AWS

SUMMIT

Architecting for the Cloud

クラウドにおけるアーキテクチャの設計原則

Daichi Egawa, AWS Solutions Architect

June 1, 2017

amazon web services

Agenda

- はじめに
- クラウドコンピューティングの特徴
- クラウドの特徴を活かすための設計原則
- まとめ

Agenda

- はじめに
- クラウドコンピューティングの特徴
- クラウドの特徴を活かすための設計原則
- まとめ

本日お話しすること

- 想定する参加者
  - AWS の概要/メリットを理解している
  - AWS を触ったことがある
  - AWS でのシステム設計やサービス選定について悩んでいる

- お話しする内容
  - クラウドらしいアーキテクチャを構築するための原則
  - そういった原則にのっとることで嬉しくなるポイント

AWS 上でシステム設計にあたっての疑問

- 例えば…

AWSの特性を活かすための設計とは？

オンプレミスとは異なる発想で臨むべき？

EC2 上でデータベースを構築するのと、RDS を使うことの違いは？

スケールアウト構成を構築する場合に、何を意識すればよいの？

Agenda

- はじめに
- クラウドコンピューティングの特徴
- クラウドの特徴を活かすための設計原則
- まとめ

AWS / クラウドコンピューティングとは

初期費用ゼロ
低価格

継続的な値下げ

サイジングから
の解放

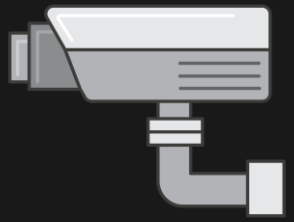
商機を逃さない
俊敏性

最先端の技術や
サービス

いつでも即時
グローバル展開

クラウドコンピューティングならではの特性

- IT 資産がプログラマブルに
- グローバル展開、可用性、無制限のキャパシティ
- ハイレベルなマネージドサービス
- Built-in Security



Agenda

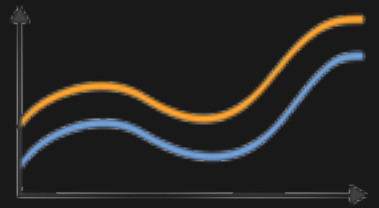
- はじめに
- クラウドコンピューティングの特徴
- クラウドの特徴を活かすための設計原則
- まとめ

Ten Design Principles

- スケーラビリティ
- 常設のサーバではなく使い捨て可能なリソース
- 自動化
- 疎結合
- サーバではなく、サービスの利用(マネージドサービスの活用)
- データベースの使い分け
- 単一障害点の排除
- コストの最適化
- キャッシュの利用
- セキュリティ

スケーラビリティ

- Scalability -



スケーラビリティ

- 運用中のシステムを拡張/縮小させる能力

- スケーラビリティを生かすことで可能になること
  - 柔軟性の向上(事業の必要性に応じたリソース確保)
  - コスト削減(従量課金によるメリット)

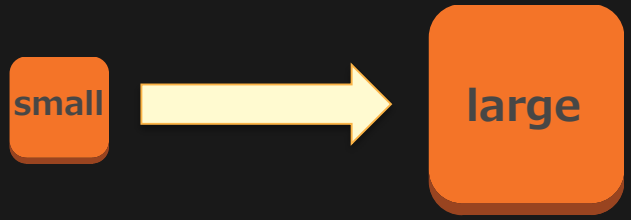
- スケールの種類
  - 水平スケーリング(スケールアウト/スケールイン)
  - 垂直スケーリング(スケールアップ/スケールダウン)

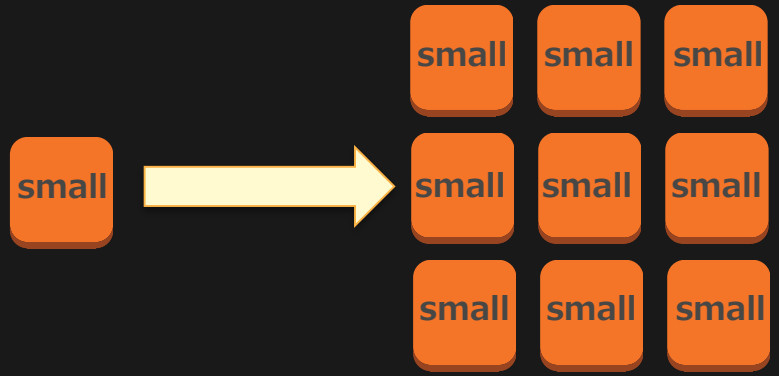
Scalability

水平・垂直のスケーリングの比較

- 垂直スケーリング
  (スケールアップ/スケールダウン)
  - 個々のリソースのスペックを増減
  - リソースの限界が存在
  - インスタンスの停止が伴う

- 水平スケーリング
  (スケールアウト/スケールイン)
  - リソースの台数を増減
  - 論理的には限界が存在しない
  - インスタンスの停止が伴わない





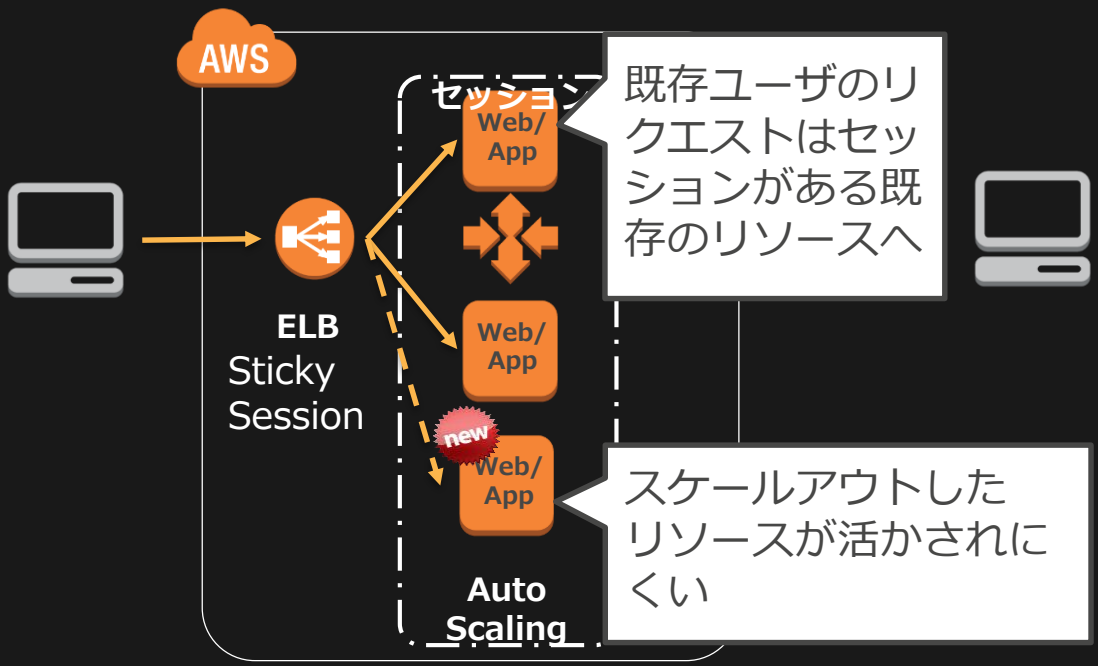
水平スケーリングを行うために心がけること

- ステートレスアプリケーション/コンポーネント
  - ステートフルになる要素を水平スケールするリソースの外部に配置
    - セッション情報は、DynamoDB/ElastiCache へ
    - バイナリファイル, ログなどは、 Amazon S3 へ

