AWSシステム開発

神戸大学大学院システム情報学研究科
中村　匡秀

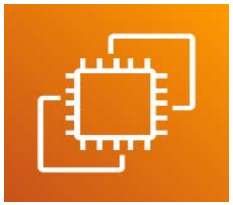
内容

- 目的：実用的なシステムをクラウドに構築する際に，AWSをどのように活用すべきかを，実例を見ながら学習する
- 第1部：AWSを活用したシステム開発
- 第2部：AWSクラウドデザインパターン（時間があれば）

第1部：AWSを活用したシステム開発

Amazon EC2

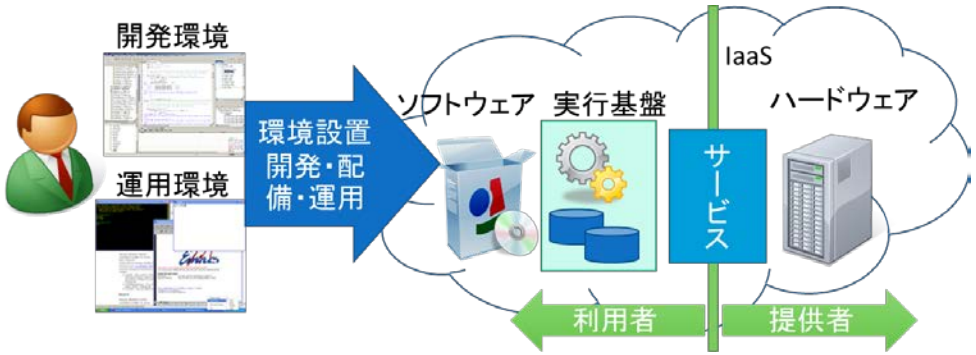
- Amazon **E**lastic **C**omputing **C**loud (EC2)
  - 汎用仮想サーバを時間貸しするIaaS
  - 必要な時に必要な数だけサーバを調達
  - 様々なタイプのインスタンスをAMIから生成
  - 従量課金 （t2.microの場合，1時間0.0116USD = 1.19円）


Amazon EC2

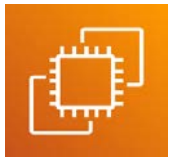
- IaaSなので利用者が自由にカスタマイズ可能
  - サーバソフトウェア
    - Apache，Tomcat, node.js, nginx,
  - プログラミング言語・実行系
    - Java, python, C, PHP
  - データベース
    - MySQL, PostgreSQL MongoDB
  - アプリケーション

```
# yum install -y httpd
```

Amazon Linux 2



EC2で何でもできる

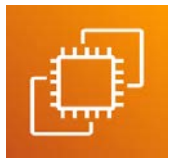



Webサーバ



Webアプリ
サーバ

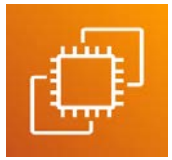
Apache
Tomcat



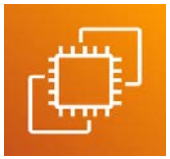

メール
サーバ


関係データ
ベース


ドキュメント
ストア

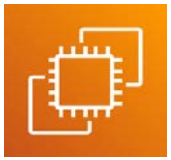



キーバリュー
ストア




コンテンツ
管理システム


データ分析
システム




ファイル
サーバ

Webサイトにブログコーナーを追加

- EC2に**LAMPスタック**を入れて，**WordPress**を導入
  - Linux, Apache, MySQL (or MariaDB), PHP
  - WordPress: 著名なCMS．ブラウザでコンテンツを編集・管理できる
    - https://docs.aws.amazon.com/ja_jp/AWSEC2/latest/UserGuide/hosting-wordpress.html

- 機能的にはこれでOK

コンテンツ管理

Webサーバ

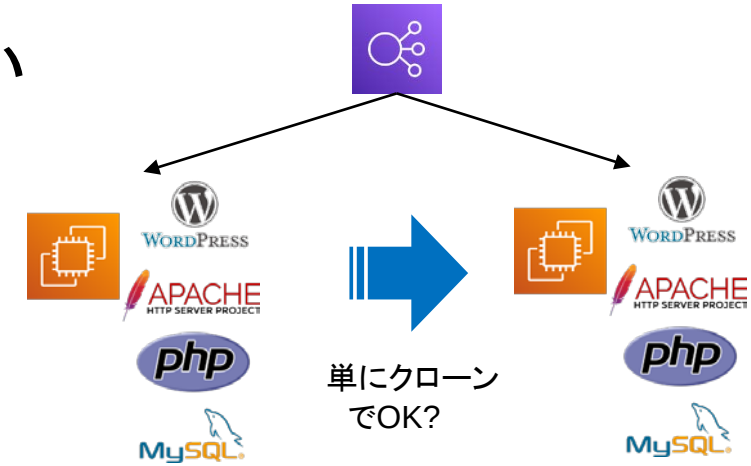
動的HTML生成

データベース

「何でもできる」の裏返しは

- 設置から運用まであらゆる管理を自分でやらないといけない
  - インスタンス作成・設定
  - ソフトウェアのインストール・アップデート
  - コンテンツの転送・配備
  - バックアップ
  - クローン＆負荷分散

- スケーラビリティの確保は簡単でない
  - アクセスが増えてきたら？
  - アプリの機能が重たくなってきたら？
  - データが増えてきたら？



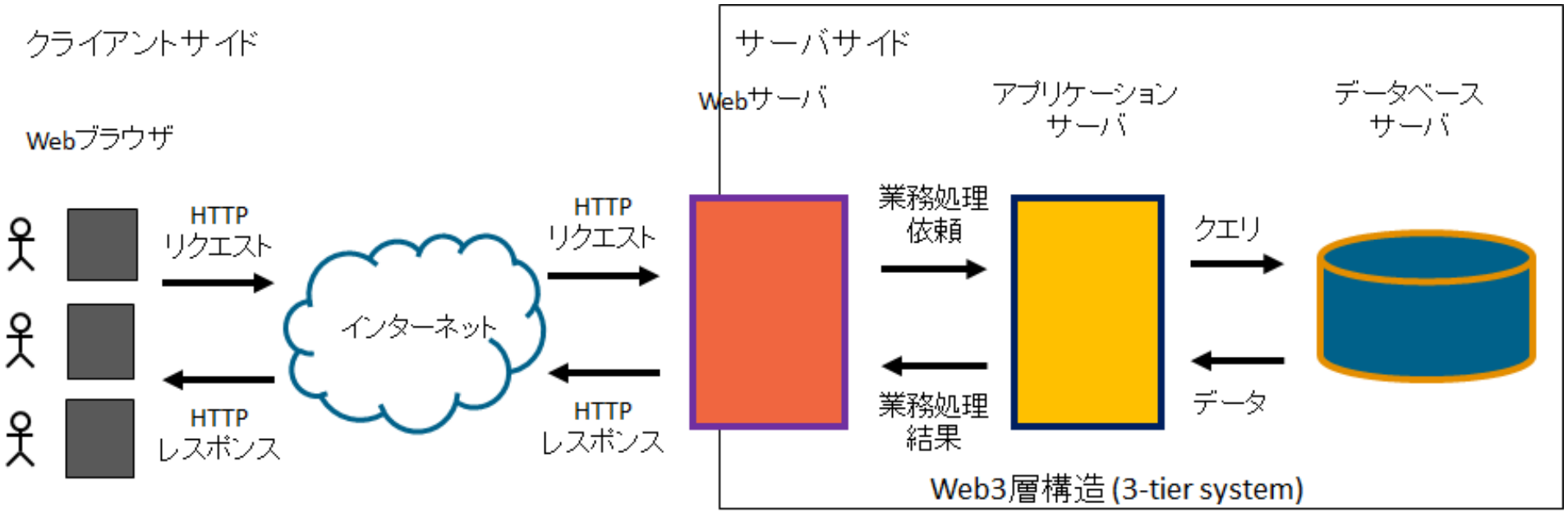
- 実用システムの構築にはアーキテクチャの設計が重要

システム・アーキテクチャ

- **定義（IEEE1471）**：アーキテクチャとは、「コンポーネント」，コンポーネント間および「環境」との「関係」，またその設計と進化の指針となる原理に体現された「システム」の基本「構造」である
  - どのような構成要素をどのようにつないで全体システムを構成するかを表した構造
- **狙い**：ある程度の枠にはめることで，設計・実装の見通しの確保，信頼性・保守性の向上，環境変化への適応等を狙う
  - 特に非機能要件 (保守性，性能，セキュリティ，スケーラビリティ等)を考慮して決められることが多い
- **パターン**：アプリケーションやサービスに応じた典型的なパターンが存在する
- **記法**：特に決まった記法はなく，ブロック図やポンチ絵で描かれることが多い

