クラウドデザインパターンから学ぶモダンなアプリケーション設計手法

～ Azure を例としたクラウドアプリケーションの設計 ～

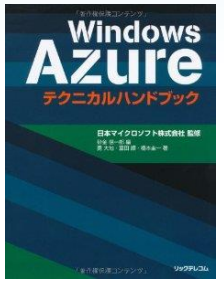
2014年9月1日

#jazug

野村総合研究所 勇大地

自己紹介

- 名前
  - 勇大地(@d_isami)
- 所属
  - 株式会社 野村総合研究所
- その他
  - Microsoft MVP for Microsoft Azure 2010～
  - Azure のコミュニティ JAZUG の運営 http://r.jazug.jp/
  - 「Windows Azure テクニカルハンドブック」執筆
  - 「クラウドデザインパターン」監訳



本セッションのゴール

- クラウドとオンプレミスの違いを理解する
- クラウドの特性を生かす設計観点を理解する
- クラウドデザインパターンの概要を理解する

- はじめに
- クラウドとオンプレミスの違い
- クラウドデザインパターンについて
- まとめ

クラウドの特徴（の一部）

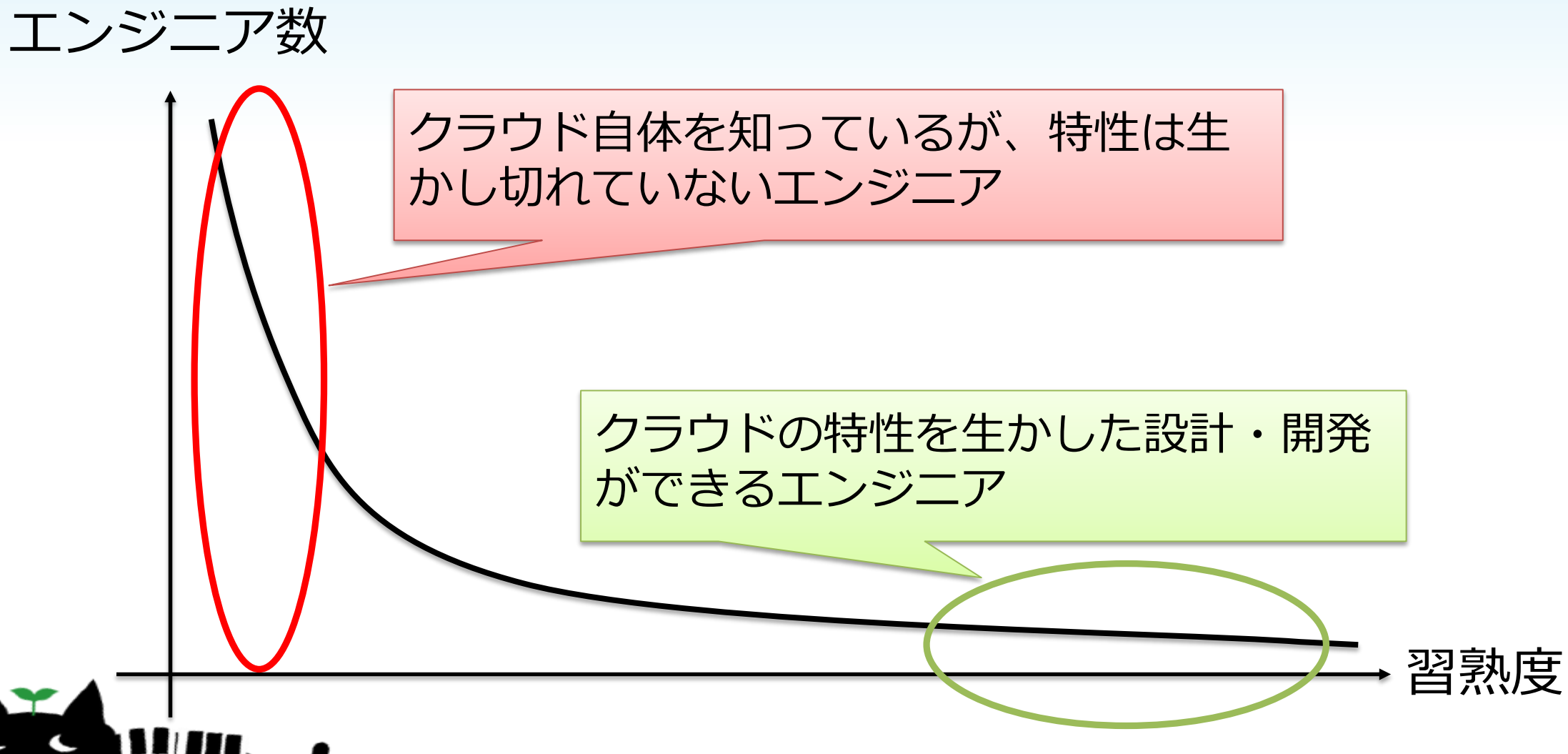
- リソースの即時調達、リソース破棄が容易
- 多様な外部サービスとの連携が多く、トランザクションの保証が難しい
- 利用する外部サービスが常に利用可能とは限らない
- オンプレミスに比べて監視が難しい

クラウド利用で良くある失敗

- 既存システムのクラウド移行を実施するが、システム移行が失敗
- クラウドに適したアプリケーション設計になっておらず、オートスケール等のクラウドの利点を享受できない
- クラウドを利用したが、当初想定したよりもコストが抑えられない