- ステートフルになるコンポーネントをどう扱うか
  - 水平スケールするマネージドサービスの利用
  - 水平スケールができない場合に注意すべき制約を洗い出す

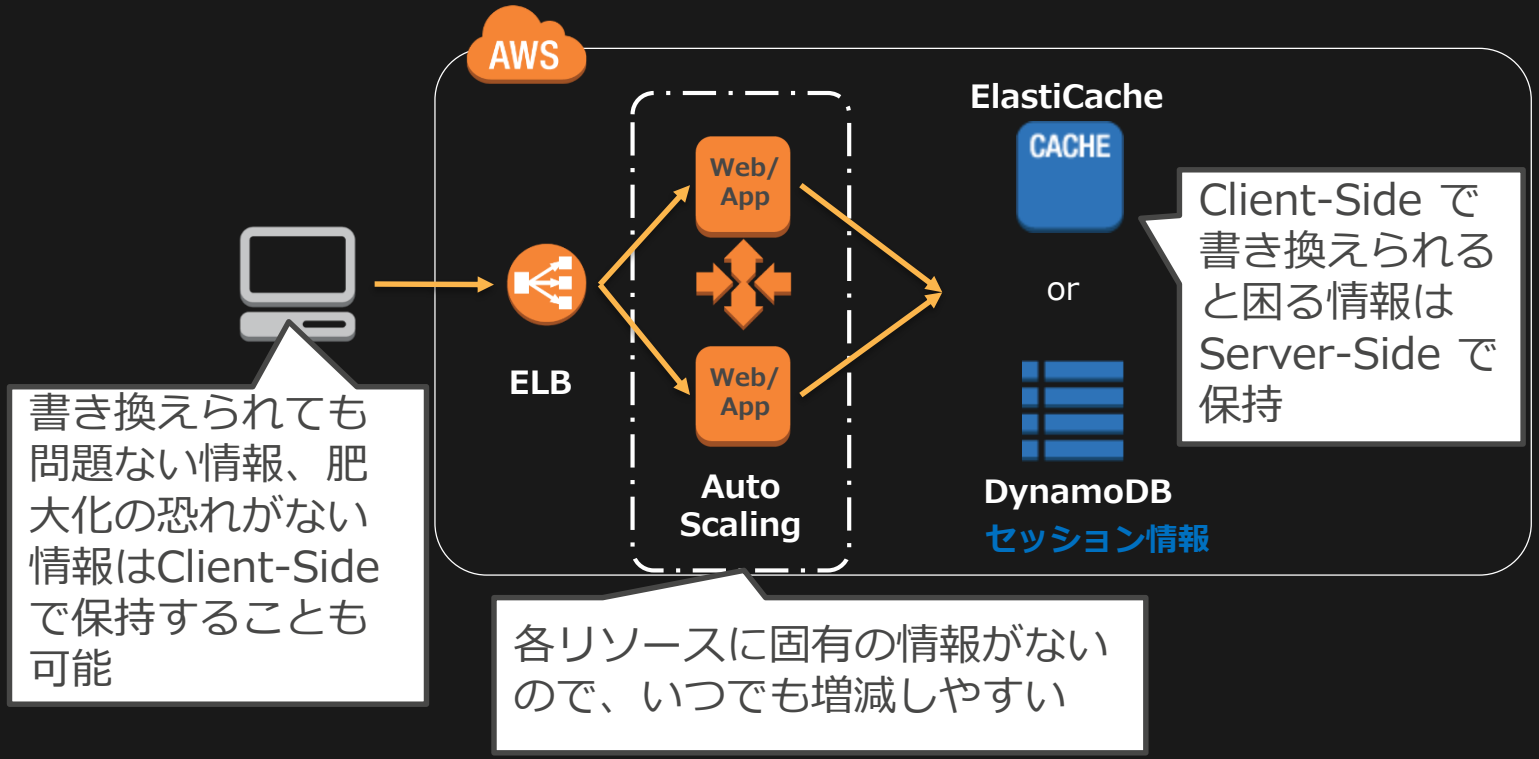
ステートレスにするためのセッション情報の扱い

- セッション情報を水平スケールさせたいコンポーネントで持つと
  - ➢スケールアウト時
    - スケールアウトしたリソースが使われにくい
  - ➢スケールイン時
    - セッションも一緒に落とすことになるので、困難



ステートレスにするためのセッション情報の扱い

- スケールさせやすくするためにセッション情報は外だし



常設のサーバではなく使い捨て可能なリソース

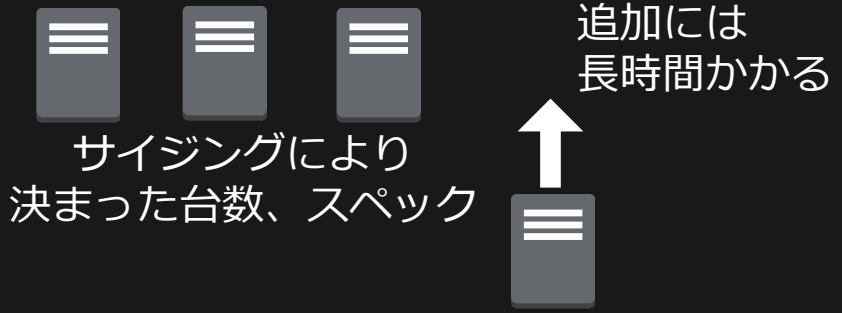
- Disposable Resources Instead of Fixed Servers -

リソースに対する考え方

今まで(オンプレミス)

固定のリソースで運用

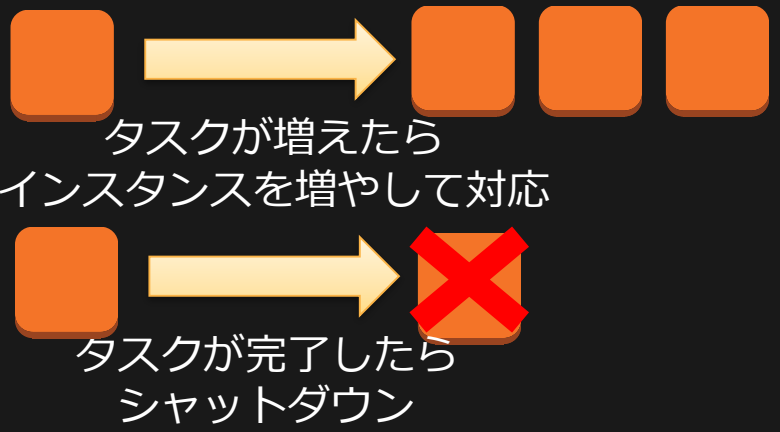
- リソース購入は前払い
- リソース追加には長いリードタイム
  → 事前に厳密なサイジングが必須



AWS

いつでも増減・変更可能(使い捨て可能)

- 初期費用不要
- 即時に追加可能
  → 厳密なサイジングは不要



Disposable Resources Instead of Fixed Servers

マインドセットを変える

今まで(オンプレミス)

- 固定リソース運用により
  助長されがちなプラクティス
  - 手作業による設定変更
  - IP アドレスのハードコード
  - シーケンシャルな処理
  - 常に電源 ON

AWS

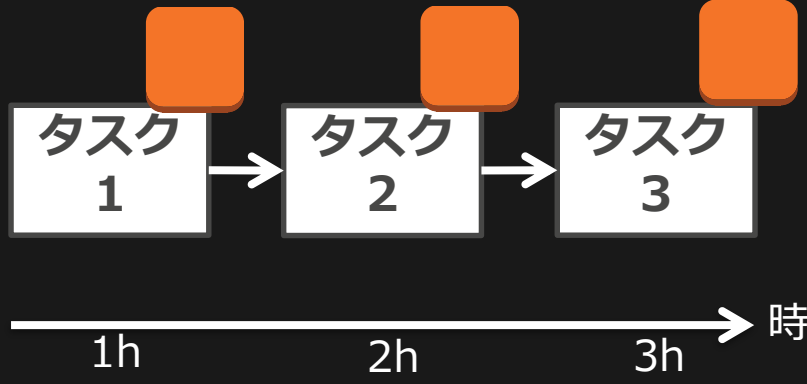
リソースが常に存在する/変更しない
前提の設計、運用を止める

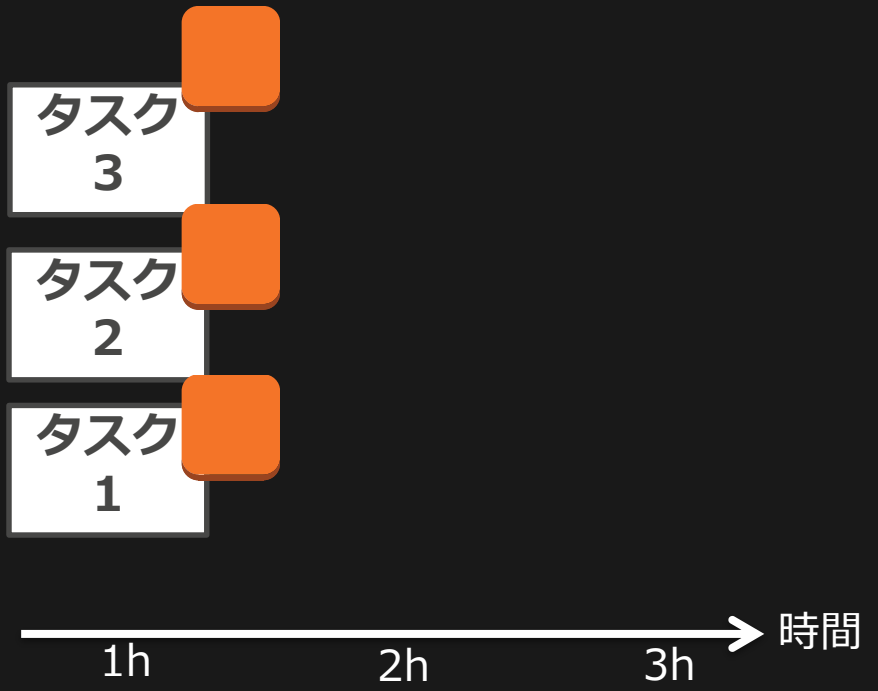
- 自動化
- DNS によるアクセス
- バッチスケジュールの見直し
- 利用時間帯の確認