Web3層アーキテクチャ

- Webアプリケーションを，Webサーバ，アプリケーションサーバ，データベースサーバの3層で構成するアーキテクチャ

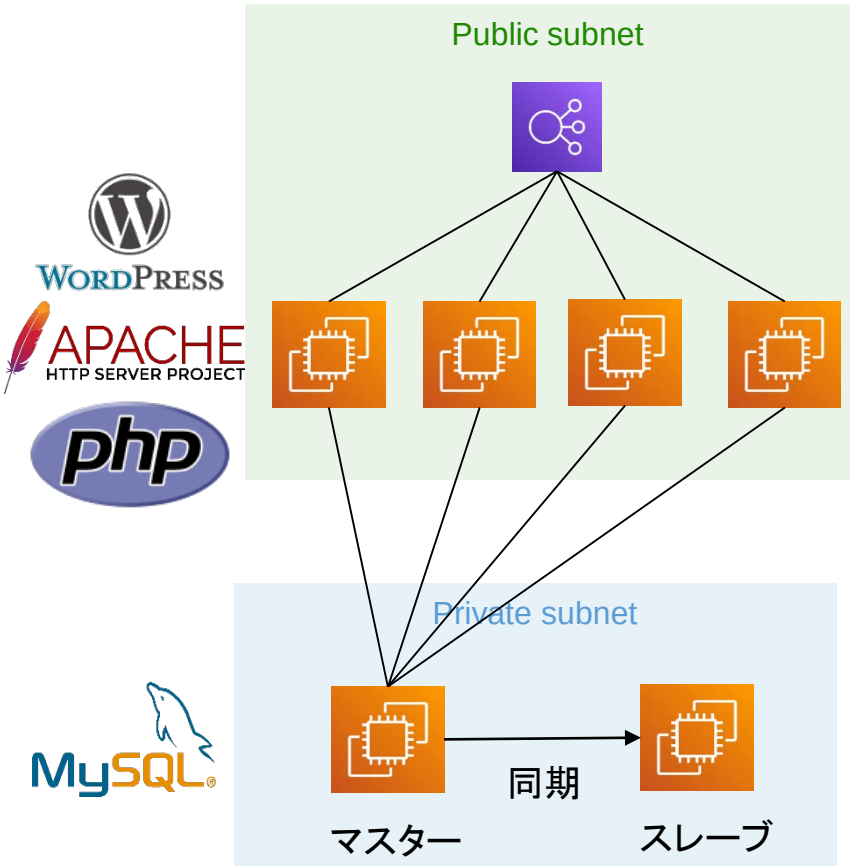


※PHPの場合，Apacheと同じサーバで動作させるケースが多い (Web2層)

アーキテクチャを考えておくと

- システムに対する非機能要件（品質要件）に対応しやすくなる
  - 保守性：運用・管理しやすいか？
  - 性能：応答が早いか？
  - 可用性：いつでも利用できるか？
  - セキュリティ：ハッキングや情報漏洩がないか？
  - スケーラビリティ：規模増加に耐えられるか？

リソースが増えれば管理の手間も増える



■Webサーバの運用管理
- アプリケーション更新
- セキュリティパッチ
- アクセスの監視
- 負荷分散設定
- ・・・

■ DBサーバの運用管理
- 性能チューニング
- データバックアップ
- 障害時の復旧
- ・・・

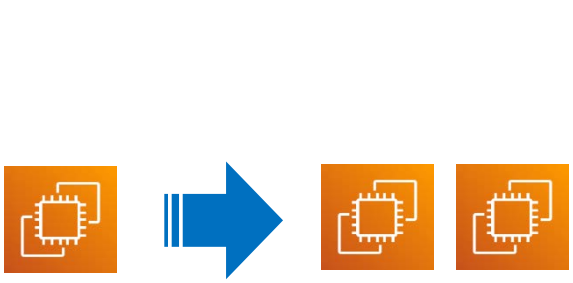
やること多すぎでしょ. . .

管理・運用を自動化する

- AWSが提供するサービスを利用して自動化を考える

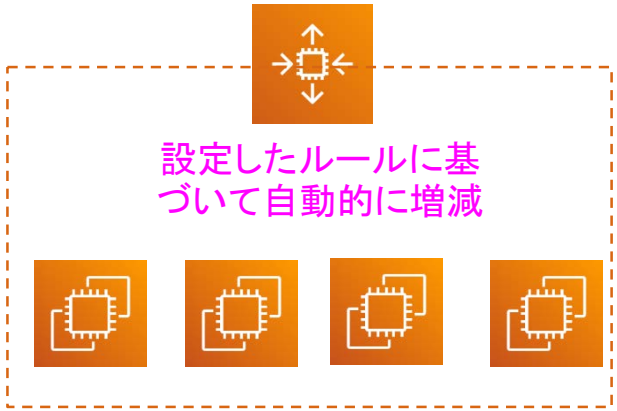
- 例：Amazon EC2 Auto Scaling
  - サーバの状況に応じて，EC2インスタンスを自動的に追加・削除する
  - 負荷に基づくスケーリング
  - スケジュールに基づくスケーリング

Amazon EC2 Auto Scaling



手動でクローン

自動化



マネージド型サービス (Managed Services)

- 機能に加えて運用・管理の作業も提供してくれるサービス
  - 特定の業務やアプリに依存しない部分をAWS側で自動化
  - 一般に価格は高くなる

Amazon RDS

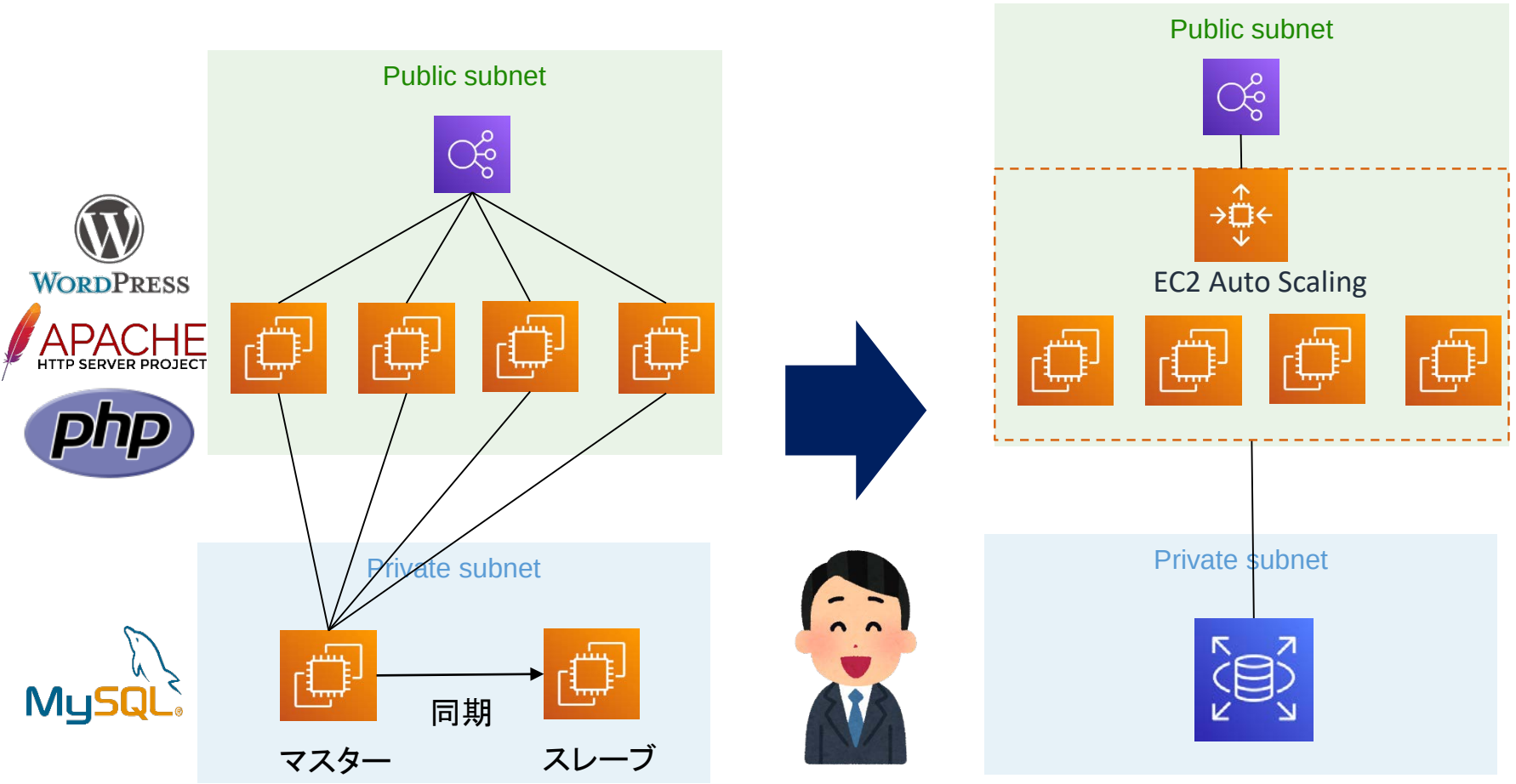
- 例：Amazon RDS
  - 関係データベース(RDB)のためのマネージド型サービス
  - サーバインスタンスの配備，データベースセットアップ，パッチ適用，バックアップ等の管理タスクを自動化
  - MySQL等の6つのDBエンジンから選択可能
    - 中身はDBがインストールされたEC2が動いている