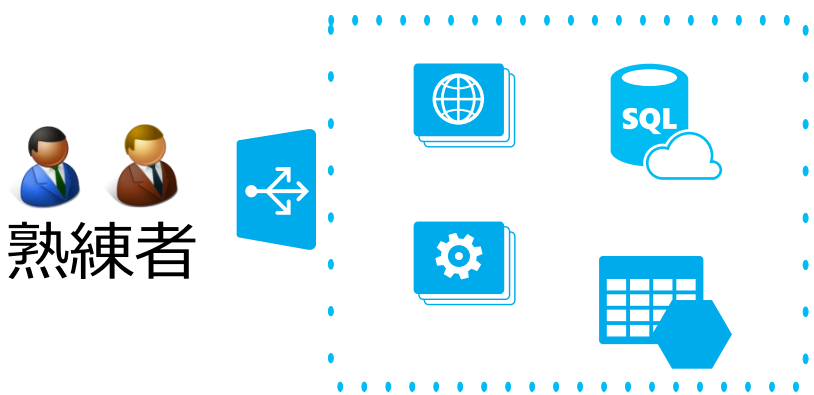
なぜこの様な問題が起きるのか？

クラウドのエンジニア数と習熟度分布

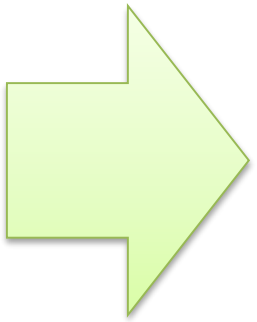


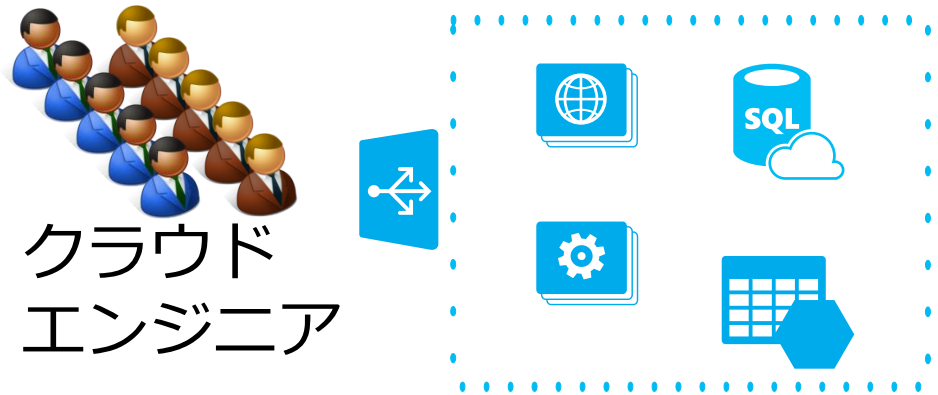
クラウド利用の体系化、形式知化が重要

- 体系化、形式知化により、複雑なシステムをクラウド上に構築する敷居が下がる



クラウドデザインパターン
普及前

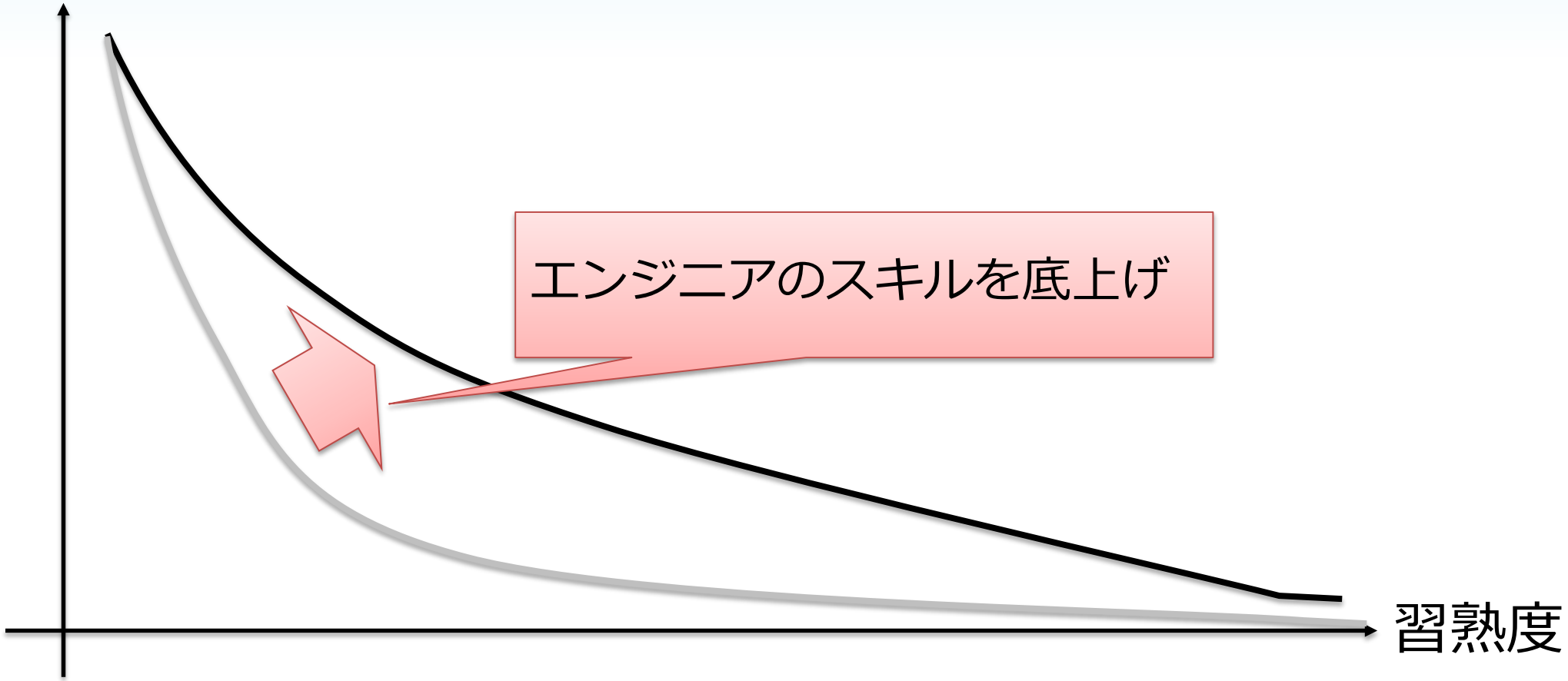




クラウドデザインパターン
普及後

クラウド利用の体系化、形式知化が重要

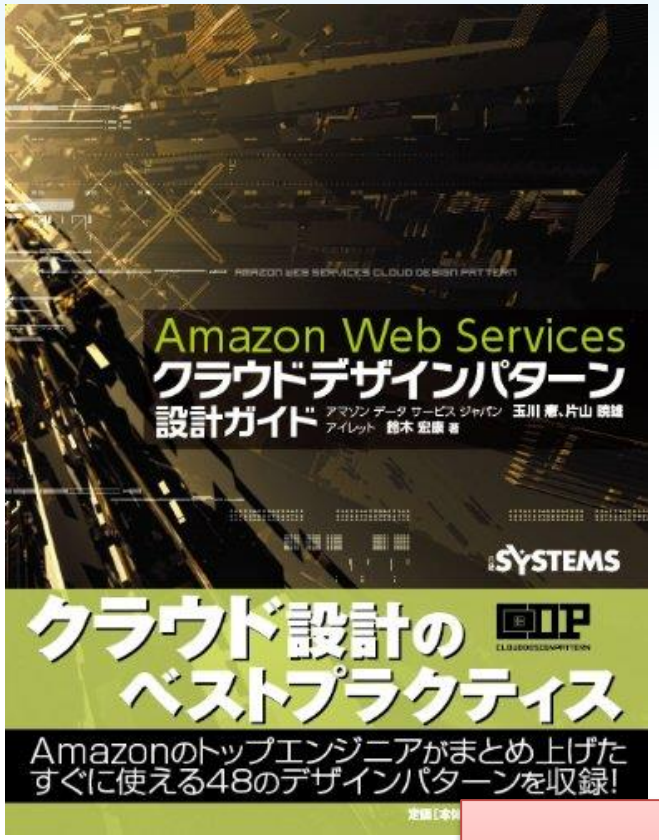
- 体系化、形式知化により、複雑なシステムをクラウド上に構築する敷居が下がる



- はじめに
- クラウドとオンプレミスの違い
- クラウドデザインパターンについて
- まとめ

クラウドデザインパターンの種類

- AWS, Azure それぞれでデザインパターンを公開



今回はこちらを解説

クラウドデザインパターンの観点

#	観点	概要
1	可用性	システムの稼働率向上
2	データ管理	複数サーバ間のデータ処理
3	設計及び実装	品質、総所有コストの改善
4	メッセージング	コンポーネントの疎結合化
5	管理および監視	リモートの監視強化
6	パフォーマンスおよび スケーラビリティ	ピーク変動、 マルチテナント対応
7	回復性	システム障害の検知、回復
8	セキュリティ	情報漏えい、紛失対策

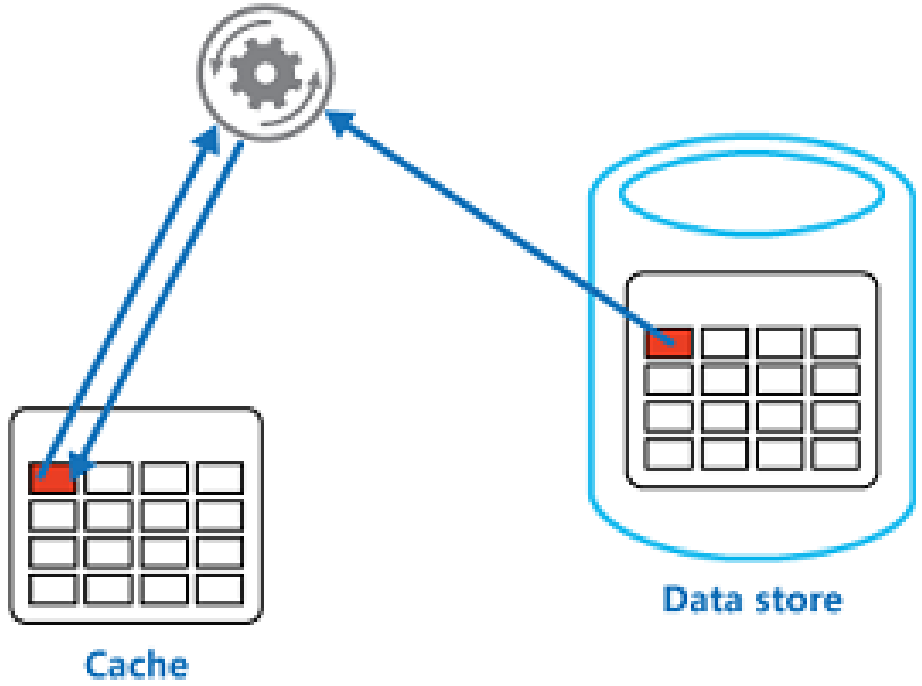
※ AWS のデザインパターンとは一部異なる

クラウドデザインパターン – Azure 編

1. Cache-Aside パターン
2. Circuit Breaker パターン
3. Compensating Transaction パターン
4. Competing Consumers パターン
5. Compute Resource Consolidation パターン
6. Command and Query Responsibility Segregation (CQRS) パターン
7. Event Sourcing パターン
8. External Configuration Store パターン
9. Federated Identity パターン
10. Gatekeeper パターン
11. Health Endpoint Monitoring パターン
12. Leader Election パターン
13. Materialized View パターン
14. Pipes and Filters パターン
15. Priority Queue パターン
16. Queue-Based Load Leveling パターン
17. Retry パターン
18. Runtime Reconfiguration パターン
19. Scheduler Agent Supervisor パターン
20. Sharding パターン
21. Static Content Hosting パターン
22. Throttling パターン
23. Valet Key パターン