Disposable Resources Instead of Fixed Servers

バッチスケジュールの見直し

- 並列化で早く終わらせる
  - 1台で n 時間も
    n台で 1時間も料金は同じ





使い捨て可能にするためのポイント

- AMI, スナップショット戦略
  (いつでもリソースを作成/復元できるように準備)
  - 全部入り AMI
  - Bootstrapping
  - Hybrid(AMI と Bootstrapping の組み合わせ)

- Infrastructure as Code
  - 個々のリソースだけでなく、システム全体をいつでも作成できるよう準備
    (例)バッチシステムを CloudFormation で作成し、完了したら削除

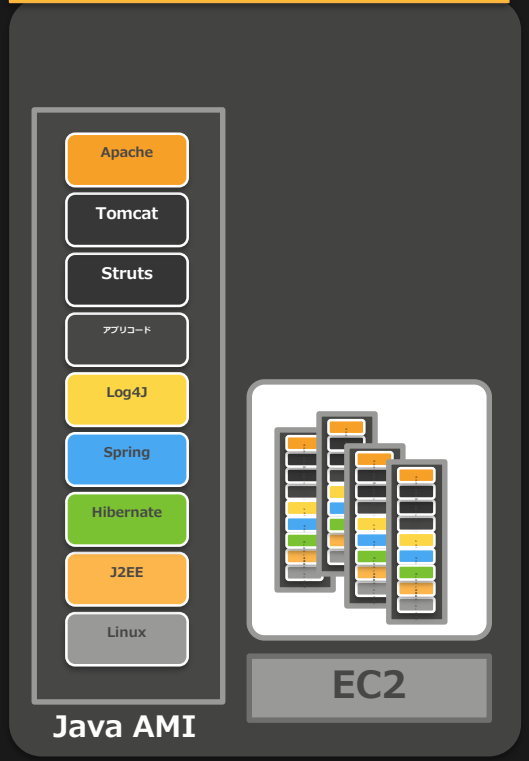
Disposable Resources Instead of Fixed Servers

AMI 戦略

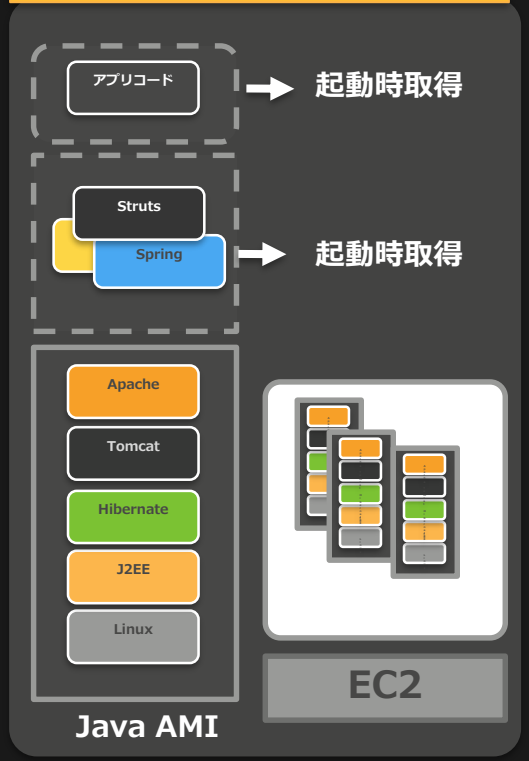
Apache
Tomcat
Struts
アプリコード
Log4J
Spring
Hibernate
J2EE
Linux

Javaスタック

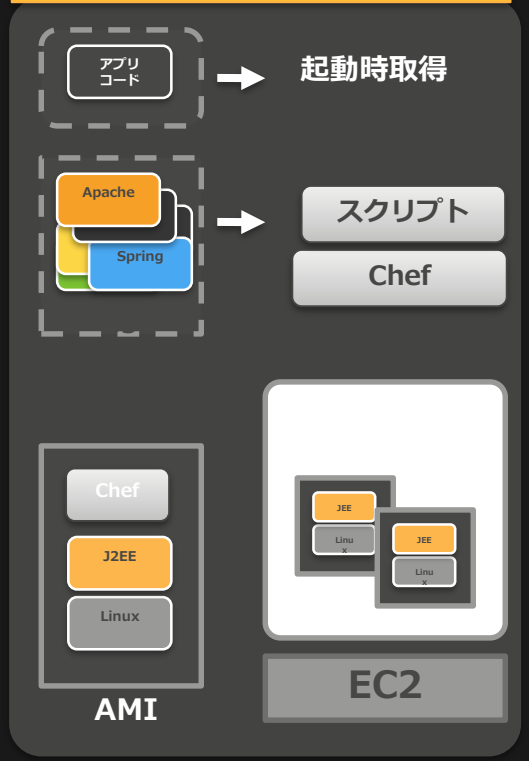
[1]全部入りAMI



[2]部分設定済 AMI



[3]最小構成AMI



Disposable Resources Instead of Fixed Servers

AMI 戦略 Pros/Cons

方式	利点	欠点
[1]全部入りAMI	同一性の確保 起動時間が早い	デプロイ毎にAMIを作り直す必要がある(作成プロセスを自動化すれば欠点とはならない)。
[2]部分設定済AMI	あとはアプリコードのみ最新を取得すれば良く扱いやすい	
[3]最小構成AMI	柔軟に中身を変更できる	起動処理に時間がかかる。 場合によっては外部ライブラリが取得できないことも

自動化

- Automation -

自動化

- 自動化を意識することで得られるメリット
  - オペミス防止などの安定した運用
  - 可用性の向上(自動復旧)
  - 運用負荷の軽減

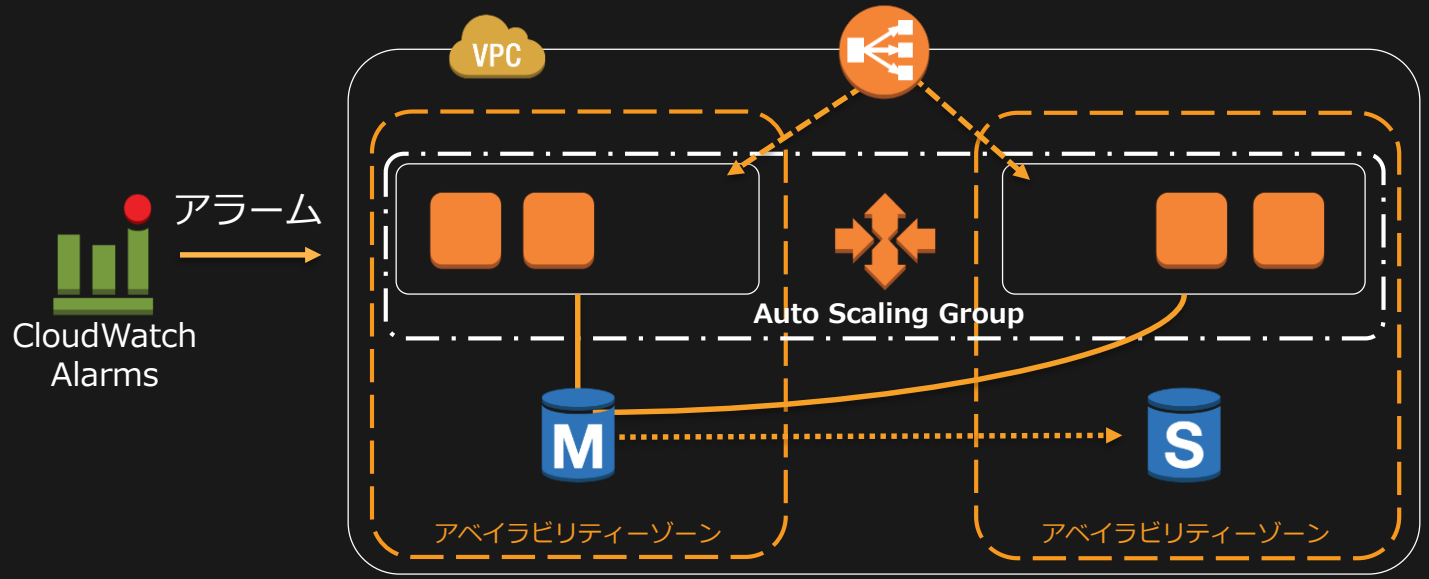
- AWS と自動化
  - API で動作するので、自動化しやすい
  - 自動化をサポートするサービスが豊富