マネージド型サービスの恩恵

出展: https://www.slideshare.net/AmazonWebServicesJapan/20180410-aws-white-belt-online-seminar-aws

ポータルサイトの新アーキテクチャ

- 運用・管理の手間を削減して業務の生産性を向上
  - コンテンツ作成に集中できる



IAMによる職務分掌

- システムが大きくなってくると，複数人での運用・管理が必須
  - セキュリティ，コンプライアンスの観点から，誰がどのリソースにどのようにアクセスしてよいかを決めておく必要がある
    - ルートユーザーでは何でもできてしまうので，作業者の職務に応じて権限を絞る（職務分掌）

AWS Identity & Access Management

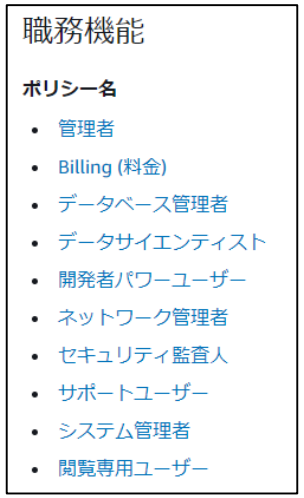
- AWS Identity & Access Management (IAM)
  - ルートユーザが，運用・管理担当者それぞれに新しいアカウント (IAM ユーザ) を発行する
  - IAMユーザごとに権限ポリシー(IAM ポリシー)を設定可能

EC2の起動，状態確認ができる

開発者

EC2の起動・停止，状態確認，設定変更ができる

システム管理者



- 人間ではなくリソースにも権限を割り当てられる (IAMロール)

さらなる便利を追求

- Amazon Lightsail
  - 自分専用の仮想サーバ(VPS)を提供するサービス
  - 必要な設定やソフトがあらかじめ入っている
  - 有名なソフトもクリックでインストール可
  - 料金は従量制ではなく月額払い

Amazon Lightsail

- AWS Elastic Beanstalk
  - Webアプリを配備・実行するプラットフォームを提供するサービス (PaaS)
  - 開発者はコードをアップロードするだけ
  - EC2, Auto Scaling, Load Balancer 等の主要サービスが必要に応じて動作する

AWS Elastic Beanstalk

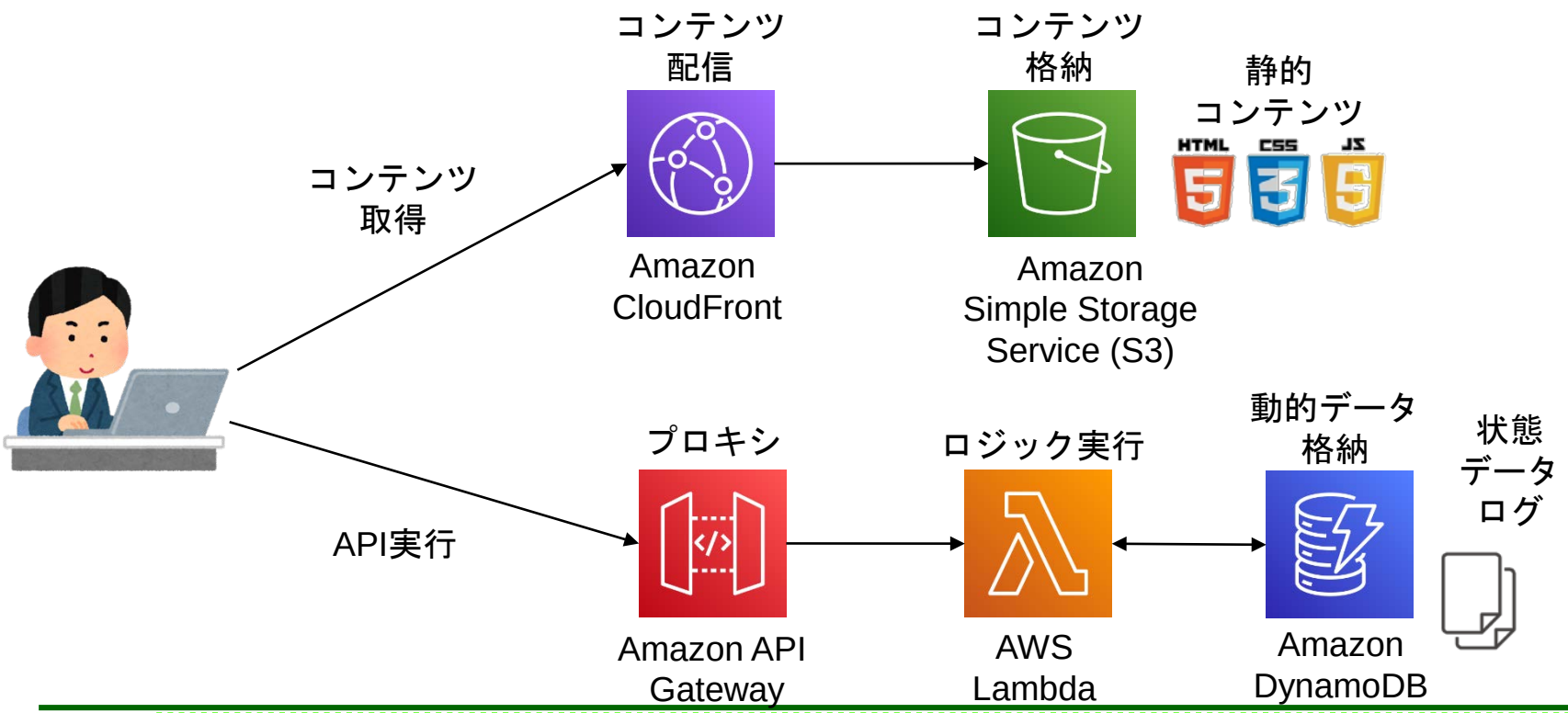
Function as a Service

- AWS Lambda
  - 任意のプログラム（関数）をAWSの基盤で実行してくれるサービス
    - 任意の条件（イベント，リクエスト）で関数をトリガ
    - 他のAWSリソースと連携可能
  - 呼び出し回数，実行時間，割当メモリで課金
    - 100万リクエストで0.2USD
    - GB-秒あたり0.0000166667USD
    - 呼び出しがなければ無料
  - 制約
    - ステートレスな関数に限る
    - ディスク容量：512MB
    - メモリ：128MB-1536MB
    - タイムアウト：15分

AWS
Lambda

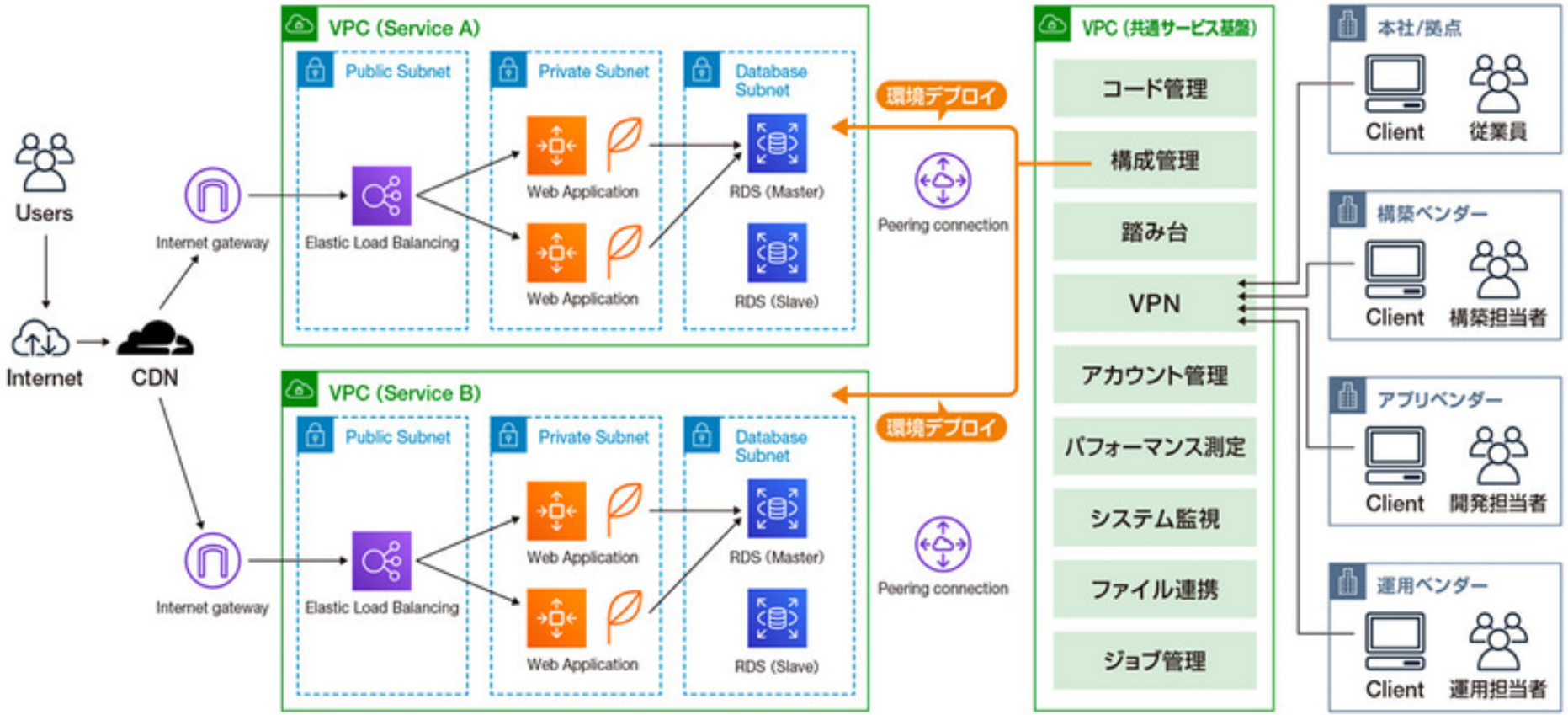
サーバレスアーキテクチャ

- FaaSを使ってサーバそのものを持たなくするアーキテクチャ
  - 処理はLambdaで実行
  - コンテンツやステートはS3やDynamoDBで管理
  - リクエストがあった時だけ，コンテンツやサービスを提供