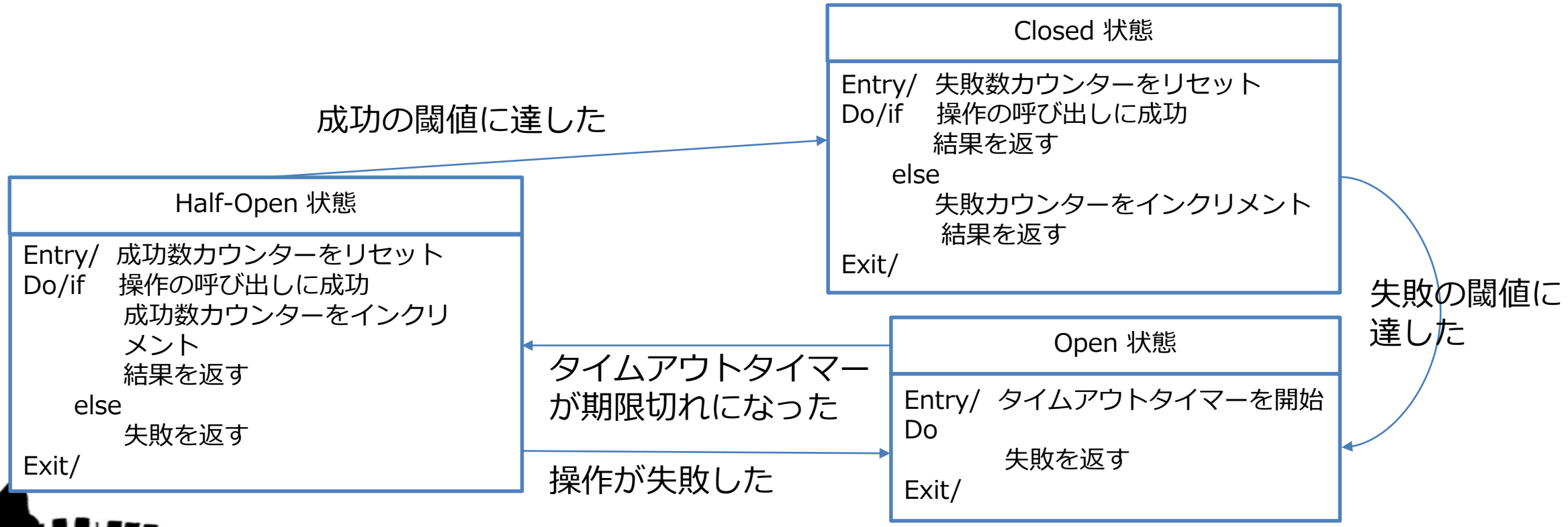
1. Cache-Aside パターン

- データストアからオンデマンドでキャッシュにデータを読み込むことでパフォーマンスを向上させる
- データストアとのデータ整合性維持を実現



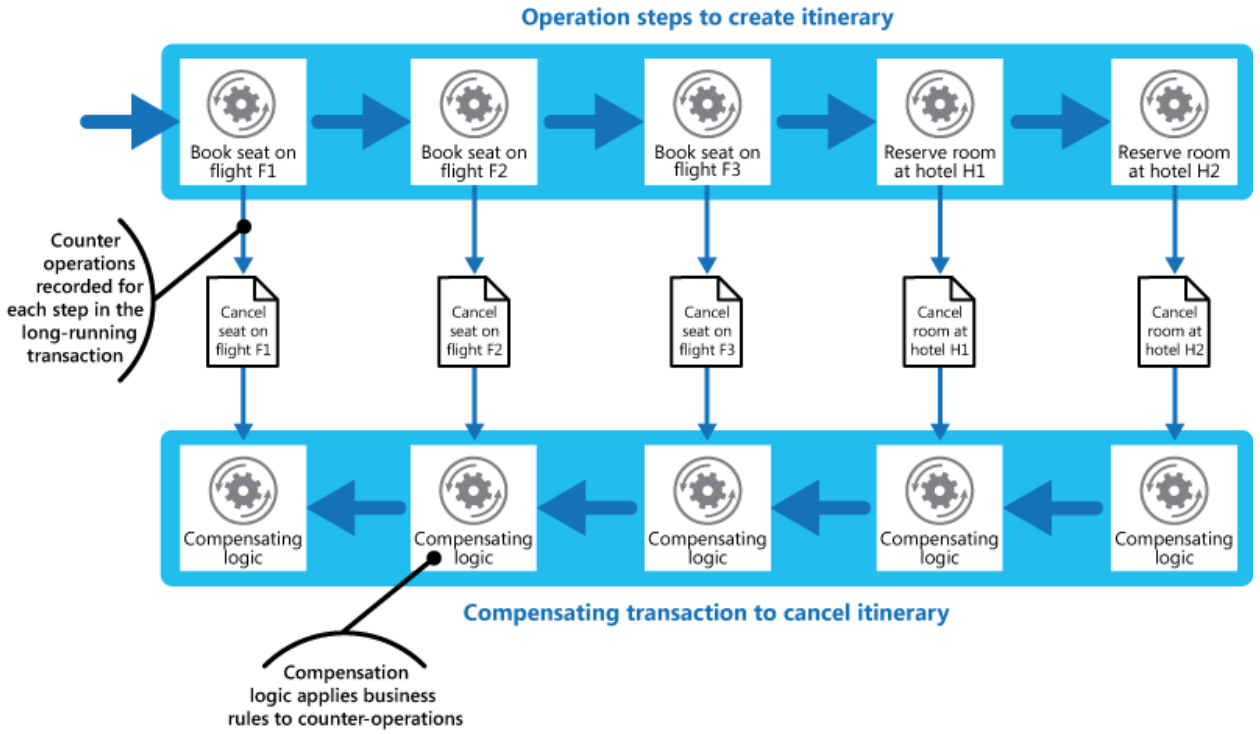
2. Circuit Breaker パターン

- リモート サービスやリソースに接続している際に、解決に可変長の時間がかかるフォールトをハンドルする
- アプリケーションの安定性と復元性を改善



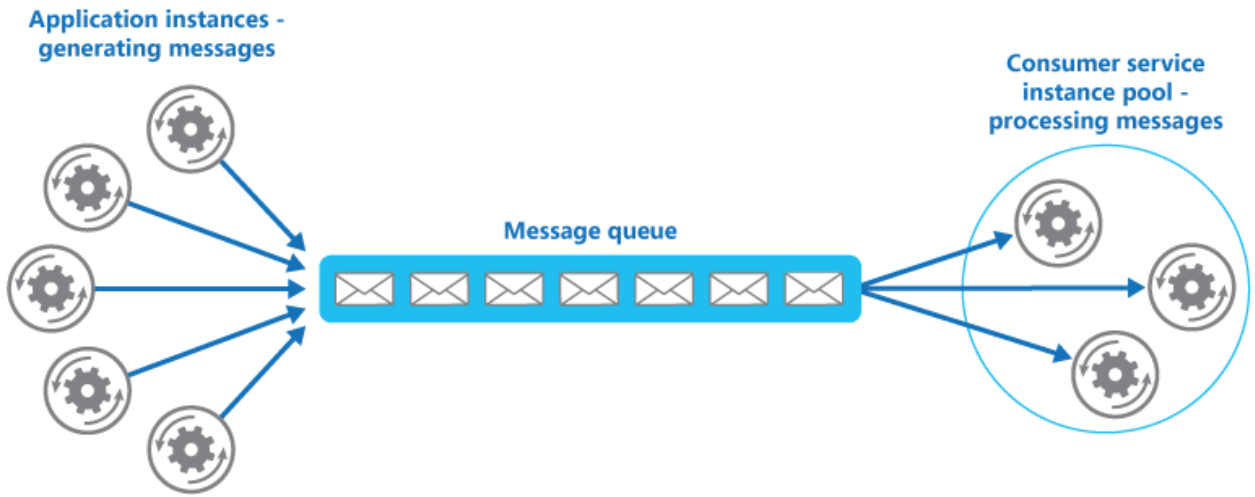
3. Compensating Transaction パターン

- 複数の地域に散在するデータソースに対して「強いトランザクション整合性」でなく「結果整合性」を実装する
- 処理が失敗した場合の補正トランザクションを実装する



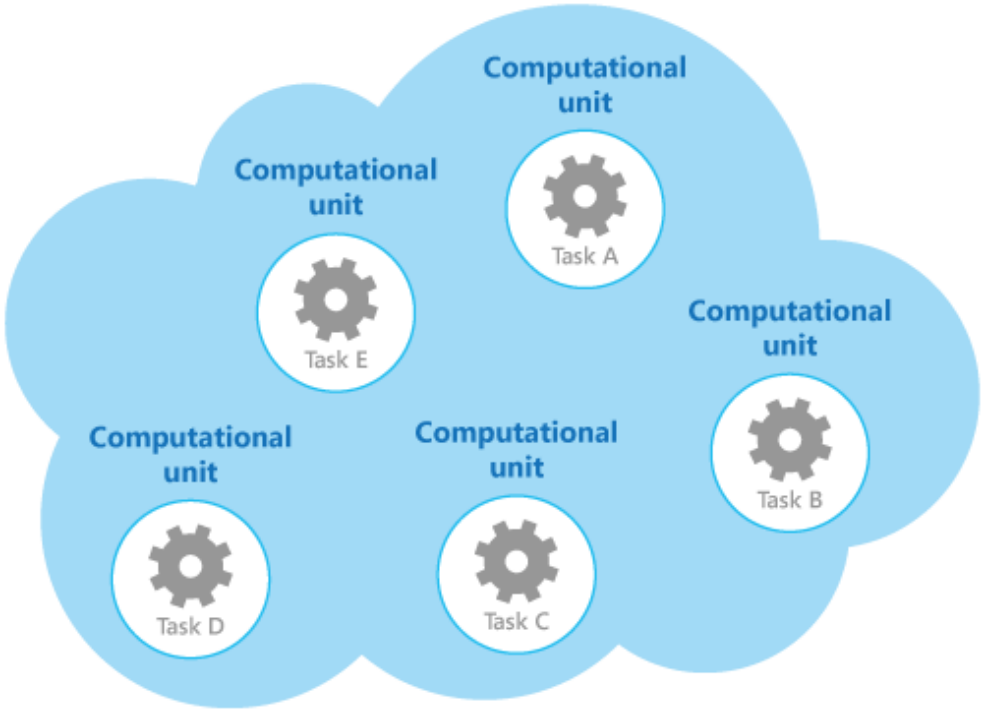
4. Competing Consumers パターン

- ビジネスロジックをリクエストから粗にすることで、スループットの最適化、スケーラビリティと可用性の向上、ワークロードの分散を行う
- ポイズンメッセージ（処理できないリクエスト）に注意