自動化をサポートする代表的なサービス/機能

- AWS Elastic Beanstalk
- AWS OpsWorks
- Auto Scaling
- Amazon EC2 Auto Recovery
- Amazon EC2 Systems Manager
- Amazon CloudWatch Alarms
- Amazon CloudWatch Events
- AWS Lambda Scheduled events

水平スケーリングの自動化

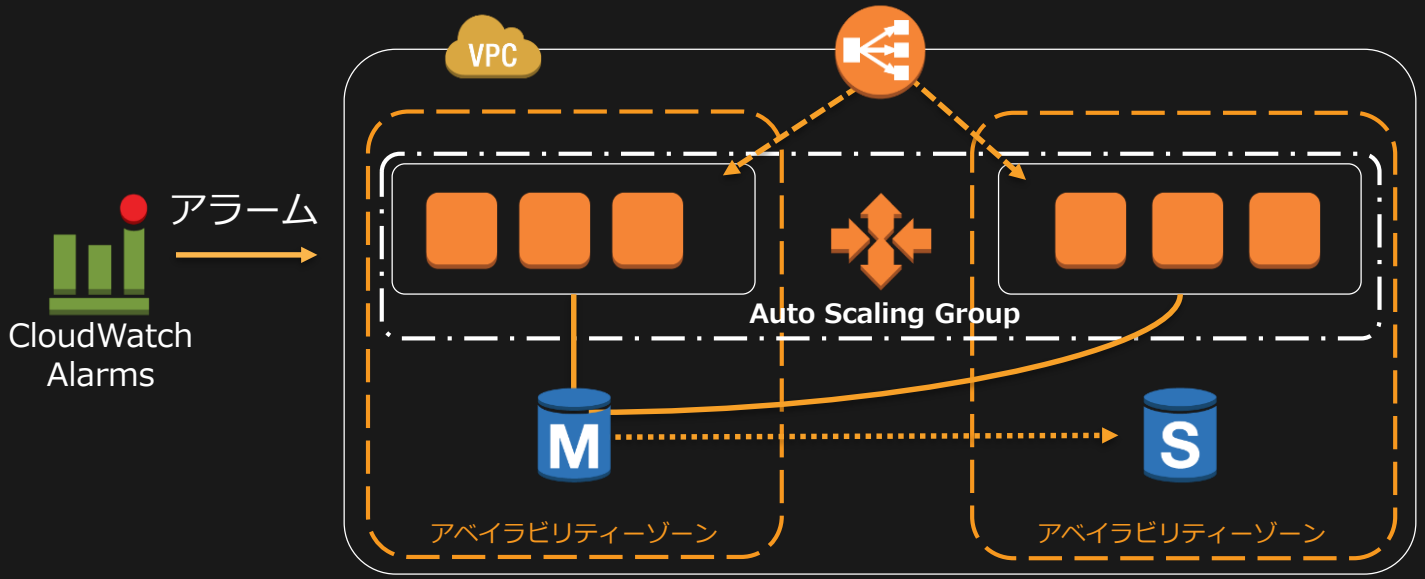
- システムの状況を監視し、自動的にスケールアウト/イン



Automation

水平スケーリングの自動化

- システムの状況を監視し、自動的にスケールアウト/イン
- スケールアウトしたインスタンスは、ユーザーデータのカスタムスクリプトなどで自動設定



疎結合

- Loose Coupling -

疎結合(Loose Coupling)

- 結合度(Coupling)は低いほど好ましい
  - 不具合の影響範囲を最小限にとどめられる
  - スケーリングしやすい

- 疎結合にするために心がけること
  - Well-Defined Interfaces
  - Service Discovery
  - Asynchronous Integration
  - Graceful Failure

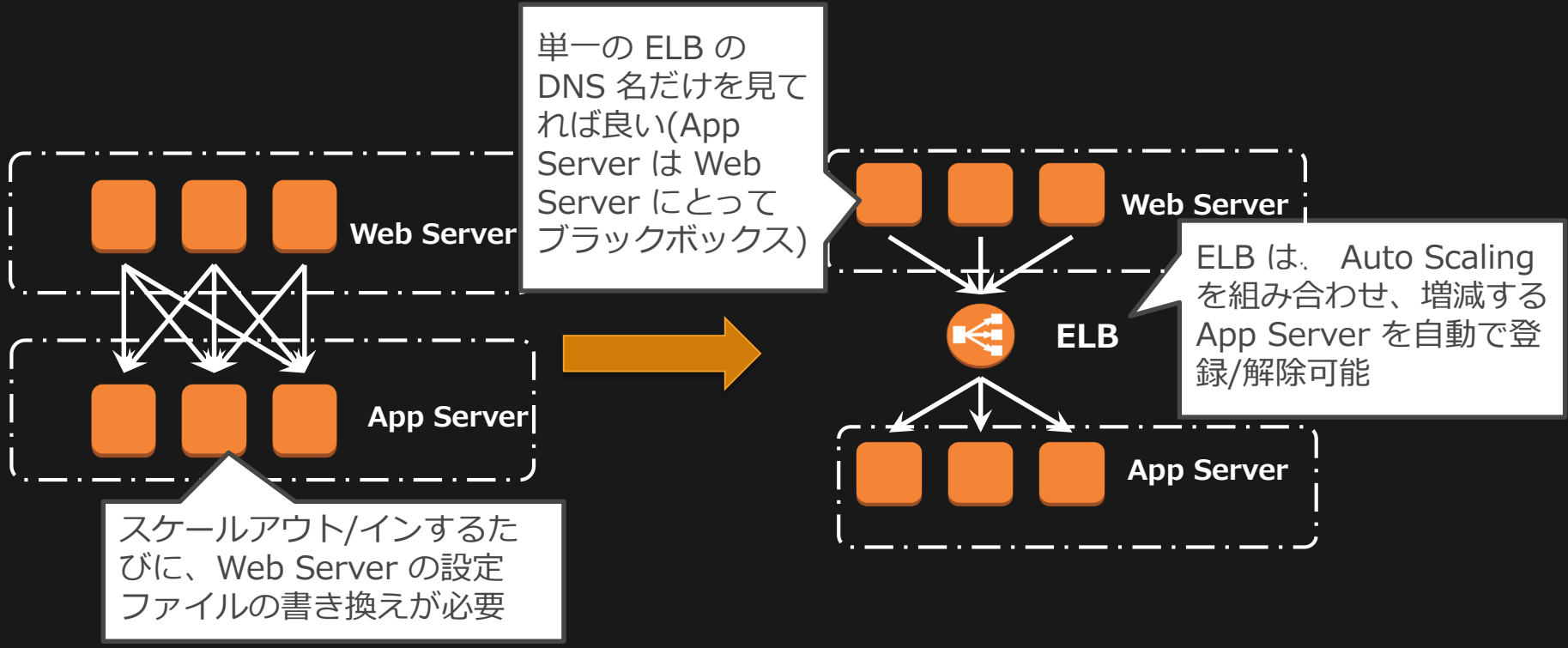
Well-Defined Interfaces

- アクセス元から見て、アクセス先をブラックボックスに
  - 必要な情報はインターフェースで全て提供
    - 内部情報に依存しない
    - 必要な情報だけをわたし、API でアクセス
    - IP アドレスではなく DNS 名でアクセス
    - 処理を実施するインスタンスへの直接アクセスを避け、ELB, SQS へアクセスさせるよう設計

Service Discovery

- 透過的にアクセスするために、増えたリソース、減ったリソースを検知することが必要
- Service Discovery の例
  - Auto Scaling を使ったインスタンスの ELB への自動登録
  - EC2 ユーザーデータ × Amazon Route 53
  - Auto Scaling Life Cycle Hook × Amazon Route 53
  - 3rd party solutions
    - Netflix Eureka
    - Airbnb Synapse
    - HashiCorp Consul
    - etc

AWS のサービスを活用した疎結合の例



ウェブサーバーは
アプリサーバーと密結合

ELB を挟んで疎結合に

Loose Coupling

Asynchronous Integration

- 同期処理である必要がなければ非同期でつながる
- 非同期処理の利点
  - ユーザ側の利点
    - アプリケーションがブロックされない
  - サーバ側の利点
    - スケーラブルかつ高可用なバックエンド
    - Frontend を停止させることなく Backend を容易にメンテナンス可能
    - 少ないFrontend のキャパシティで多くのリクエストを受付可能
    - リクエストの処理順序やリトライ等の制御が容易に