事例1：企業のAWSマイグレーション

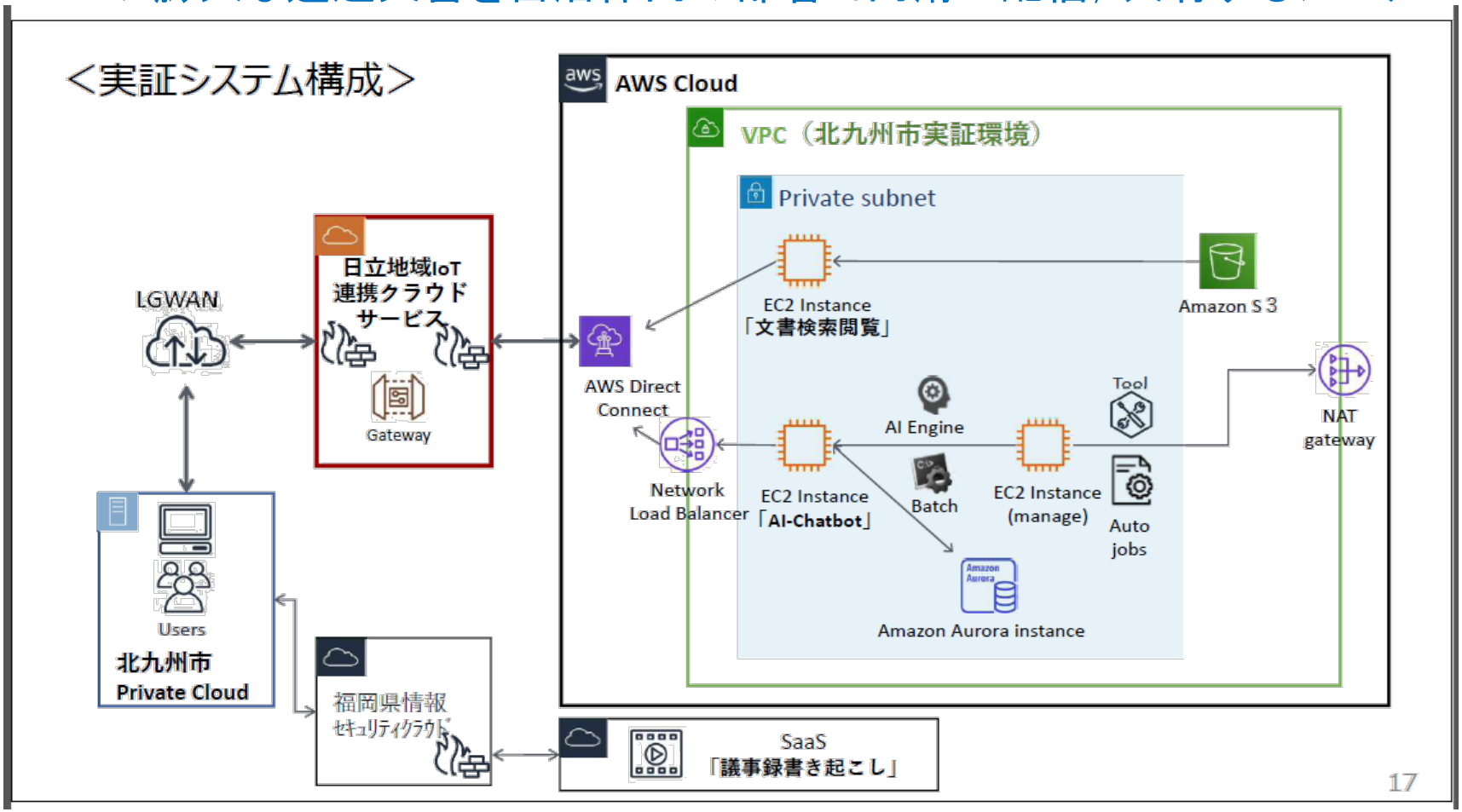
- 蔦屋書店が15のサービスをAWSへ移行、システム標準化や保守作業の削減に成功



出典：クラウドWatch 2021年1月14日：https://cloud.watch.impress.co.jp/docs/news/1299886.html

事例2：自治体のAWS活用

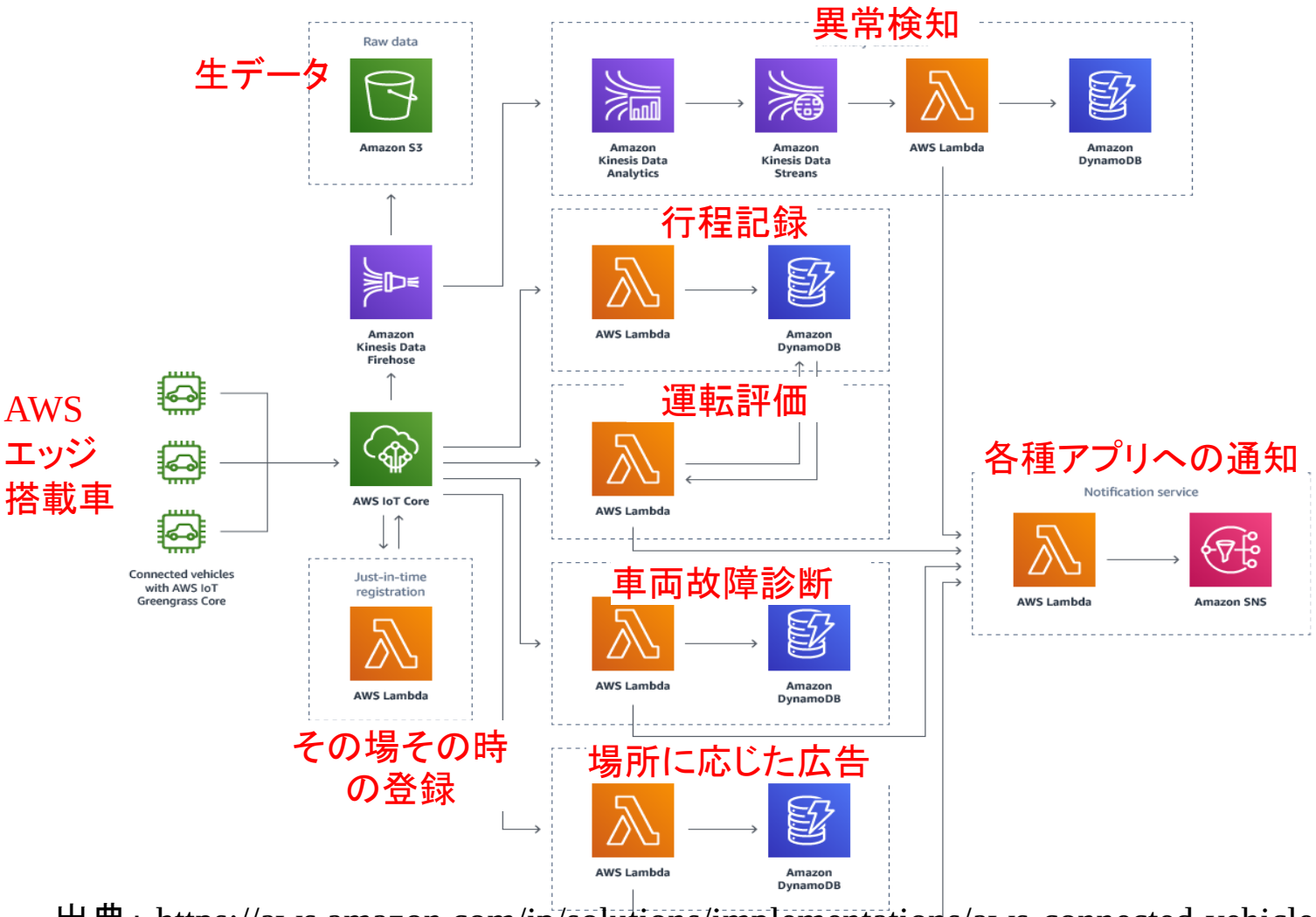
- 北九州市「AIによる自治体業務総合支援実証事業」
  - ◆膨大な通達文書を自治体内の部署で円滑に配信，共有するシステム