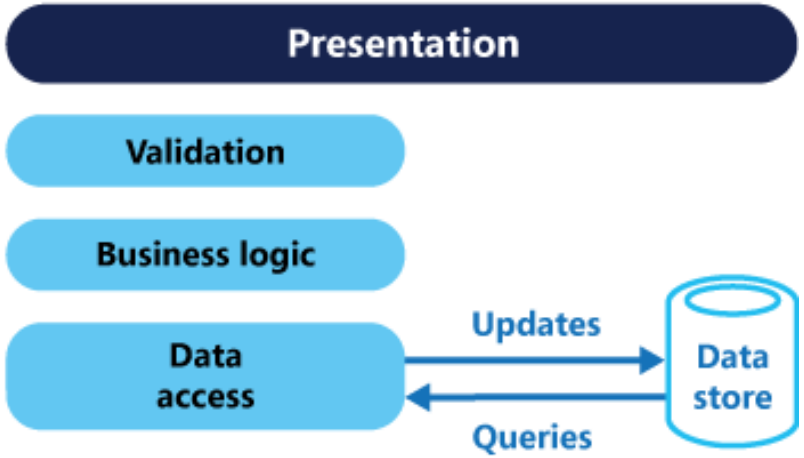
5. Compute Resource Consolidation パターン

- 複数のタスクや操作を一つの計算ユニットに統合するパターン
- 処理を分割しすぎてコスト増加、スループット低下の場合に有効

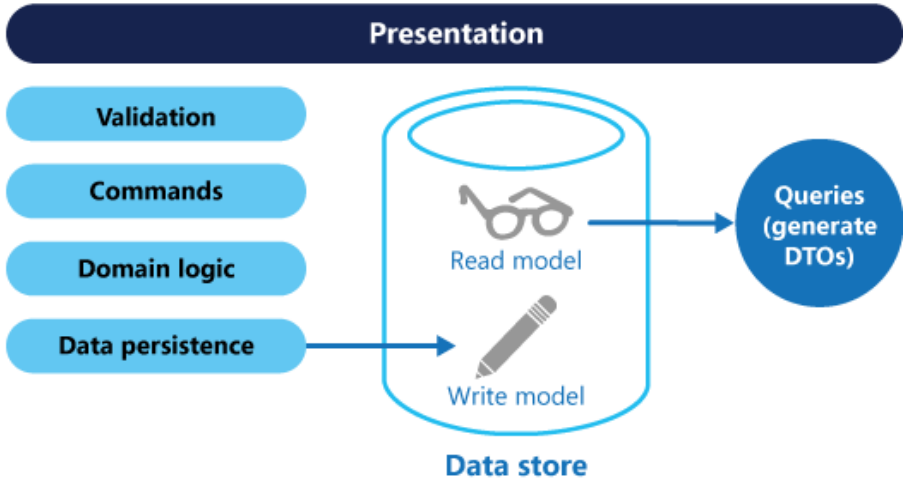


6. CQRS パターン

- 読み取り処理／書き込み処理で異なるモデルを利用することで、パフォーマンス・スケーラビリティを向上する
- Materialized View パターン、Event Sourcing パターン等を組み合わせて利用する



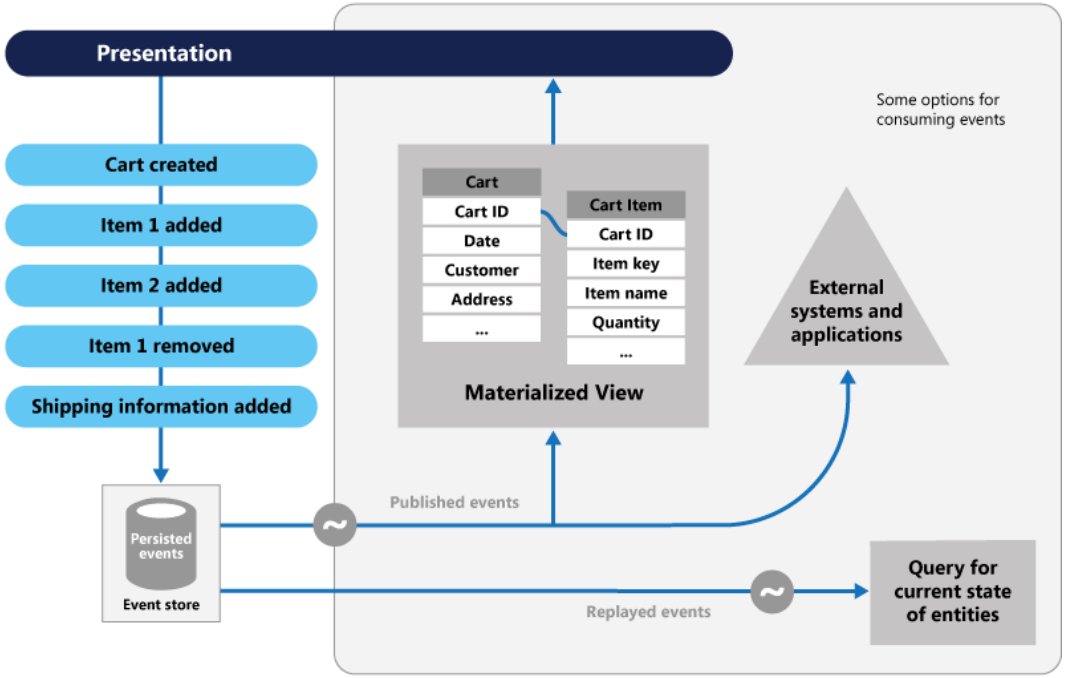
CQRS パターン未適用



CQRS パターン適用

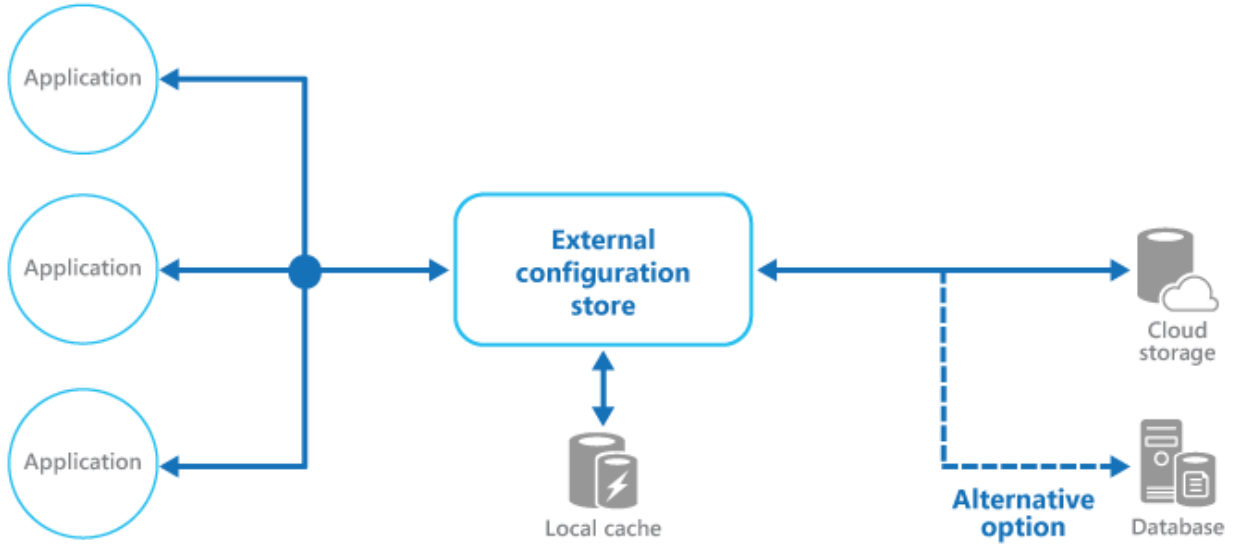
7. Event Sourcing パターン

- データストアとビジネスドメインの同期を避けることで、パフォーマンスとスケーラビリティを向上させる
- 従来の CRUD 処理で発生しやすい更新競合を避けるため、データに対して発生した変更イベントを処理する



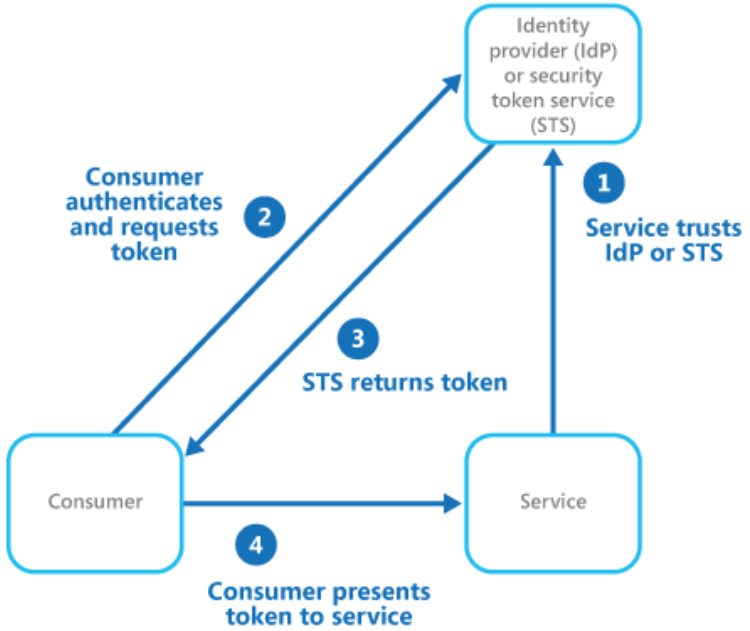
8. External Configuration Store パターン

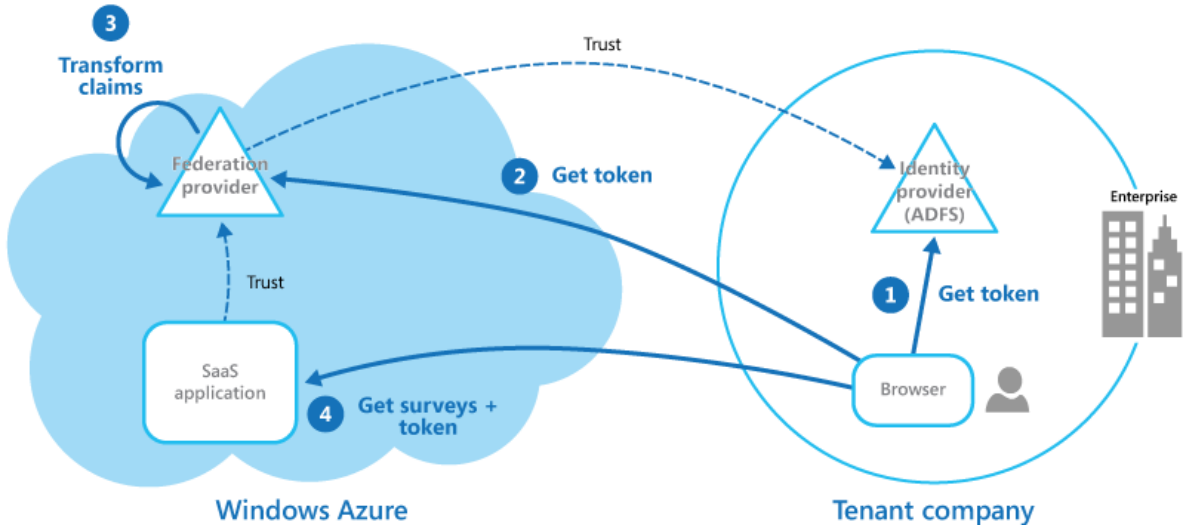
- クラウド上にホストされた複数インスタンスに対して構成の変更を管理するパターン
- Runtime Reconfiguration パターンと組み合わせ、再デプロイや再起動を必要とせずに再構成が可能となる