Loose Coupling

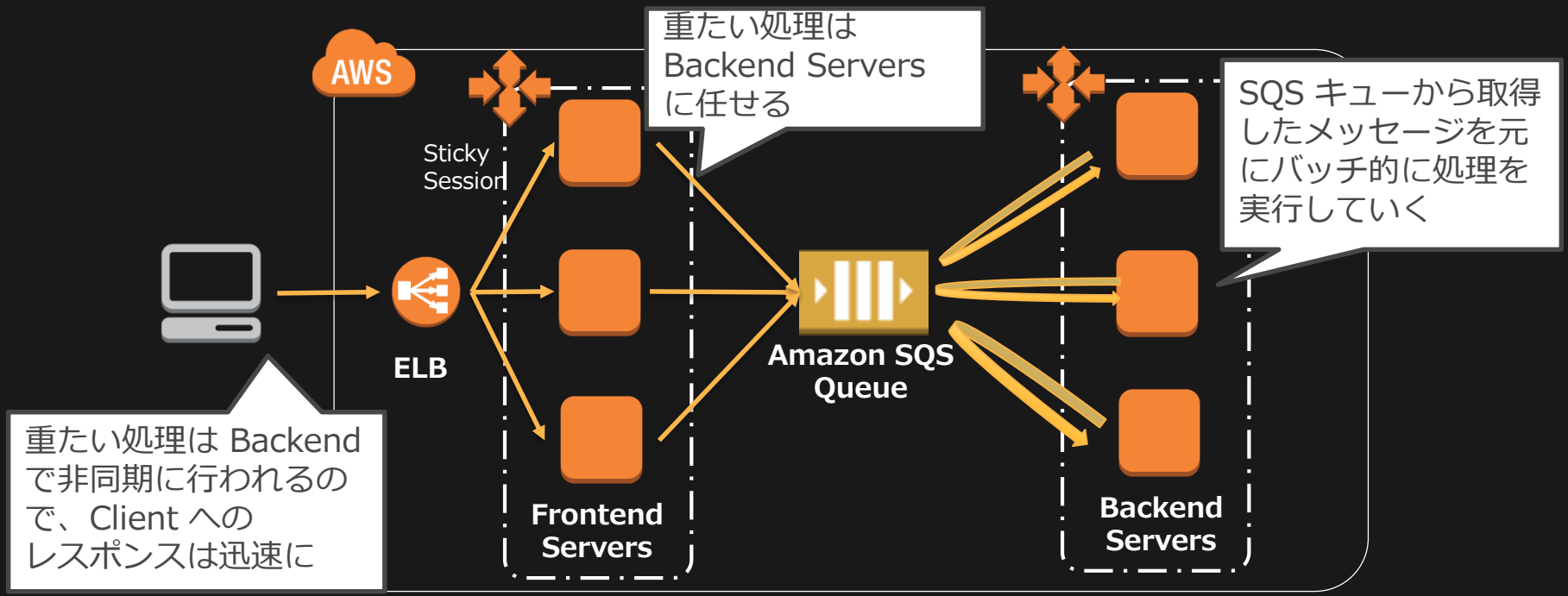
AWS のサービスを活用した非同期処理の例

- Amazon SQS
  - メッセージキューを挟んで、Frontend と Backend を疎結合に
- Amazon Kinesis
  - ストリームデータの生成・受け取りとそのデータ処理部を疎結合に
- AWS Lambda
  - DynamoDB へのアップデートなどのイベントと後続の処理を疎結合に
- Amazon Simple Workflow
  - 複雑なワークフローにのっとる複数のタスク、ワークフローの状態管理を疎結合に
- AWS Step Functions
  - 連続して実行したい処理を疎結合に

Loose Coupling

AWS のサービスを活用した非同期処理の例

- Amazon SQS を挟んで Frontend と Backend を疎結合に



Loose Coupling

Graceful Failure

- 安全に落とすための戦略
  - リソースの停止、削除、障害時の クライアントや他のリソースへの影響を最小限にするように設定しておく

- Graceful Failure の例
  - ELB Connection Draining
    - 新規割り振りは中止して、処理中のリクエストは終わるまで一定期間待つ
  - Route 53 による DNS Failover
    - 障害発生時は、CloudFront + S3 で構築した Sorry Site へ誘導
  - 再試行処理の実装
    - 失敗した場合にはリトライで対応

サーバではなく、サービスの利用

- Services, Not Servers -

サーバではなく、サービスの利用

- マネージドサービスの利用によって得られるメリット
  - コア業務、アプリケーション部分への集中
  - 高可用性
  - 運用負荷軽減
  - 自動スケール(サイジング不要)

- マネージドサービスの活用により Serverless での構築も可能
  - AWS上でのサーバーレスアーキテクチャ入門
    - http://www.slideshare.net/AmazonWebServicesJapan/aws-black-belt-online-seminar-2016-aws

Services, Not Servers

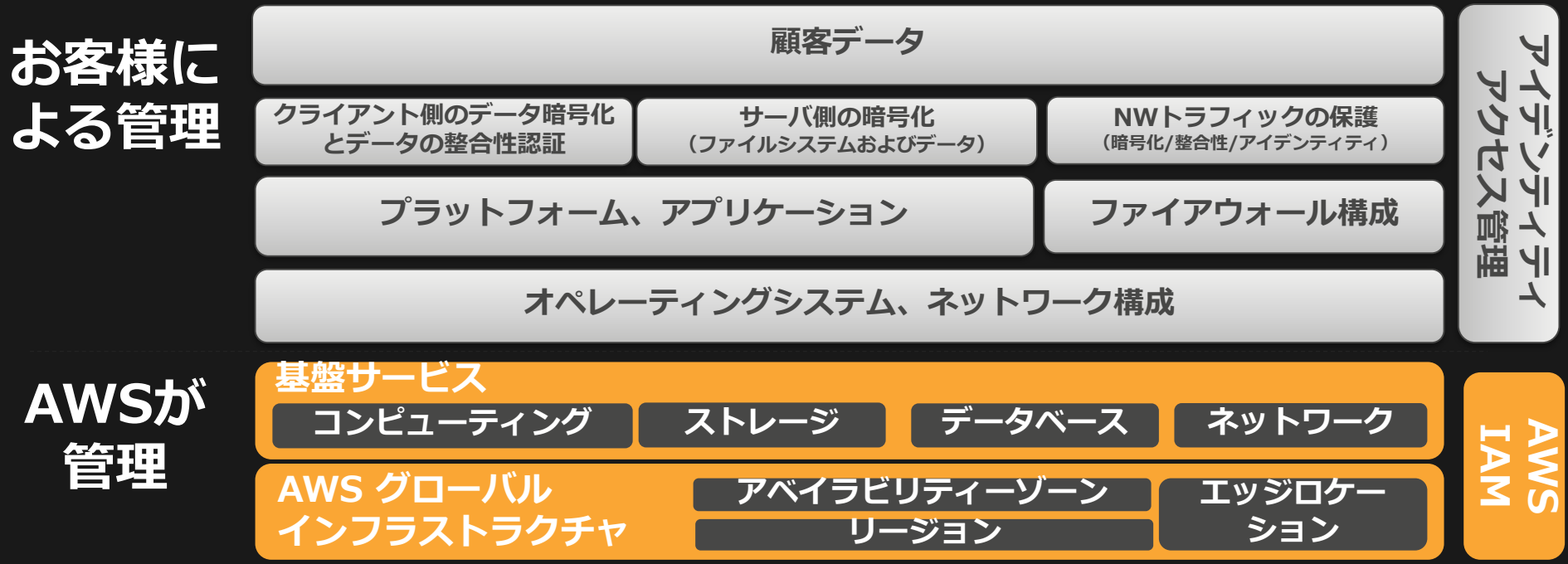
責任共有モデルから見るマネージドサービスの管理負荷

- 責任共有モデルから見るサービス区分
  - **Infrastructure Services**:
    - コンピュートとそれに関わるサービス
    - OS, Apps などの設定を自由に行うことが可能
    - 例）Amazon EC2, Amazon EBSなど
  - **Container Services**:
    - Amazon EC2などのインフラストラクチャ上で提供されるマネージドサービス
    - オプション(例:RDS Multi-AZ)などの設定により簡単に可用性を確保可能
    - 例）Amazon RDS, ELBなど
  - **Abstracted Services** :
    - プラットフォームが抽象化されたマネージドサービス(サーバレスサービス)
    - 可用性は AWS によって自動的に確保
    - 例）Amazon S3, Amazon DynamoDBなど

