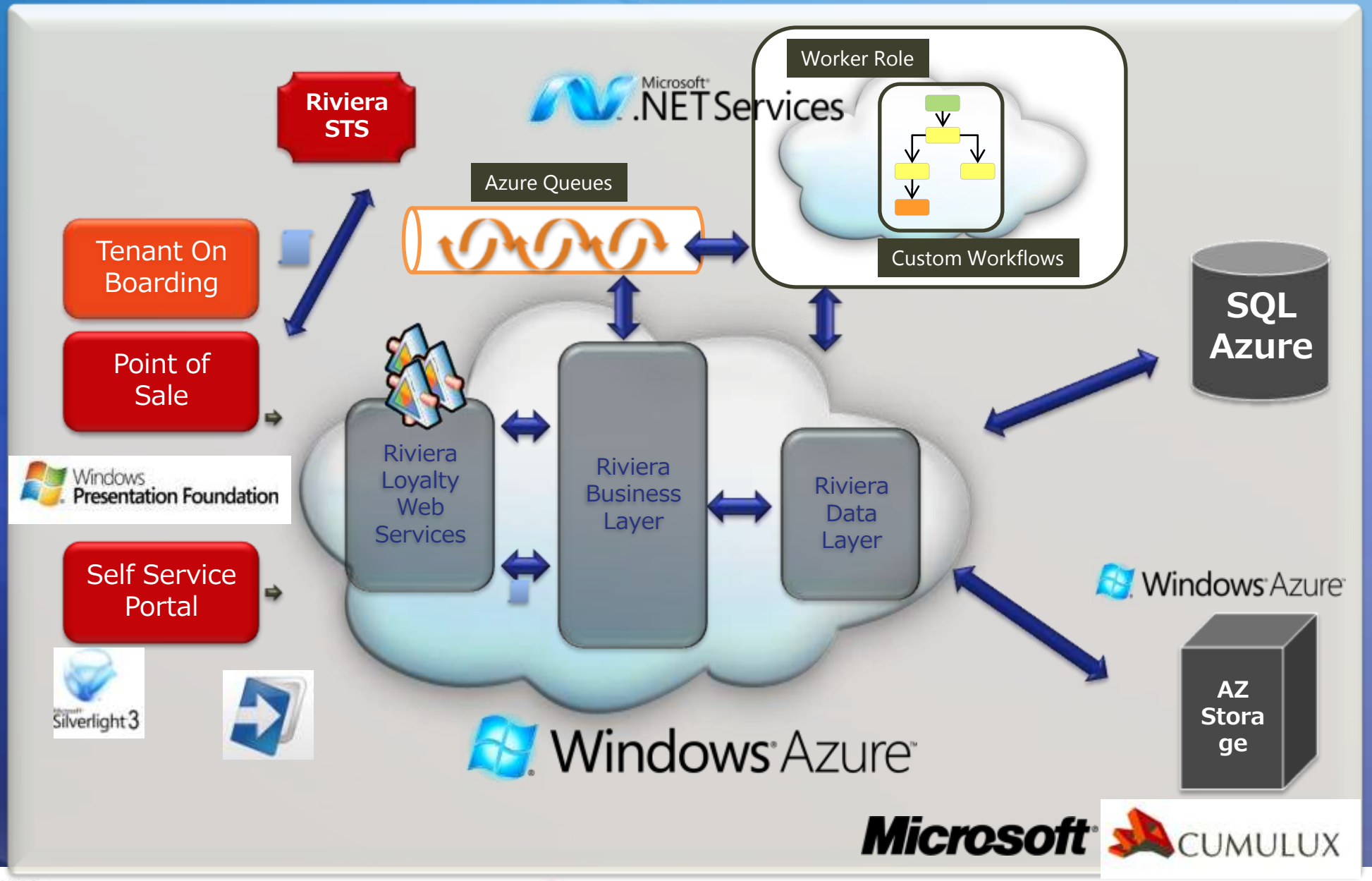
- 共通スキーマ + 拡張スキーマとのペア
  - 同じテーブルにすべてのテナントデータ
  - カスタムフィールドのための“無制限の”数/オプションを提供
  - 別のテーブルに Extension-value のペア
  - メタデータテーブルで拡張部分のデータラベルとデータ型を管理
- 特長:
  - カスタムフィールドのための“無制限の”数/オプション
- トレードオフ:
  - インデクス/クエリ/更新 の複雑性が増大し、遅延する
- 採用する条件:
  - テナントデータが混在してもよい場合
  - カスタムフィールドが価値の高い機能である場合
  - カスタムフィールドを予測するのが難しい場合



Demo

- マルチテナント データベース

“Riviera” 参照アーキテクチャ



Future
Technology Days


Microsoft®