9. Federated Identity パターン

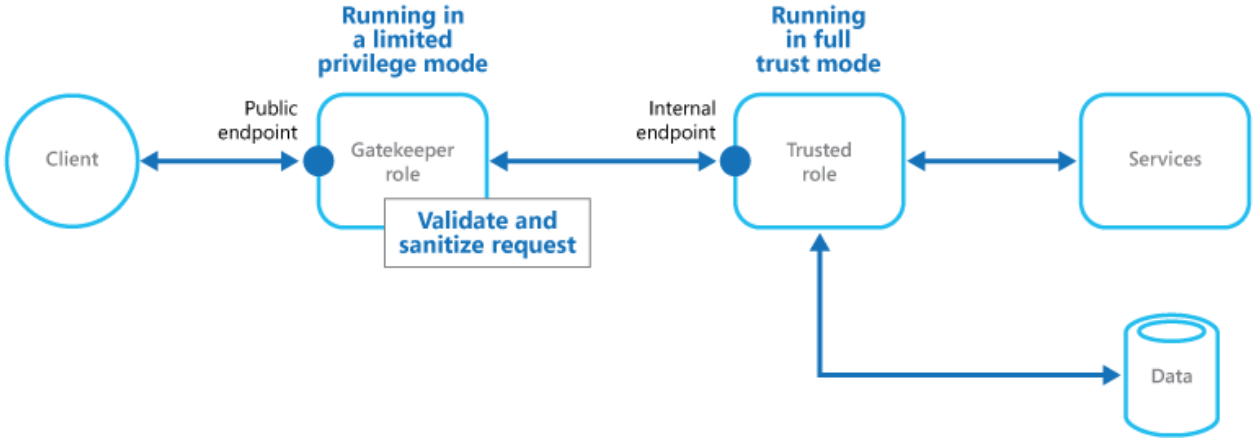
- ユーザ認証を必要とする開発を簡素化し、認証を外部の ID プロバイダに委任するパターン
- 委任先の ID プロバイダの信頼性の検討、単一障害点になる場合に対する検討が必要





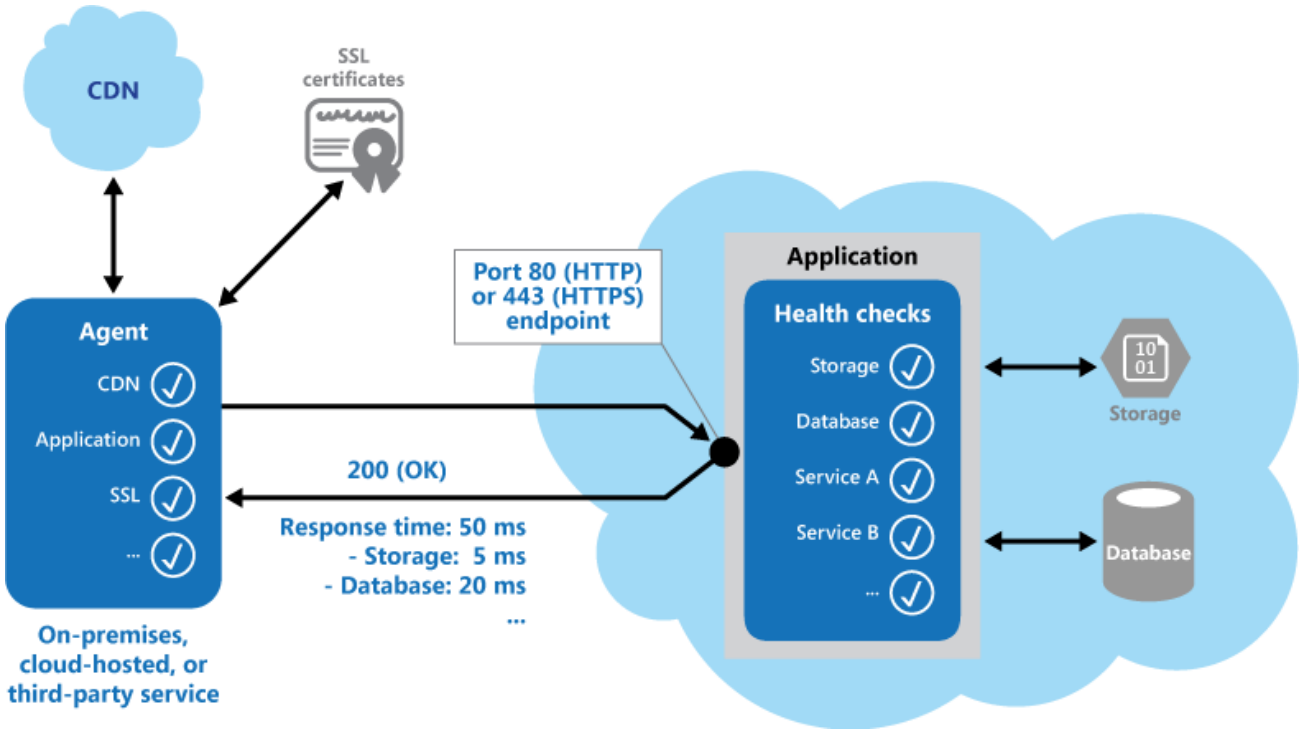
10. Gatekeeper パターン

- データストアにアクセスするビジネスロジックに対して認証／検知を行うセキュリティレイヤーを追加し、システムに対する攻撃対象領域を狭める
- Valet Key パターンと組み合わせ、アクセス権限をキーかトークン利用に限定することでセキュリティを強化する



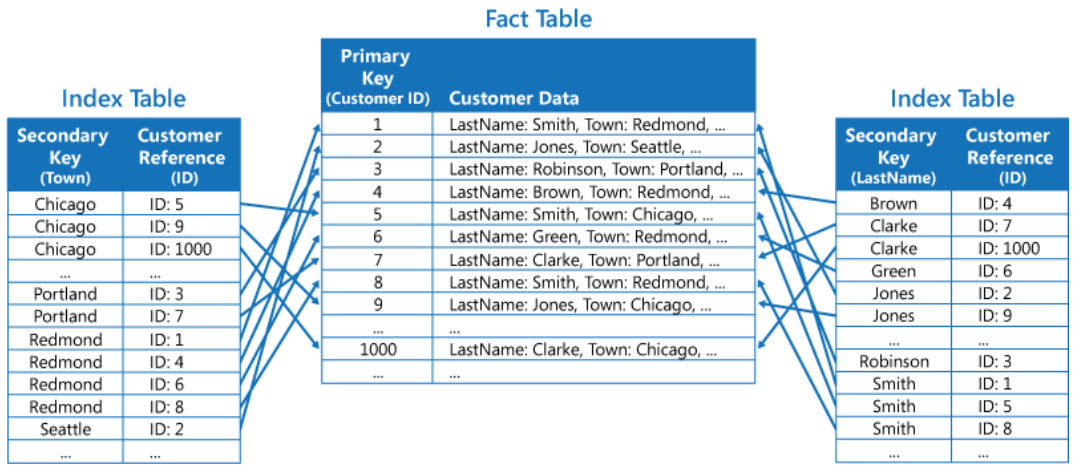
11. Health Endpoint Monitoring パターン

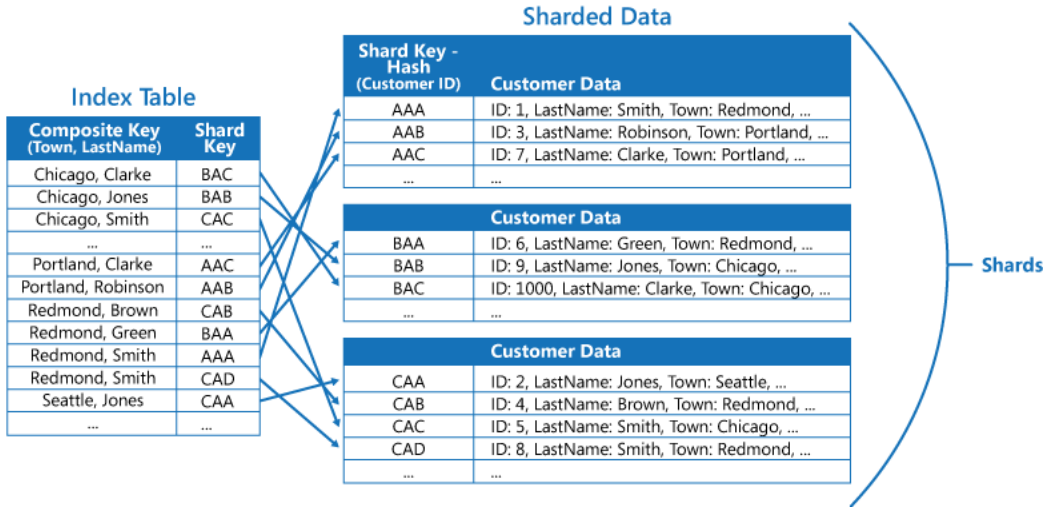
- システムが正しく実行されているかを確認するエンドポイントを作成し、クラウド上での監視を容易にするパターン