出典：AWS Summit 2020 - CUS11自治体でのクラウドバイデフォルトの実現

事例3：AWS コネクテッドカーソリューション

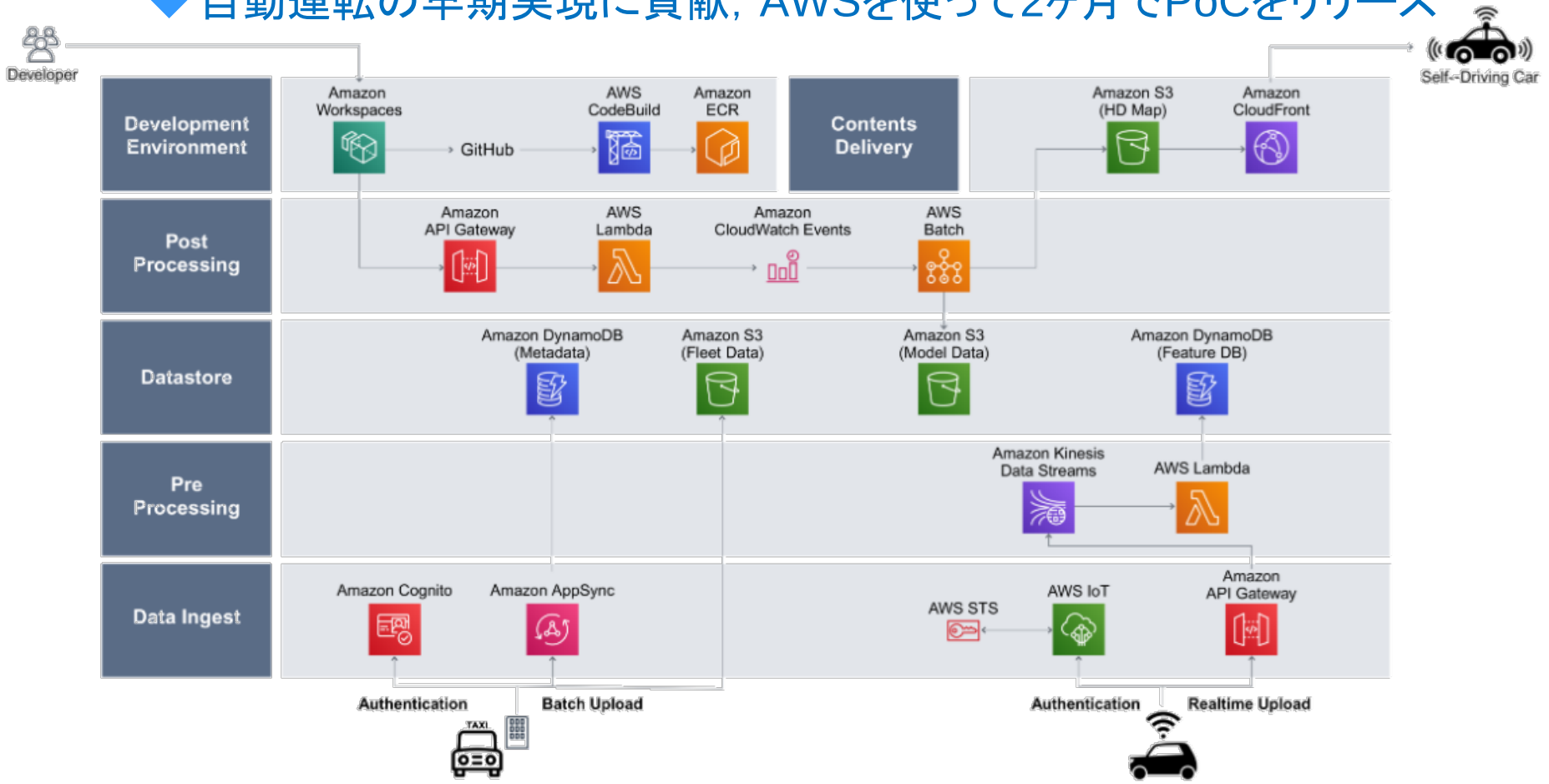
- AWS IoTで車からデータをクラウドに集め，Lambdaで処理



出典：https://aws.amazon.com/jp/solutions/implementations/aws-connected-vehicle-solution/

事例4: TRI-AD自動地図生成プラットフォーム

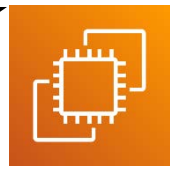
- Automated Mapping Platform（AMP）
  - 一般道の高精度の地図を生成，更新していく地図情報基盤
  - 自動運転の早期実現に貢献，AWSを使って2ヶ月でPoCをリリース



https://aws.amazon.com/jp/solutions/case-studies/tri-ad/

まだまだたくさんあるAWSのサービス

計算装置

★  Amazon EC2

★  Amazon Lightsail

★  AWS Lambda

★  AWS Elastic Beanstalk

ネットワーク

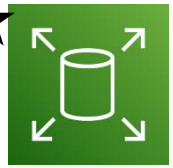
★  Amazon VPC

★  Amazon Route 53

★  Amazon CloudFront

★  Elastic Load Balancing

記憶装置

★  Amazon Elastic Block Store (EBS)

★  Amazon Simple Storage Service (S3)

データベース

 Amazon Aurora

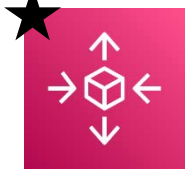
★  Amazon RDS

★  Amazon DynamoDB

管理・統制

 Amazon CloudWatch

★  AWS Mgmt Console

★  AWS Auto Scaling

サービス連携・統合

 Amazon Simple Notification Service

 Amazon Simple Queue Service

 AWS Step Functions

移行・マイグレーション

 AWS Server Migration Service

 AWS Database Migration Service

まだまだたくさんあるAWSのサービス

認証・セキュリティ


AWS Identity & Access Management


AWS Shield


AWS Resource Access Manager


AWS Firewall Manager

開発ツール


AWS Cloud9


AWS Command Line Interface

ビジネスツール


Amazon WorkMail


Alexa for Business

モバイル


AWS Amplify


Amazon API Gateway

コンテナ


Amazon Elastic Container Registry


Amazon Elastic Container Kubernetes

モノのインターネット（IoT）


AWS IoT Core


AWS IoT Analytics


AWS IoT Device Management

ビッグデータ


Amazon EMR


AWS Lake Formation

AI・機械学習


Amazon SageMaker


AWS Deep Learning AMIs


Amazon Rekognition

重要なこと

- AWSはプラットフォームを提供してくれるだけ
  - AWSはシステムを組み立てるブロックとシステム置き場所を提供
  - **価値を生む部分（＝付加価値サービス）は自分で考える必要がある**
- システム開発においても目的指向の考え方が重要
  - 何のために何を作るのか？ (Why, What)
  - どんなお客様にどんなコトを提供してどんな状態になっていただくか？
  - その結果，何の名目でいくら払っていただくか？
  - システムは手段（＝ソリューション）で目的ではない
- 技術駆動の考え方では失敗
  - 「最新の車載システム技術をビジネスにできないか？」
  - 「ディープラーニング使ってみたいから，なんか考えよう」

第1部のまとめ

- システム開発にはアーキテクチャが重要
  - EC2は何でもできるけど．．．
  - Web3層アーキテクチャ
- システム管理・運用にはコストがかかる
  - サービスを活用した自動化
  - マネージド型サービスの活用
  - IAMによる認証・認可
- 発展的なサービス
  - Lightsail (VPS), Elastic Beanstalk (PaaS)
  - Lambda (FaaS)，サーバレスアーキテクチャ
  - 実際の活用事例の紹介
- AWSはあくまでプラットフォームを提供してくれるだけ
  - 付加価値サービスは自分たちで考える必要がある

第2部：AWSクラウドデザインパターン

AWSクラウドデザインパターン

- AWSを活用してシステムを開発する際によく用いられる典型的なアーキテクチャをパターンとしてカタログにしたもの
  - Ninja of Three (玉川 憲, 片山 暁雄, 鈴木 宏康) によって策定
  - 各デザインパターンは次の項目で説明される
    - パターン名・サマリ
    - 解決したい課題
    - クラウドでの解決・パターンの説明
    - 実装
    - 構造
    - 利点
    - 注意点
    - その他

出典：http://aws.clouddesignpattern.org/

用途に応じたアーキテクチャのパターン

基本のパターン

- Snapshotパターン（データのバックアップ）
- Stampパターン（サーバの複製）
- Scale Upパターン（動的なサーバのスペックアップ／ダウン）
- Scale Outパターン（サーバ数の動的増減）
- Ondemand Diskパターン（動的なディスク容量の増減）

可用性を向上するパターン

- Multi-Serverパターン（サーバの冗長化）
- Multi-Datacenterパターン（データセンターレベルの冗長化）
- Floating IPパターン（IPアドレスの動的な移動）
- Deep Health Checkパターン（システムのヘルスチェック）
- Routing-Based HAパターン（ルーティングによる接続先の透過的な切り替え）