詳細：http://media.amazonwebservices.com/jp/wp/AWS_Security_Best_Practices.pdf

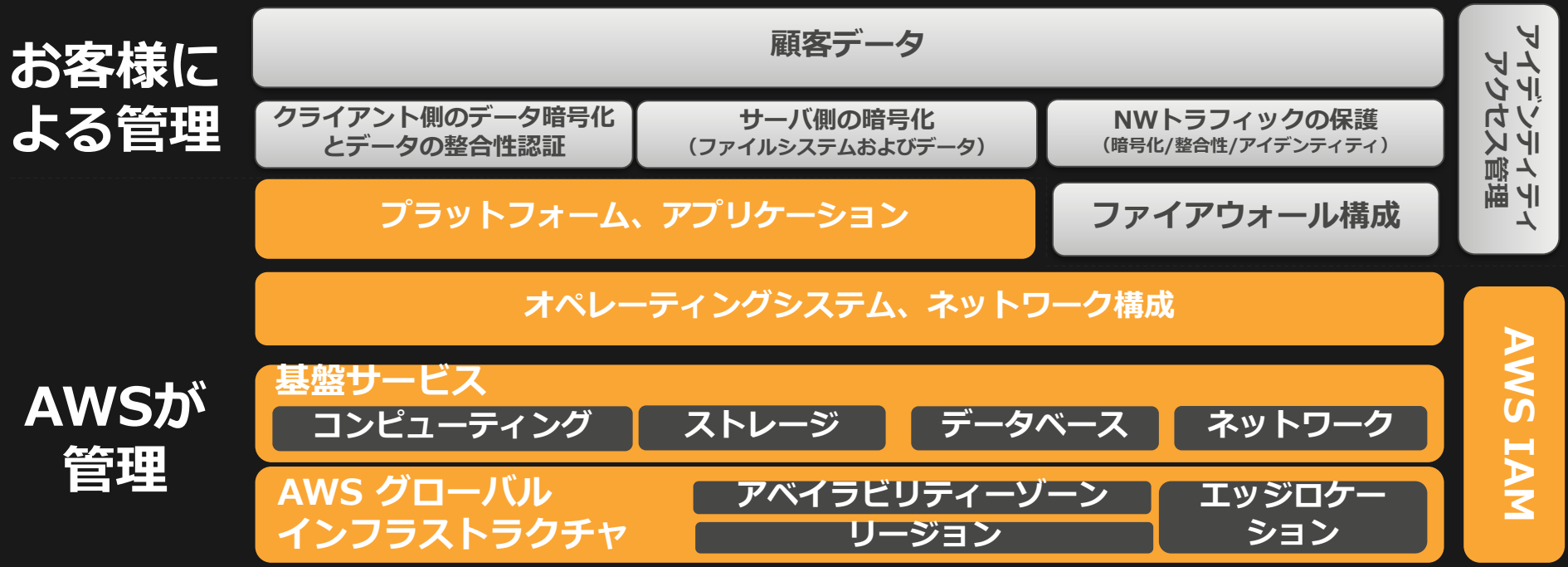
Services, Not Servers

セキュリティ責任共有モデル (Infrastructure Services)



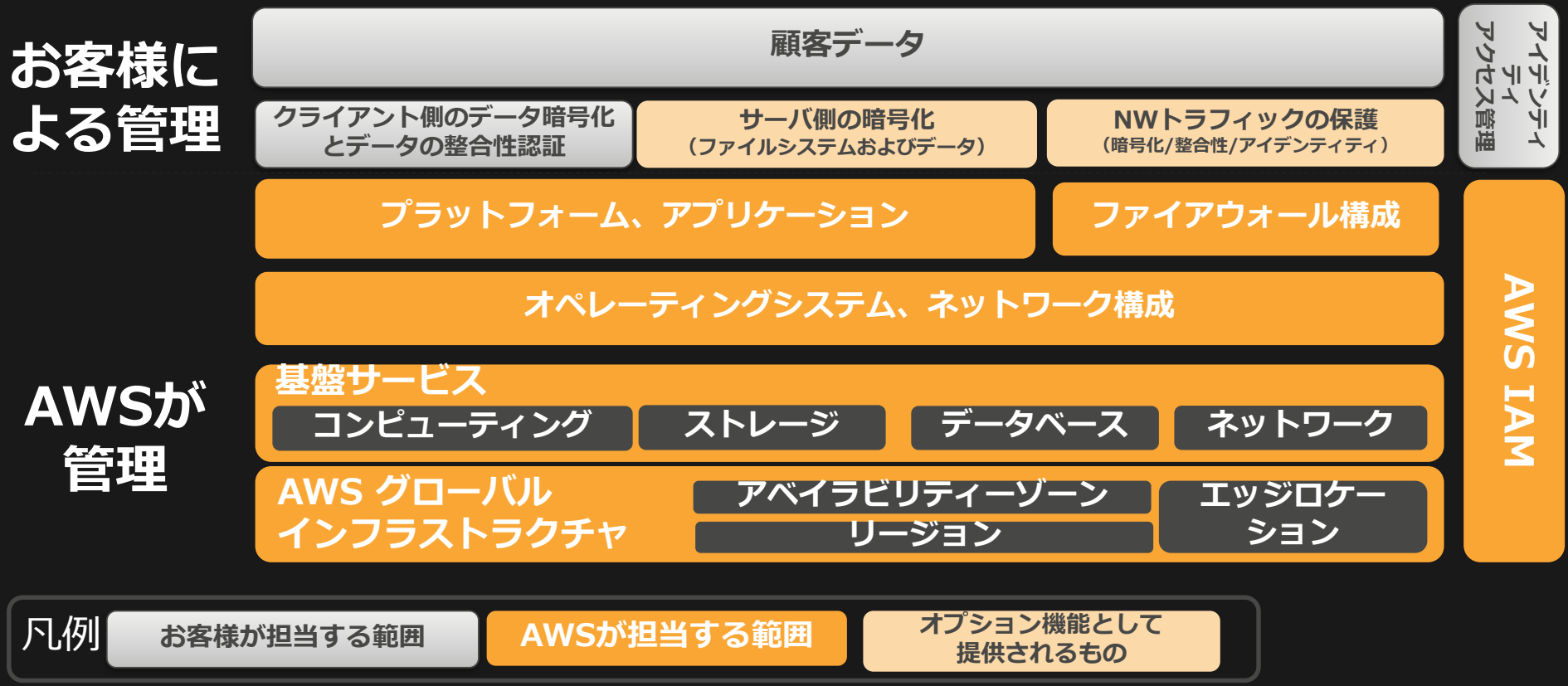
Services, Not Servers

セキュリティ責任共有モデル (Container Services)



Services, Not Servers

セキュリティ責任共有モデル (Abstracted Services)