12. Index Table パターン

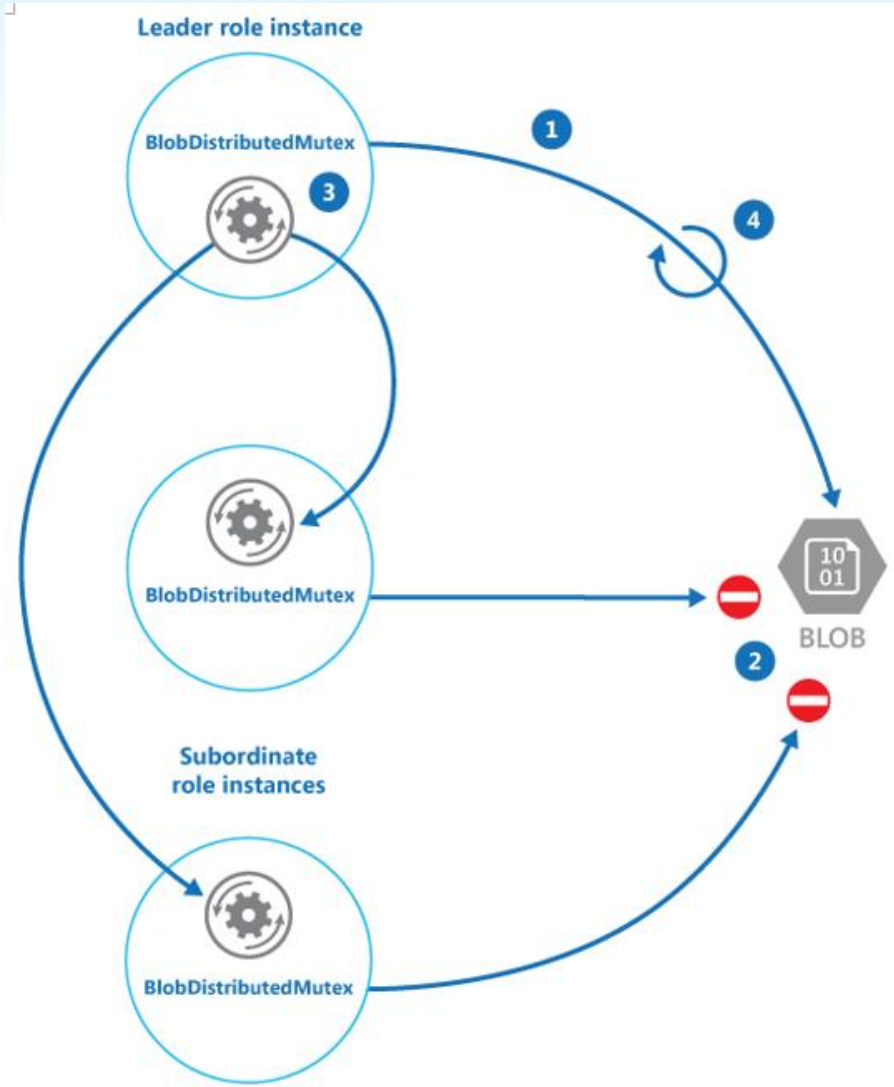
- NoSQL データストアのフィールドにインデックスを作成し、データ取得を高速化するパターン
- リレーショナルデータベースと異なり NoSQL はインデックスを独自に設計する場合が多い





13. Leader Election パターン

- 複数インスタンスの中からリーダーとして稼働するインスタンスを選出し、インスタンス群を強調して動作させるパターン
- 外部リソースの Mutex 処理等を利用し、インスタンス群の中からリーダーを選出する



14. Materialized View パターン

- 複数のデータストアに最適なビューを事前に作成することで、パフォーマンスとスケーラビリティを向上させる
- Event Sourcing パターンと組み合わせることでデータストアとの同期を実現させる

Order table

Partition key	Row key	Order date	Shipping address	Total invoice	Order status
001 (Customer ID)	1 (Order ID)	11082013	One Microsoft way Redmond, WA 98052	$400	In process
005	2	11082013	One Microsoft way Redmond, WA 98052	$200	Shipped

OrderItem table

Partition key	Row key	Product	Unit Price	Amount	Total
1 (Order ID)	001_1 (OrderItem ID)	XX	$100	2	$200
1	001_2	YY	$40	5	$200
2	002_1	ZZ	$200	1	$200

Customer table

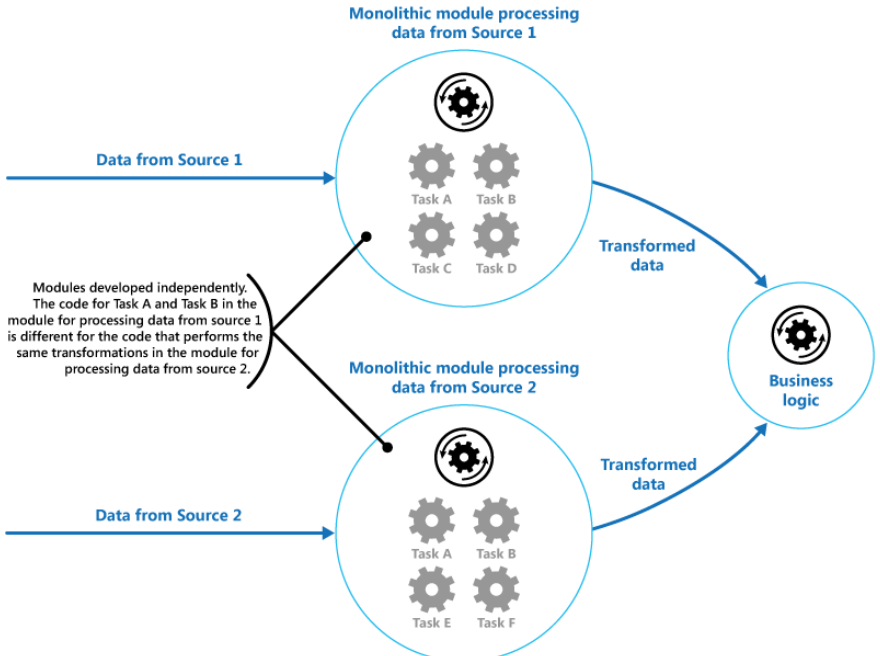
Partition key	Row key	Billing Information	Shipping address	Gender	Age
US East (region)	001 (Customer ID)	*****0001	One Microsoft way Redmond, WA 98052	Female	30
US East	002	*****2006	One Microsoft way Redmond, WA 98052	Male	40

Materialized View

Partition key	Row key	Product Name	Total sold	Number of customers
Electronics (Product category)	001 (Product ID)	XX	$30,000	500
Electronics	002	YY	$100,000	400

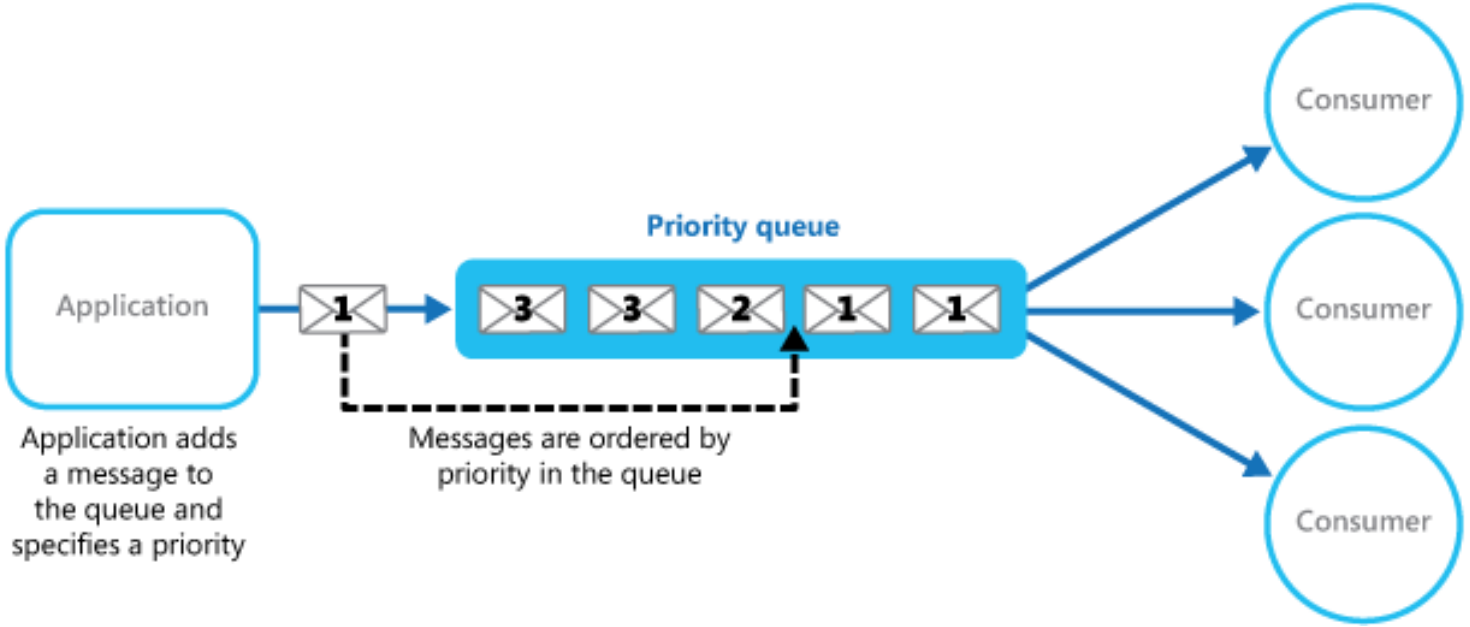
15. Pipes and Filters パターン

- 複雑な処理を再利用可能な個別要素に分割し、パフォーマンスとスケーラビリティを向上させる
- Compensation Transaction パターンと組み合わせ、結果整合性を維持する分散トランザクションを実装する