動的コンテンツを処理するパターン

- Clone Serverパターン（サーバのクローン）
- NFS Sharingパターン（共有コンテンツの利用）
- NFS Replicaパターン（共有コンテンツの複製）
- State Sharingパターン（ステート情報の共有）
- URL Rewritingパターン（静的コンテンツの退避）
- Rewrite Proxyパターン（URL書き換えプロキシの設置）
- Cache Proxyパターン（キャッシュの設置）
- Scheduled Scale Outパターン（サーバ数のスケジュールにあわせた増減）
- IP Poolingパターン（接続許可済みIPアドレスのプール）

静的コンテンツを処理するパターン

- Web Storageパターン（可用性の高いインターネットストレージ活用）
- Direct Hostingパターン（インターネットストレージで直接ホスティング）
- Private Distributionパターン（特定ユーザへのデータ配布）
- Cache Distributionパターン（ユーザに物理的に近い位置へのデータ配置）
- Rename Distributionパターン（変更遅延のない配信）
- Private Cache Distributionパターン（CDNを用いたプライベート配信）
- Latency Based Origin（地域によりオリジンサーバを変更）

データをアップロードするパターン

- Write Proxyパターン（インターネットストレージへの高速アップロード）
- Storage Indexパターン（インターネットストレージの効率化）
- Direct Object Uploadパターン（アップロード手順の簡略化）

リレーショナルデータベースのパターン

- DB Replicationパターン（オンラインDBの複製）
- Read Replicaパターン（読込専用レプリカによる負荷分散）
- Inmemory DB Cacheパターン（頻度の高いデータのキャッシュ化）
- Sharding Writeパターン（書き込みの効率化）

非同期処理/バッチ処理のパターン

- Queuing Chainパターン（システムの疎結合化）
- Priority Queueパターン（優先順位の変更）
- Job Observerパターン（ジョブの監視とサーバの追加・削除）
- Fanoutパターン（複数種類の処理を非同期かつ並列に実行）

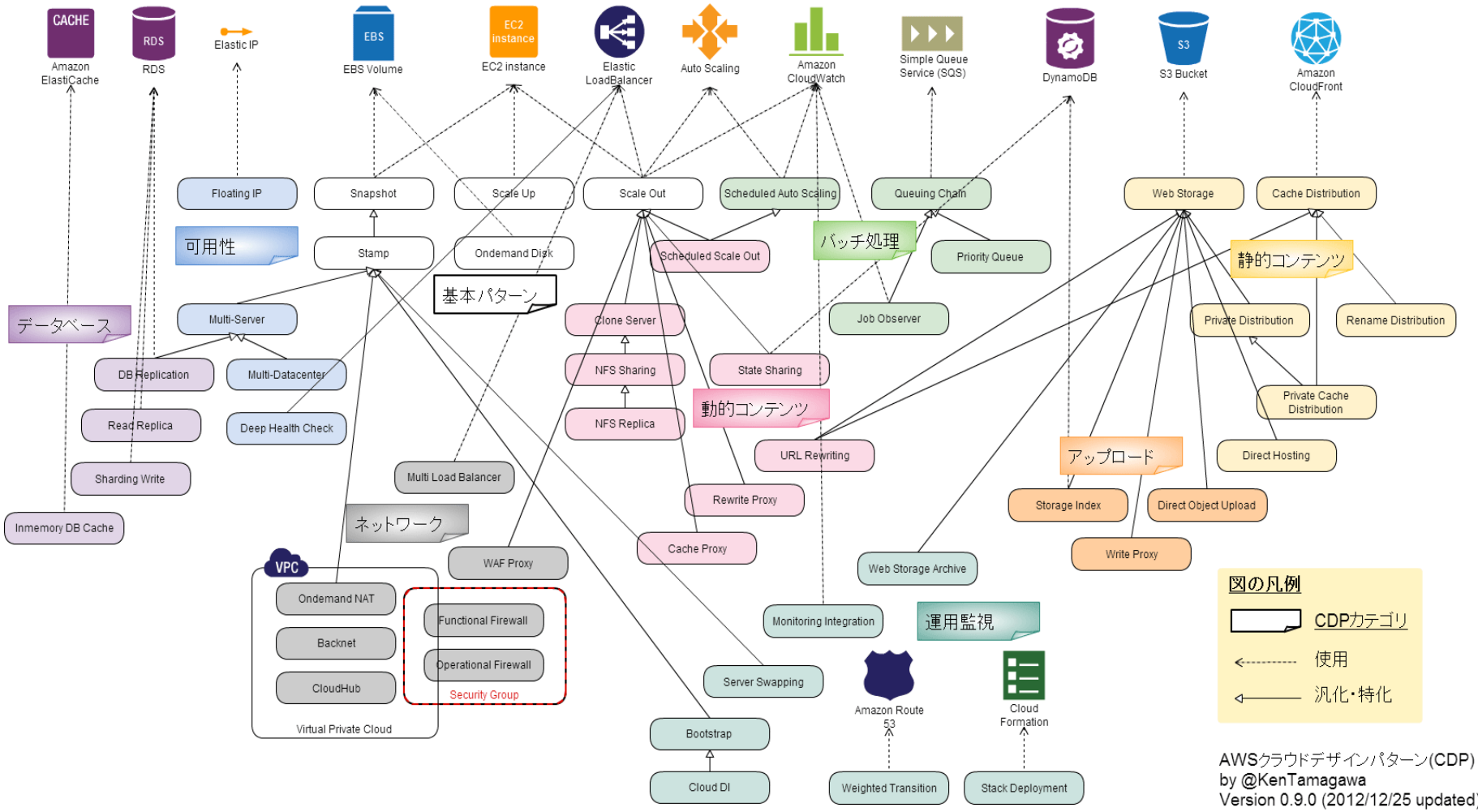
運用保守のパターン

- Bootstrapパターン（起動設定の自動取得）
- Cloud DIパターン（変更が多い部分の外出し）
- Stack Deploymentパターン（サーバ群立ち上げのテンプレート化）
- Server Swappingパターン（サーバの移行）
- Monitoring Integrationパターン（モニタリングツールの一元化）
- Weighted Transitionパターン（重みづけラウンドロビンDNSを使った移行）
- Log Aggregation パターン(ログの集約)
- Ondemand Activationパターン（メンテナンス時の一時的な設定変更）

ネットワークのパターン

- Backnetパターン（管理用ネットワークの設置）
- Functional Firewallパターン（階層的アクセス制限）
- Operational Firewallパターン（機能別アクセス制限）
- Multi Load Balancerパターン（複数ロードバランサの設置）
- WAF Proxyパターン（高価なWeb Application Firewallの効率的な活用）
- CloudHubパターン（VPN拠点の設置）
- Sorry Pageパターン(バックアップサイトへの自動切り替え)
- Self Registrationパターン（自分の情報をデータベースに自動登録）
- RDP Proxyパターン（Windowsインスタンスへのセキュアなアクセス）
- Floating Gatewayパターン（クラウド上のネットワーク環境の切り替え）
- Shared Serviceパターン(システム共通サービスの共用化)
- High Availability NATパターン(冗長化されたNATインスタンス)