Services, Not Servers

AWS では多くのサーバーレスオプションを用意


ストレージ


ネットワーク


データベース


コンピューティング


セキュリティ


メッセージングとキュー


コンテンツ配信


ユーザー管理


モニタリングとログ記録


機械学習


ゲートウェイ


ストリーミング分析


IoT

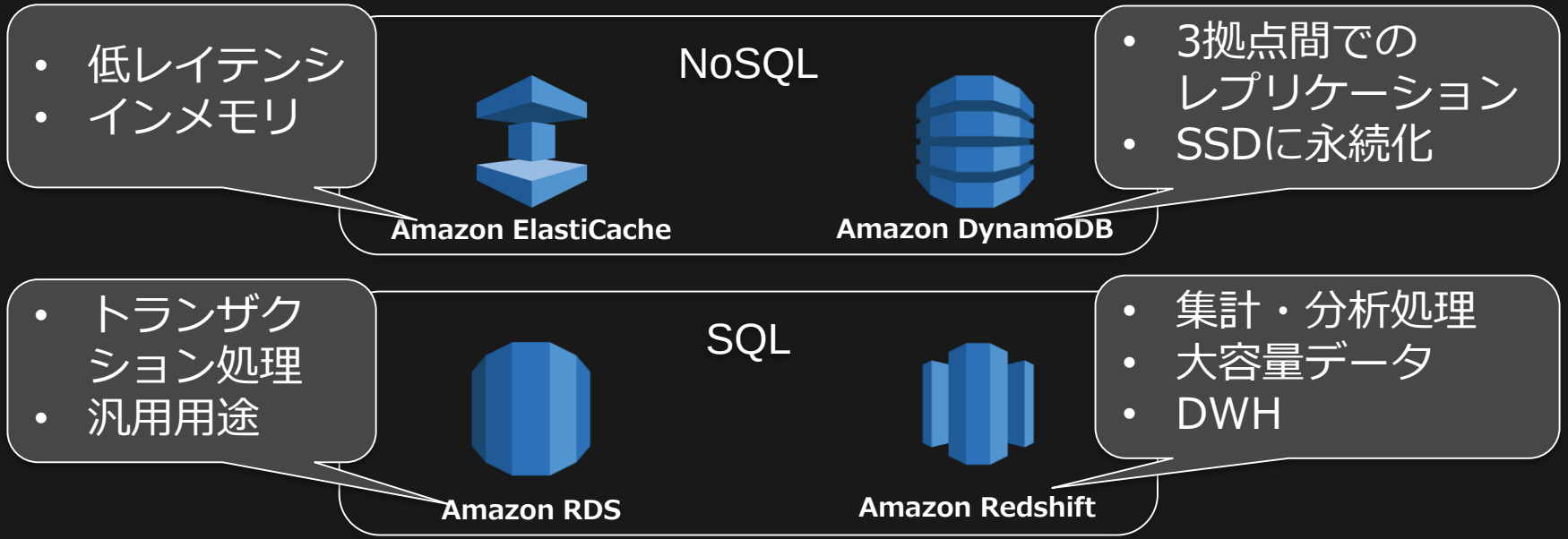
Services, Not Servers

データベースの使い分け

- Database -

データベースの使い分け

- 万能なデータベース・ストレージは存在しない
- データストアの特性(得意不得意)に応じた使い分け

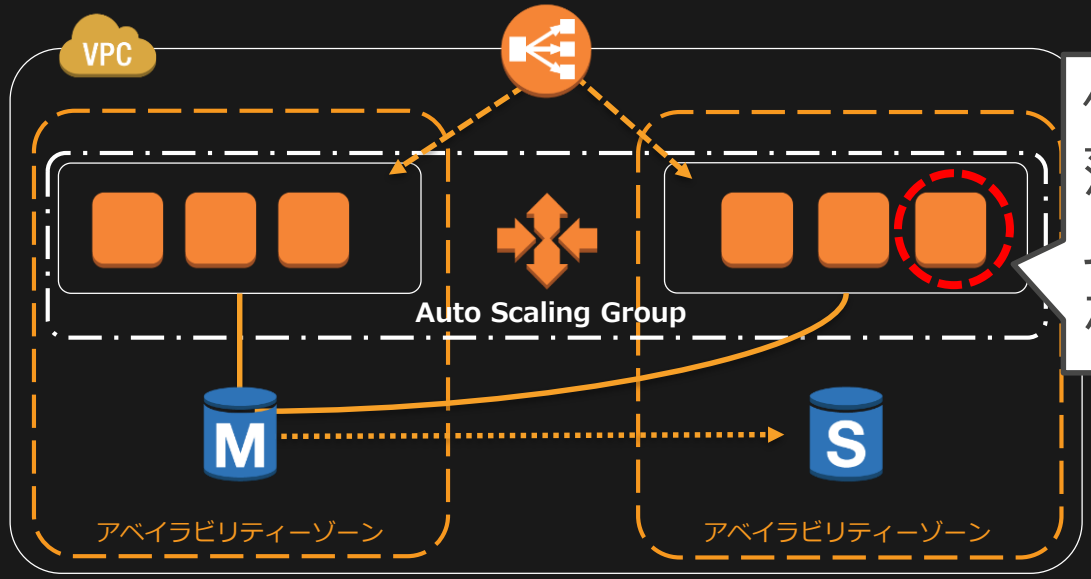


単一障害点の排除

- Removing Single Points of Failure -

単一障害点の排除

- 障害に備えておくことで、その影響を極小化
  - あらかじめ冗長化をしておくなどの準備が重要



仮に、この1台が落ちても、システムの全停止にはつながらない

Removing Single Points of Failure

単一障害点の排除

- 単一障害点排除のためのポイント
  - 冗長化
  - 障害検知
  - 堅牢なデータストレージ
  - Multi-AZ の利用
  - 疎結合な構成による障害分離

参考：https://d0.awsstatic.com/whitepapers/aws-building-fault-tolerant-applications.pdf

Removing Single Points of Failure

コストの最適化

- Optimize for Cost -

コストの最適化

- 単に既存システムをクラウドに移行するだけでも
  初期投資を減少させ、AWS の規模の経済の恩恵を享受可能

- クラウド環境の特性を活かすことにより、
  さらなるコスト最適化が可能
  - 適切なサイジング
  - 伸縮自在性
  - 適切な購入オプションの利用

Optimize for Cost

適切なサイジング

- オーバープロビジョニングの回避
  - AWS のリソースはいつでも増減可能

- 継続的な改善
  - 適切なインスタンスタイプを見極めるために常に監視
    - Amazon CloudWatch によるリソース監視
    - AWS Trusted Advisor で使用率の低いリソースを検知

Amazon CloudWatch

AWS Trusted Advisor

伸縮自在性

- 水平スケーリングにより必要な時に必要なリソースを確保

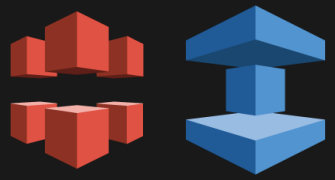
- マネージドサービスのキャパシティを利用
  - ELB, CloudFront, Amazon SQS, AWS Lambda etc
  - キャパシティを設定できるリソースの活用
    - Amazon DynamoDB, Amazon Kinesis Stream

適切な購入オプションの利用

- Amazon EC2 の購入オプション
  - リザーブドインスタンス
  - スポットインスタンス
- Amazon S3 のストレージ価格
  - スタンダードストレージ
  - 標準 – 低頻度アクセスストレージ
  - 低冗長化ストレージ
- その他のサービスの購入オプション
  - CloudFrontのリザーブドキャパシティ
  - DynamoDBのリザーブドキャパシティ

キャッシュの利用

- Caching -

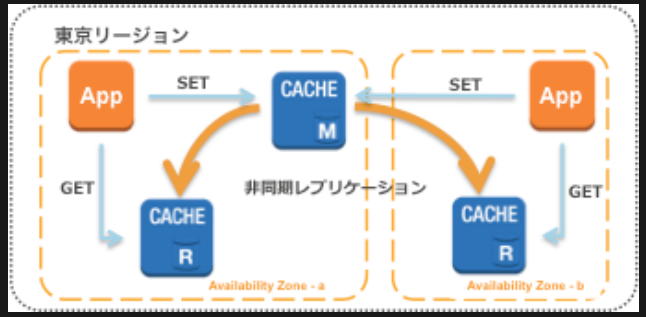


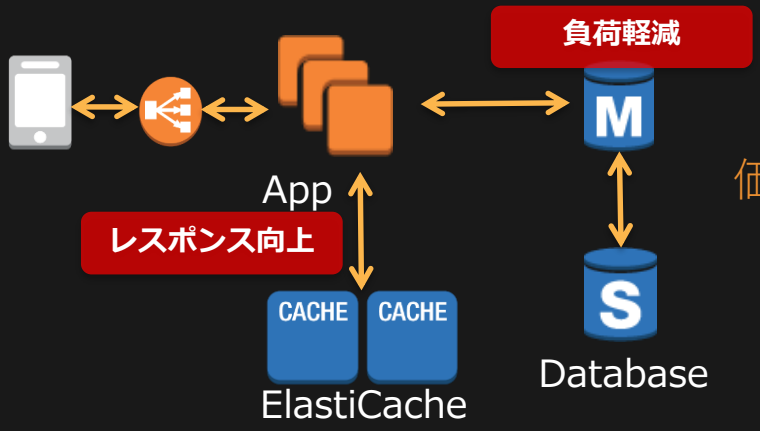
キャッシュの利用

- 繰り返し利用されるデータをキャッシュ
  - 性能向上
  - コスト節約

Amazon ElastiCache による Application Data Caching

フルマネージド インメモリキャッシュサービス





特徴 (https://aws.amazon.com/jp/elasticache/)