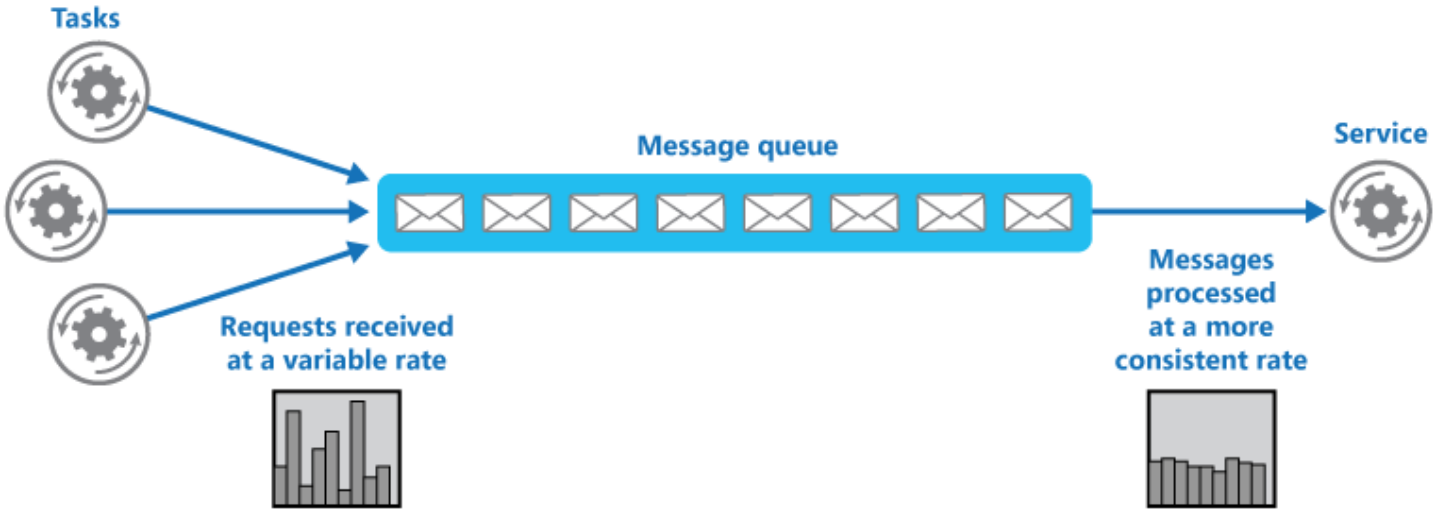
16. Priority Queue パターン

- 異なるユーザや複数のタスクに対し、異なる優先度で処理が必要な場合に利用するパターン
- バックグラウンド処理の割り当てを適正化し、個別のクライアントに異なるサービスレベルを保証する



17. Queue-Based Load Leveling パターン

- キューをバッファーとして利用することで、高負荷時の可用性と応答性の影響を最小限に抑えるパターン
- Throttling パターンと組み合わせ、キューのメッセージを介してルーティングをすることでリソース枯渇を避けることができる

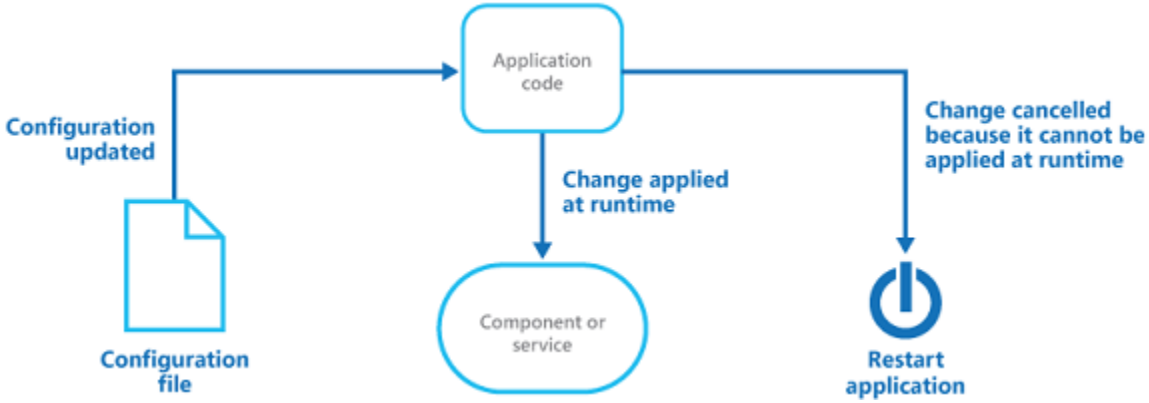


18. Retry パターン

- 障害を一時的なものとみなし、操作を透過的にリトライすることでアプリケーションの安定性を改善するパターン
- 長く続く障害に対しては、Circuit Breaker パターンを適用することで縮退した機能を提供する

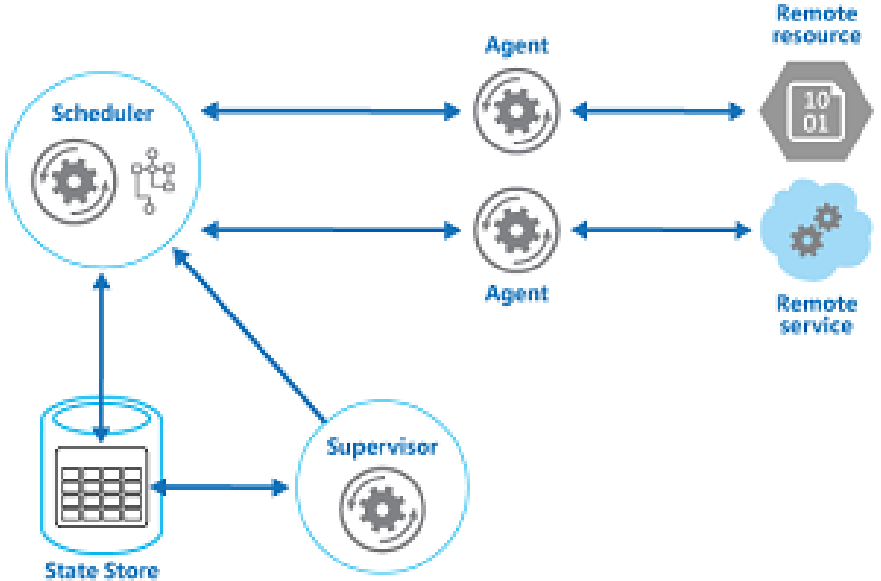
19. Runtime Reconfiguration パターン

- 再デプロイや再起動をすることなく再構成可能なアプリケーション設計を行い、ダウンタイムの最小化を可能にするパターン
- External Configuration と組み合わせ、アプリケーションの構成を管理することが容易になる



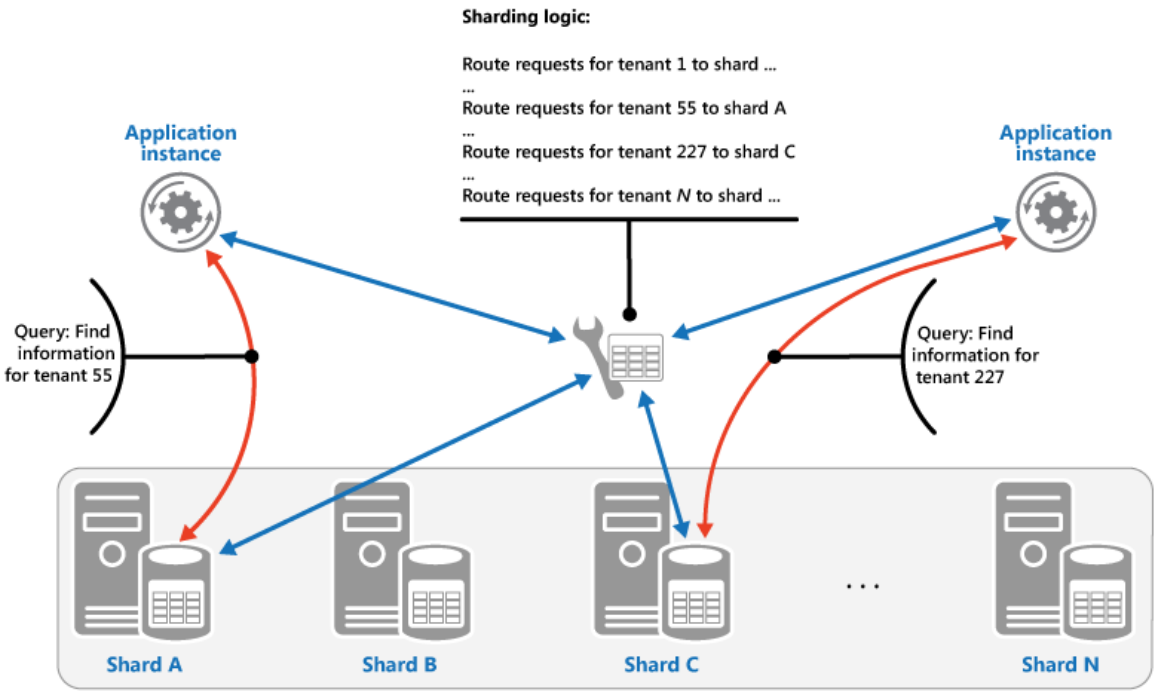
20. Scheduler Agent Supervisor パターン

- サービス全体やリモートリソースにまたがる一連のアクションを調整し、透過的に障害への対処を行うことでシステムのサービスレベルを向上させる
- Retry, Circuit Breaker, Leader Election 等の可用性を向上させるパターンと組み合わせて利用する