CDPの全体像

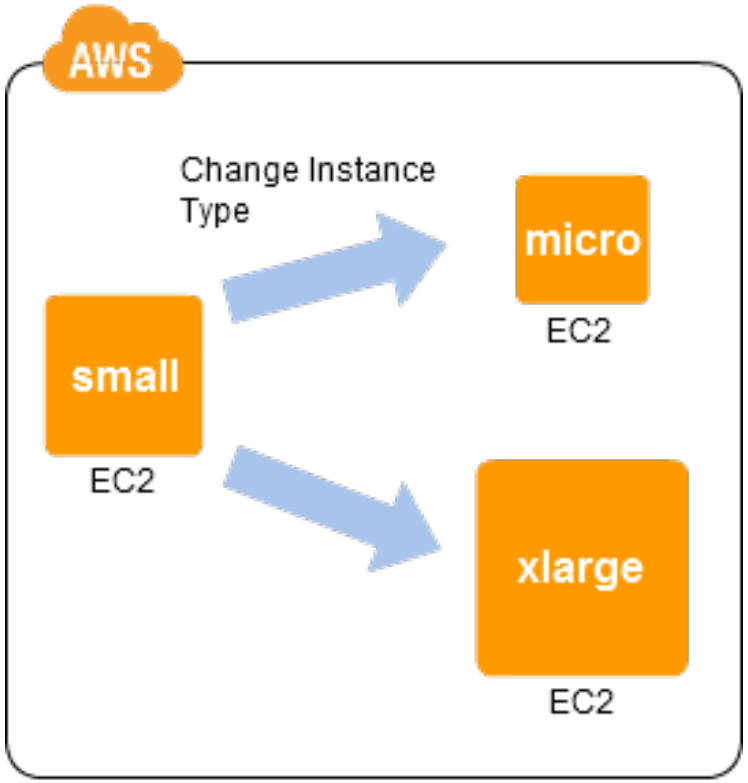

CACHE
Amazon ElastiCache
RDS
RDS
Elastic IP
EBS
EBS Volume
EC2 instance
EC2 instance
Elastic LoadBalancer
Auto Scaling
Amazon CloudWatch
Simple Queue Service (SQS)
DynamoDB
S3
S3 Bucket
Amazon CloudFront
Floating IP
Snapshot
Scale Up
Scale Out
Scheduled Auto Scaling
Queuing Chain
Web Storage
Cache Distribution
可用性
Stamp
Ondemand Disk
Scheduled Scale Out
バッチ処理
Priority Queue
静的コンテンツ
基本パターン
データベース
Multi-Server
Clone Server
Job Observer
Private Distribution
Rename Distribution
DB Replication
Multi-Datacenter
NFS Sharing
State Sharing
Private Cache Distribution
動的コンテンツ
Read Replica
Deep Health Check
NFS Replica
URL Rewriting
アップロード
Direct Hosting
Sharding Write
Multi Load Balancer
Rewrite Proxy
Storage Index
Direct Object Upload
Inmemory DB Cache
ネットワーク
Cache Proxy
Write Proxy
VPC
WAF Proxy
Web Storage Archive
Ondemand NAT
Functional Firewall
Monitoring Integration
運用監視
Backnet
Operational Firewall
Server Swapping
Amazon Route 53
Cloud Formation
CloudHub
Security Group
Virtual Private Cloud
Bootstrap
Cloud DI
Weighted Transition
Stack Deployment
図の凡例
CDPカテゴリ
使用
汎化・特化
AWSクラウドデザインパターン(CDP) 系譜図
by @KenTamagawa
Version 0.9.0 (2012/12/25 updated)

Scale Up

- 解決したい課題：システム稼働後にサーバーリソースの性能をアップしたい

- クラウドでの解決：仮想サーバを使えば起動後でもスペック変更が可能．リソース利用量を確認しながら，随時スペックを変更する

- 実装：EC2でシステムを構築．リソースモニターやCloudWatch等で利用量を確認．必要に応じて，インスタンスタイプを変更

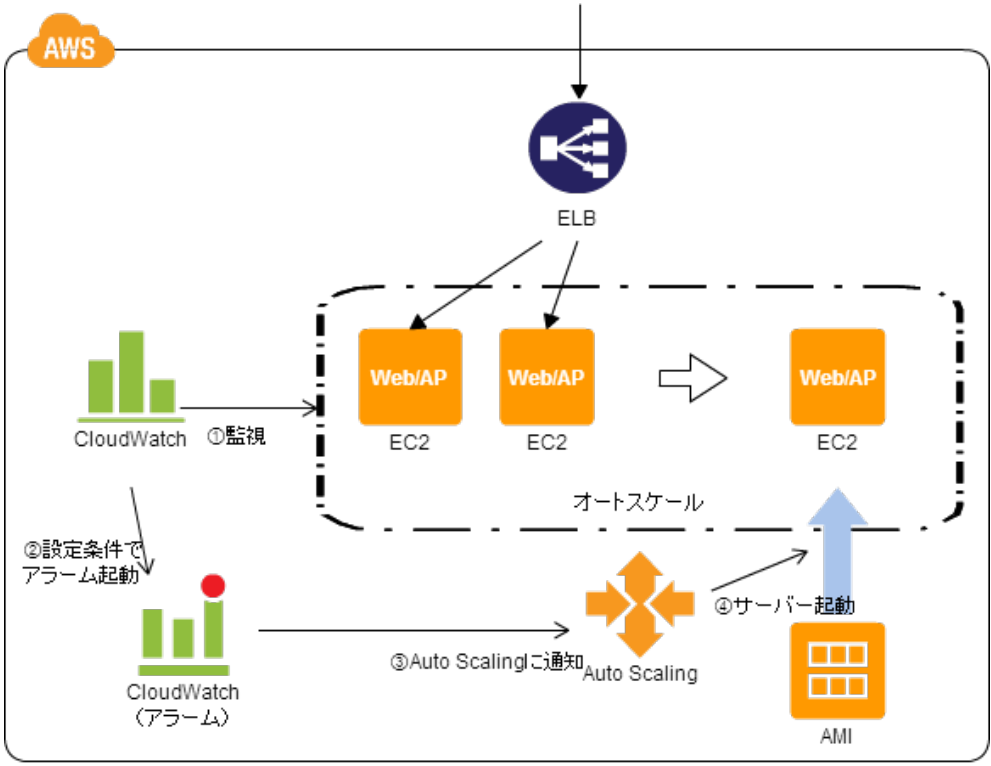


Scale Out

- 解決したい課題：高トラフィックのリクエストを処理するために，高スペックのサーバが必要だが，Scale Upでは限界があり，無制限に高いスペックにはできない．

- クラウドでの解決： 複数の仮想サーバを起動し，ロードバランサーで負荷分散する

- 実装： ELB, CloudWatch, Auto Scalingを組み合わせて容易にスケールアウトシステムを構築できる

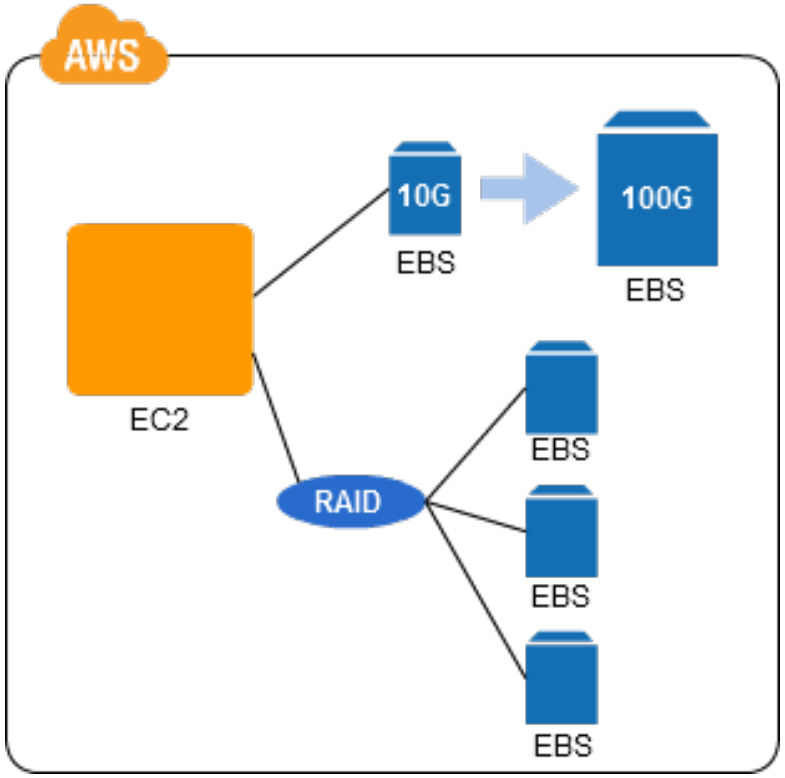


On-demand Disk

- 解決したい課題：システム稼働後に利用するディスクの容量を変更したい．また，ストライピングによるI/O性能を向上したいが，適切なキャパシティがわからない

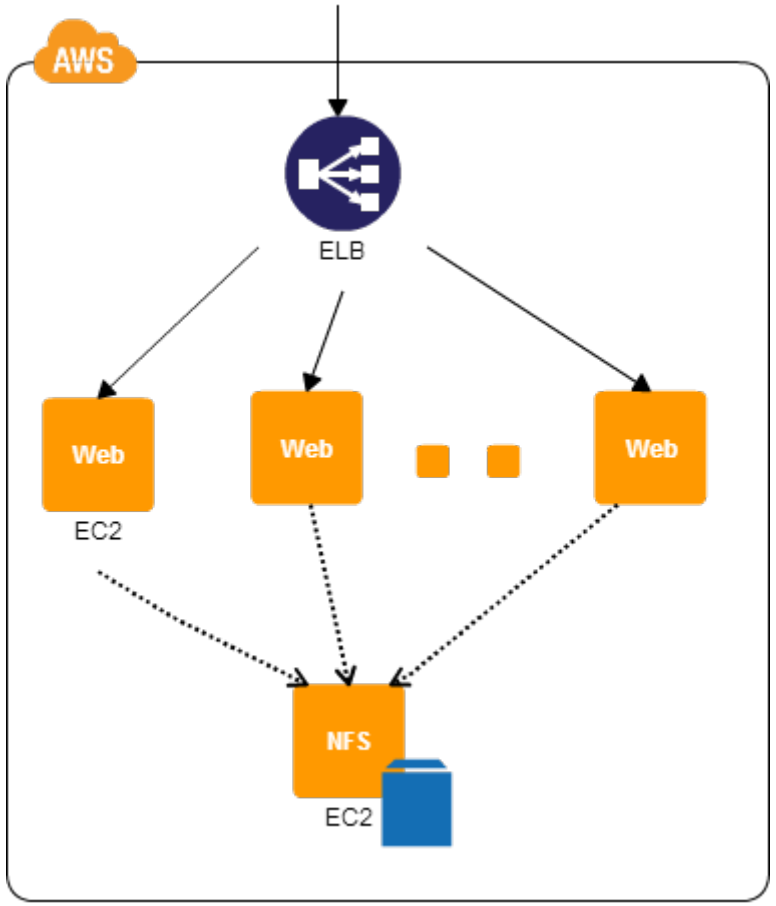
- クラウドでの解決： 仮想ディスクを使えば起動後でも容量変更可能．最初は最小限だけ確保し，利用を確認しながら，随時容量を変更する