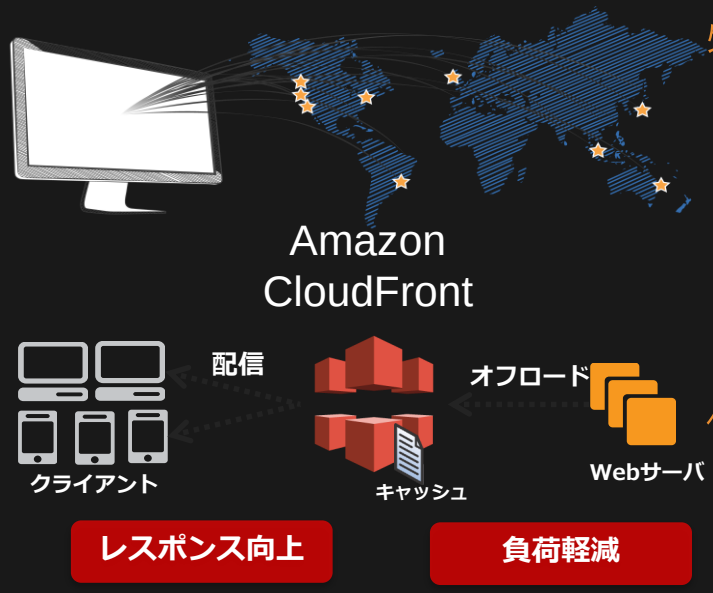
- フルマネージド環境でMemcached / Redisが利用可能
- RedisはMulti-AZ配置することで可用性向上
- 一部パラメータ以外はアプリケーション特性に応じて変更可能
- フェイルオーバーやパッチの適用、バックアップ(Redis)も自動で行われる
- Memcached用のAuto Discovery対応Client Libraryも提供中

価格体系 (https://aws.amazon.com/jp/elasticache/pricing/)

- インスタンスタイプに応じて
- Redisを利用しバックアップを有効にした場合はバックアップストレージの利用量に応じて

Amazon CloudFront による Edge Caching

マネージドCDN(Contents Delivery Network)サービス



特徴 (http://aws.amazon.com/jp/cloudfront/)

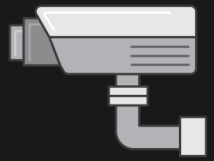
- 簡単にサイトの高速化が実現できると共に、サーバの負荷も軽減
- 様々な規模のアクセスを処理することが可能
- 世界53箇所のエッジロケーション

価格体系 (http://aws.amazon.com/jp/cloudfront/pricing/)

- データ転送量(OUT)
- HTTP/HTTPSリクエスト数
- (利用する場合)SSL独自証明書 など

セキュリティ

- Security -



セキュリティ

- すべてのレイヤーでセキュリティを確保
  - 一箇所でもセキュリティホールがあれば、そこを突かれてしまう
- セキュリティ確保のためのポイント
  - AWS の機能の活用
  - AWS へセキュリティの担保をオフロード
  - 最小権限の原則
  - Security as Code
  - リアルタイム監査

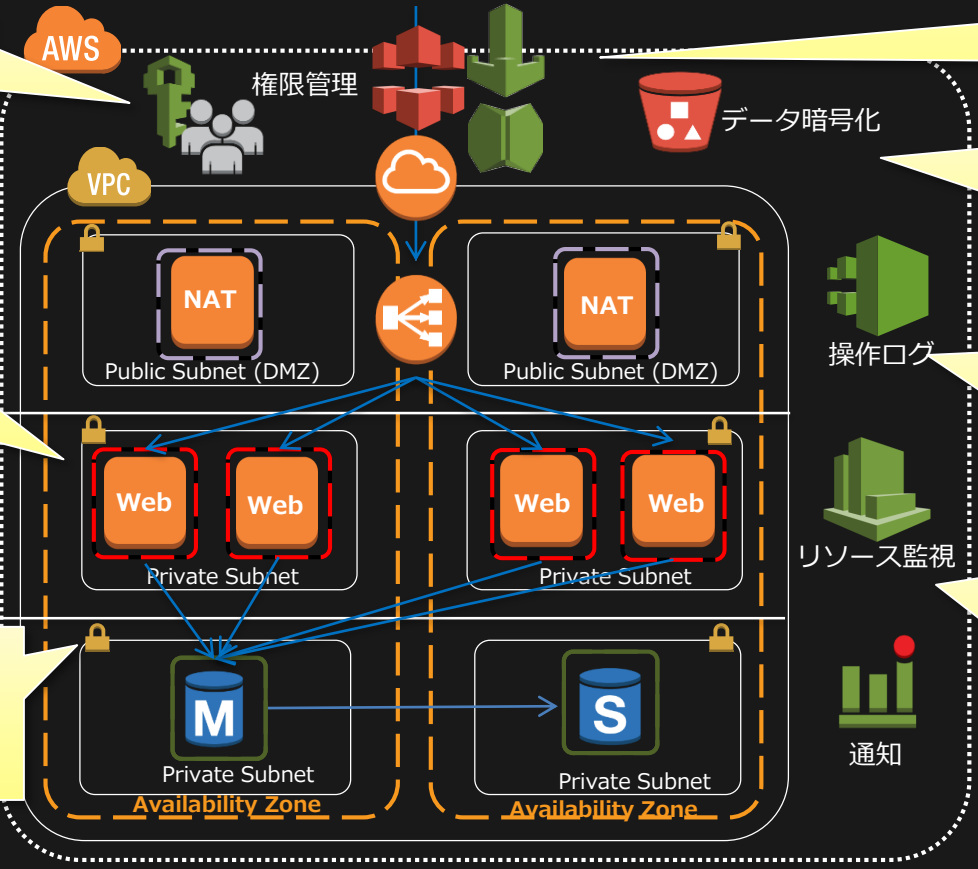
詳細：http://www.slideshare.net/AmazonWebServicesJapan/awswebinar-aws-56260969

すべてのレイヤーでセキュリティを確保

【アクセス管理】
AWSアカウント・IAMユーザーの管理。AWSリソースへのアクセス制御(最小権限の原則を順守可能)

【サブネット】
外部からアクセスできるサブネットと、外部からはアクセスできないサブネットの作成

【ネットワークアクセス制御】
SecurityGroup及びNetwork ACLを使ってアクセス制御を実施



【DDoS対策】AWS WAF, Shield による DDoS 緩和を実施

【保管するデータの暗号化】
S3やEBS、RDSといったストレージサービス上のデータを暗号化

【AWS操作ログ】
AWS操作ログの取得（管理コンソールやCLI含む）

【AWSサービス監視】
各種AWSサービス（ELB、RDS、EC2等）のリソース監視

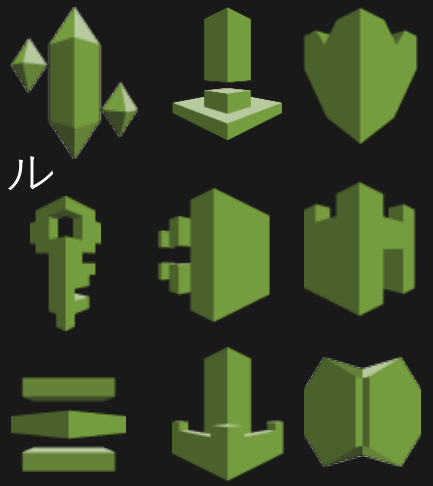
詳細： http://www.slideshare.net/AmazonWebServicesJapan/awswebinar-aws-56260969

Security

AWS の機能の活用

- AWSが提供する便利なセキュリティ機能を活用することでよりシンプルにセキュリティ確保が可能に

- AWS が提供するセキュリティ機能活用の例
  - AWS IAM による権限制御
  - セキュリティグループによるアクセスコントロール
  - AWS WAF/ Shield による DDoS 緩和
  - AWS CloudTrail による監査ログ取得
  - AWS KMS による暗号化鍵管理
  - etc



Security

Agenda

- はじめに
- クラウドコンピューティングの特徴
- クラウドの特徴を活かすための設計原則
- まとめ

まとめ(Ten Design Principles)

- スケーラビリティ
- 常設のサーバではなく使い捨て可能なリソース
- 自動化
- 疎結合
- サーバではなく、サービスの利用(マネージドサービスの活用)
- データベースの使い分け
- 単一障害点の排除
- コストの最適化
- キャッシュの利用
- セキュリティ

参考資料

- Architecting for The Cloud: AWS Best Practices
  - https://d0.awsstatic.com/whitepapers/AWS_Cloud_Best_Practices.pdf
- クラウドのためのアーキテクチャ設計 -ベストプラクティス
  - https://www.slideshare.net/AmazonWebServicesJapan/aws-black-belt-online-seminar-2016
- 失敗例を成功に変える AWS アンチパターン
  - 2016年：http://www.slideshare.net/AmazonWebServicesJapan/aws-black-belt-online-seminar-antipattern
  - 2015年：http://www.slideshare.net/AmazonWebServicesJapan/20150609-antipattern-49198289

本セッションのFeedbackをお願いします

受付でお配りしたアンケートに本セッションの満足度やご感想などをご記入ください
アンケートをご提出いただきました方には、もれなく素敵なAWSオリジナルグッズをプレゼントさせていただきます



アンケートは受付、又はパミール3FのEXPO展示会場内にて回収させて頂きます

AWS

SUMMIT

Thank you!

amazon web services