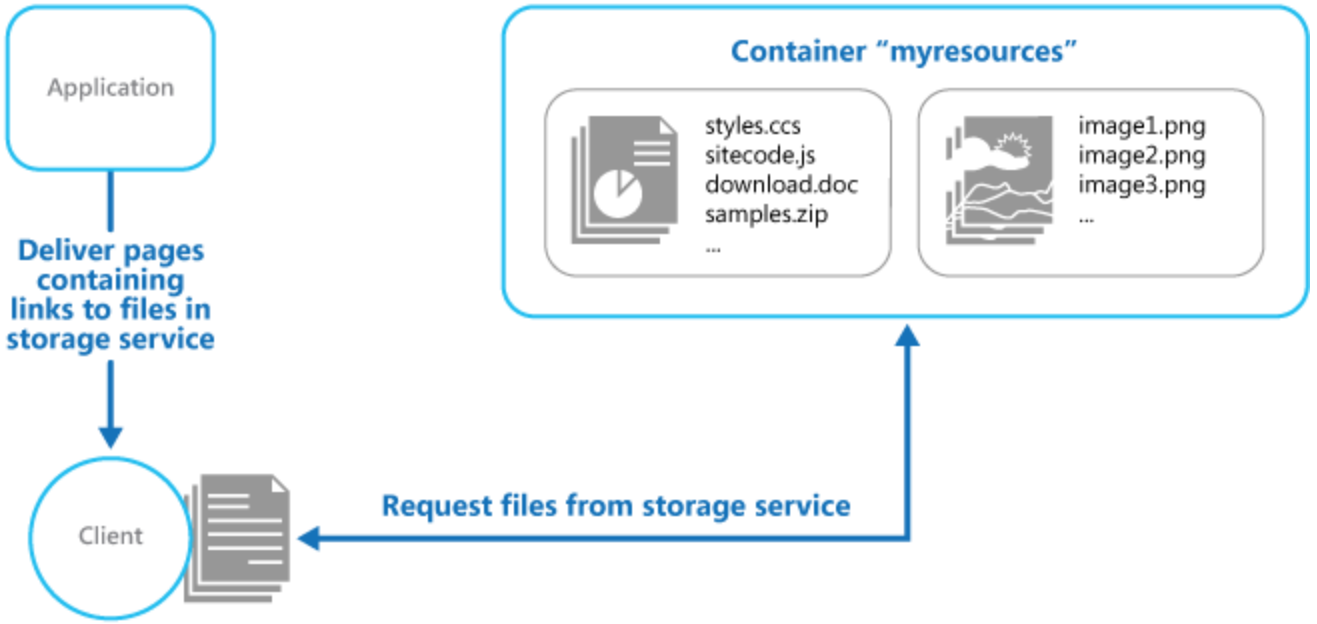
21. Sharding パターン

- データストアを水平分割された一連のシャードに分割することで、大量のデータを格納することでスケーラビリティを向上させるパターン
- Materialized View, Index Table パターンと組み合わせ、分割を最適化する



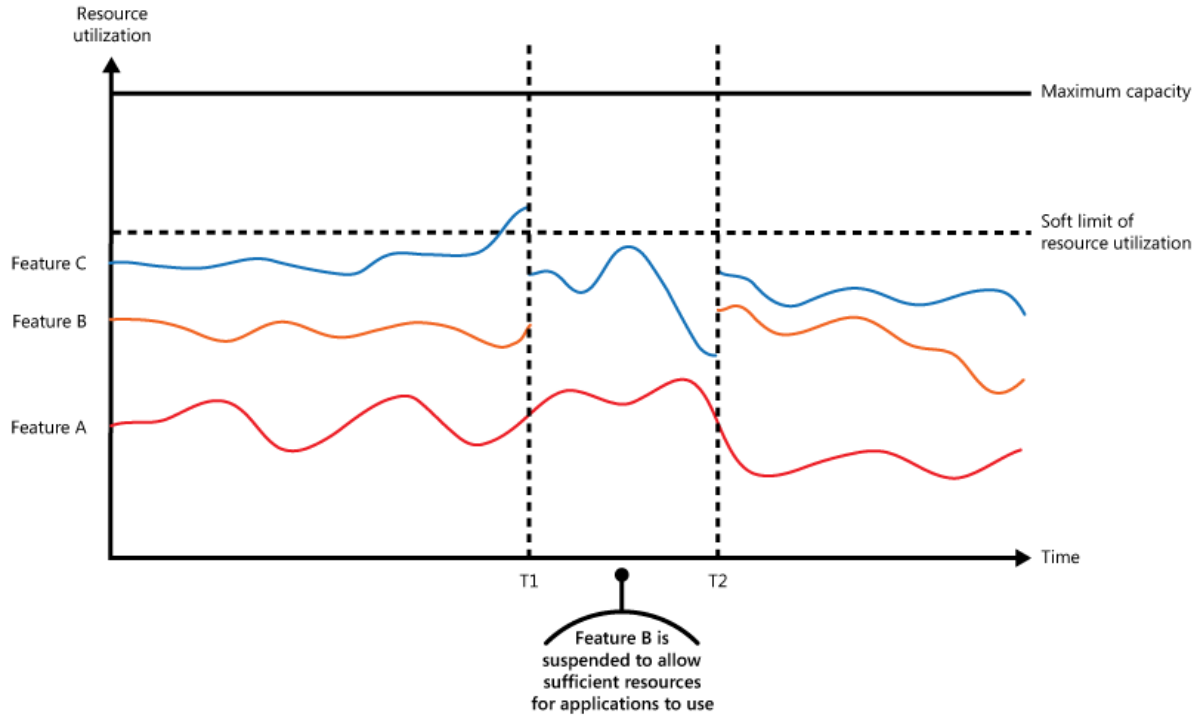
22. Static Content Hosting パターン

- 安価なクラウドストレージに直接コンテンツを配置することで、コンピューティングリソースを最小化するパターン
- コンテンツに対しての処理が必要な場合は適用できない



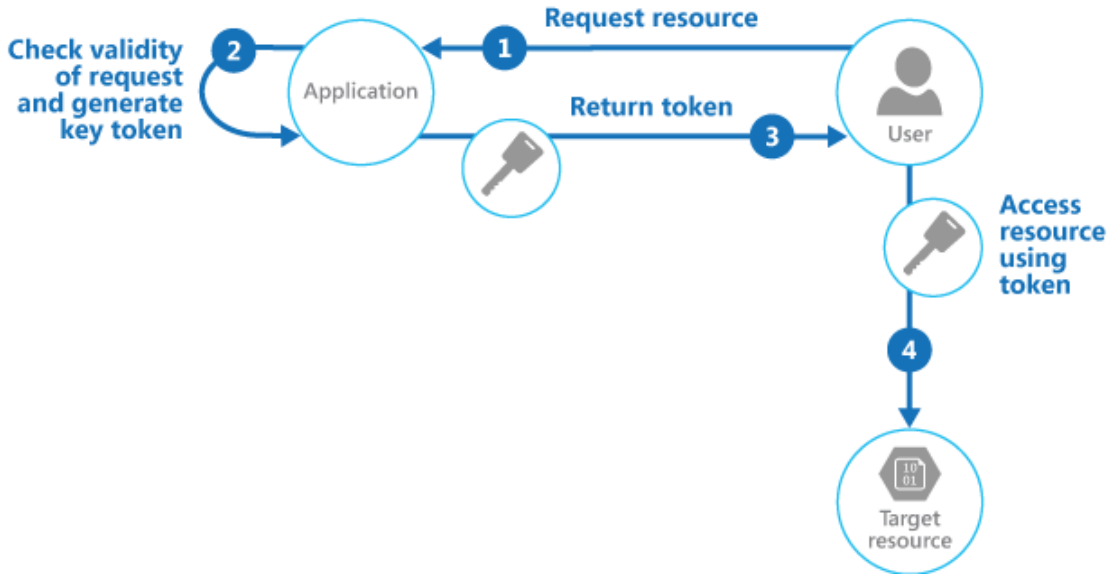
23. Throttling パターン

- サービス全体で使用するリソースを制御し、需要の増加によって高負荷時でもサービスを継続利用可能とする
- キューを利用したパターンと組み合わせることで、必要とするリソースの最適化を行う



24. Valet Key パターン

- トークンやキーを利用することで、サービスへの制限付き直接アクセスを可能とするパターン
- クラウドリソースを直接利用するアプリケーション（クライアントアプリケーション等）向けに適用することで、パフォーマンスを最大化する



- はじめに
- クラウドとオンプレミスの違い
- クラウドデザインパターンについて
- まとめ

まとめ

- クラウドとオンプレミスではアプリケーション設計の観点が異なる
- クラウドでのアプリケーション設計のノウハウであるクラウドデザインパターンを活用することで、クラウドの特性を引き出すことができる
- クラウドデザインパターンは用途に合わせて組み合わせて利用する

参考

- Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications
  http://msdn.microsoft.com/ja-jp/library/dn568099.aspx

- AWSクラウドデザインパターン
  http://aws.clouddesignpattern.org/index.php/