- 実装： EBSを使う．EBSスナップショットをとり，新規でより大きなボリュームのEBSを生成してEC2にアタッチする



NFS Sharing

- 解決したい課題：複数サーバで負荷分散した場合，コンテンツを同期させたい．マスタースレーブ方式だと，更新の遅延があるし，スレーブに書き込まれた時に反映できない
- クラウドでの解決： 共有コンテンツを保管する仮想サーバを立ててマスタとし，NFSを使ってWebサーバからマウントする
- 実装： NFSサーバをEC2で構築．スケールアウトするサーバ群からNFSで共有フォルダをマウントする

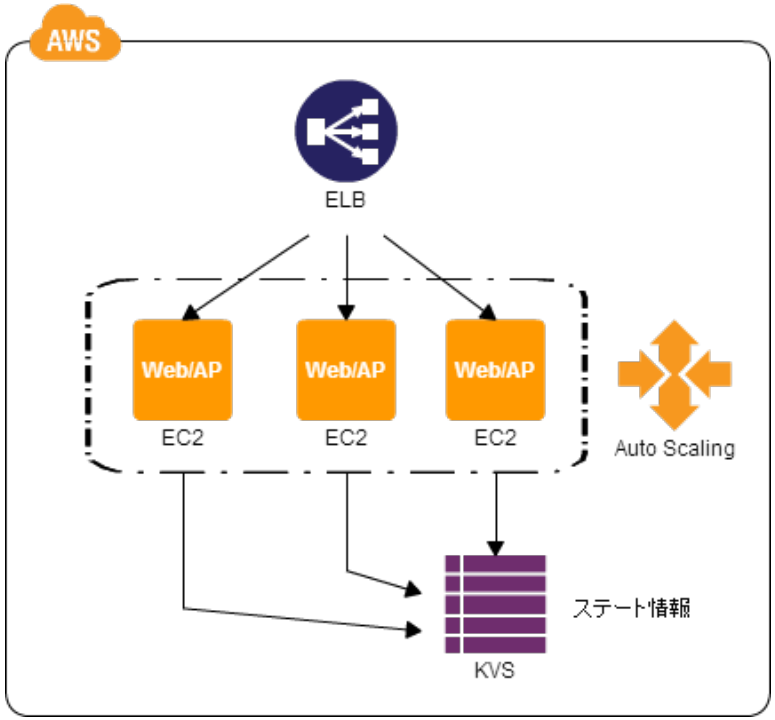


State Sharing

- 解決したい課題：複数サーバで負荷分散した場合，ステート情報（HTTPセッションの情報）をサーバに持たせると，ステート情報の引継ぎ失敗や消失が起こる場合がある

- クラウドでの解決： ステート情報を外部の共有データストアに書き出しサーバから参照するようにする

- 実装： RDSやDynamoDB等のデータストアに，ユーザIDをキー，状態をバリューとして格納する．サーバではステート情報を保管しないようにする．

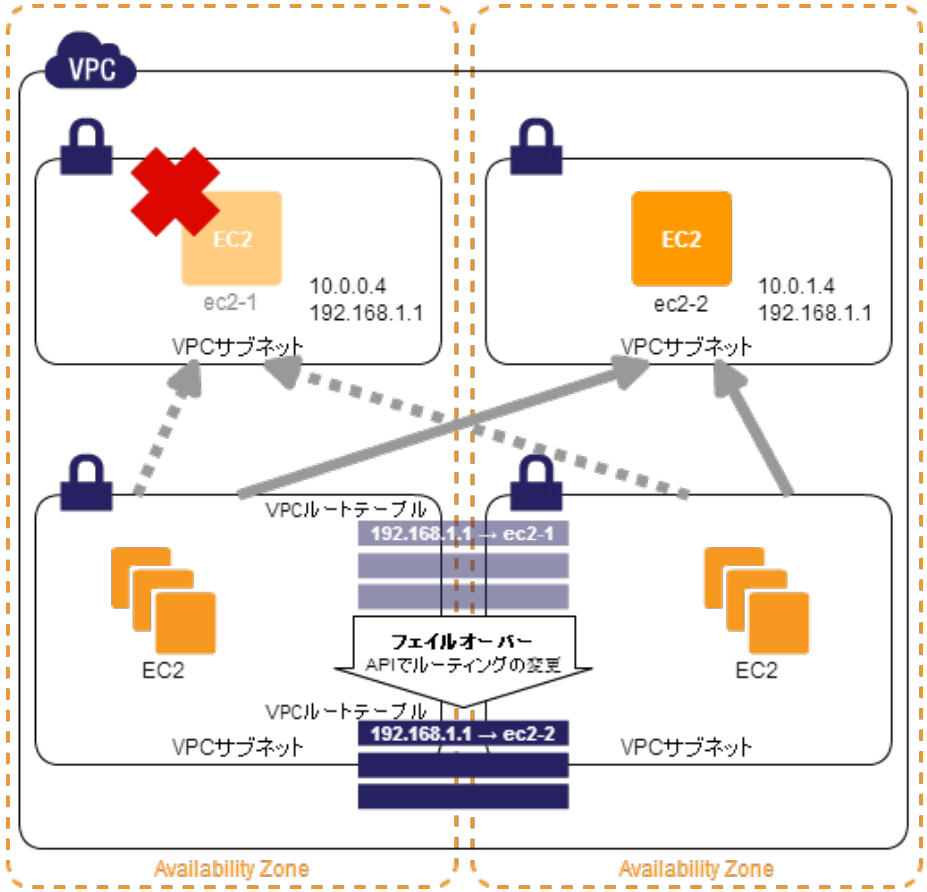


Routing-Based HA

- 解決したい課題：サーバがダウンした際に，異なるセグメントやリージョンにあるサーバに切り替えたい．DNSでのIP振り直しはTTLだけ時間がかかる

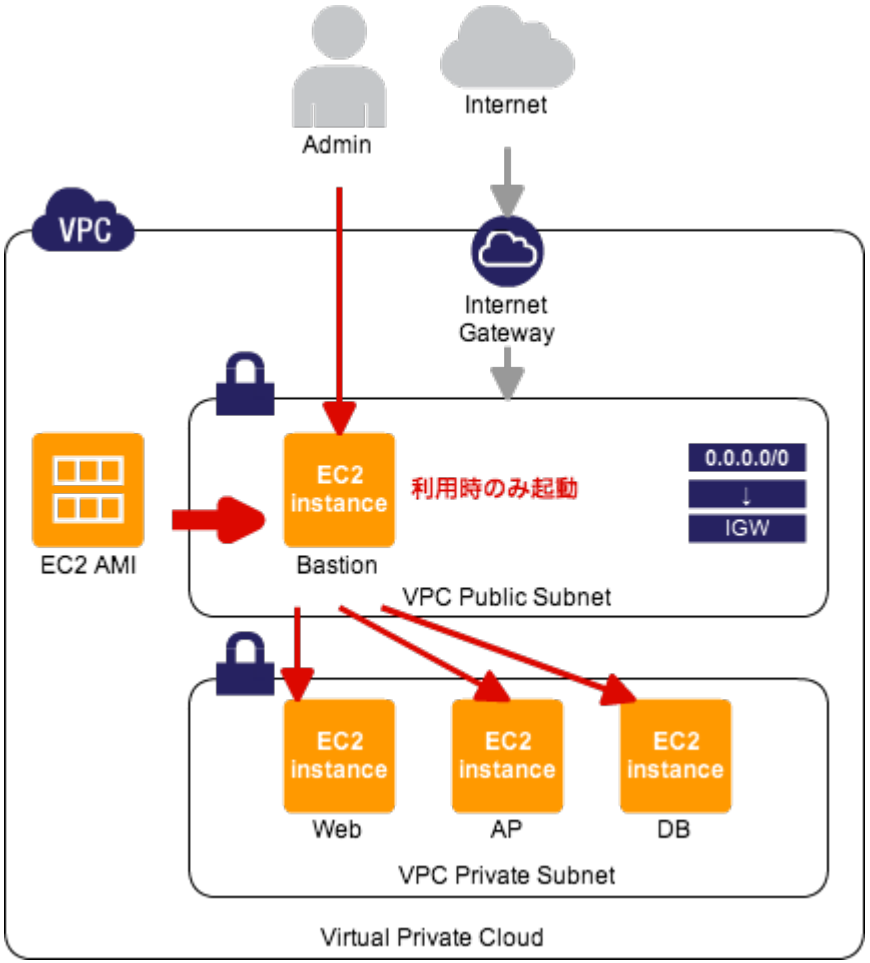
- クラウドでの解決：仮想ネットワークのルートテーブルを書き換えて代替サーバへの経路を確保する

- 実装：冗長化するサーバの仮想IPを固定する．通常はプライマリサーバへの経路を定義しておき，障害時にセカンダリへ経路を変更する

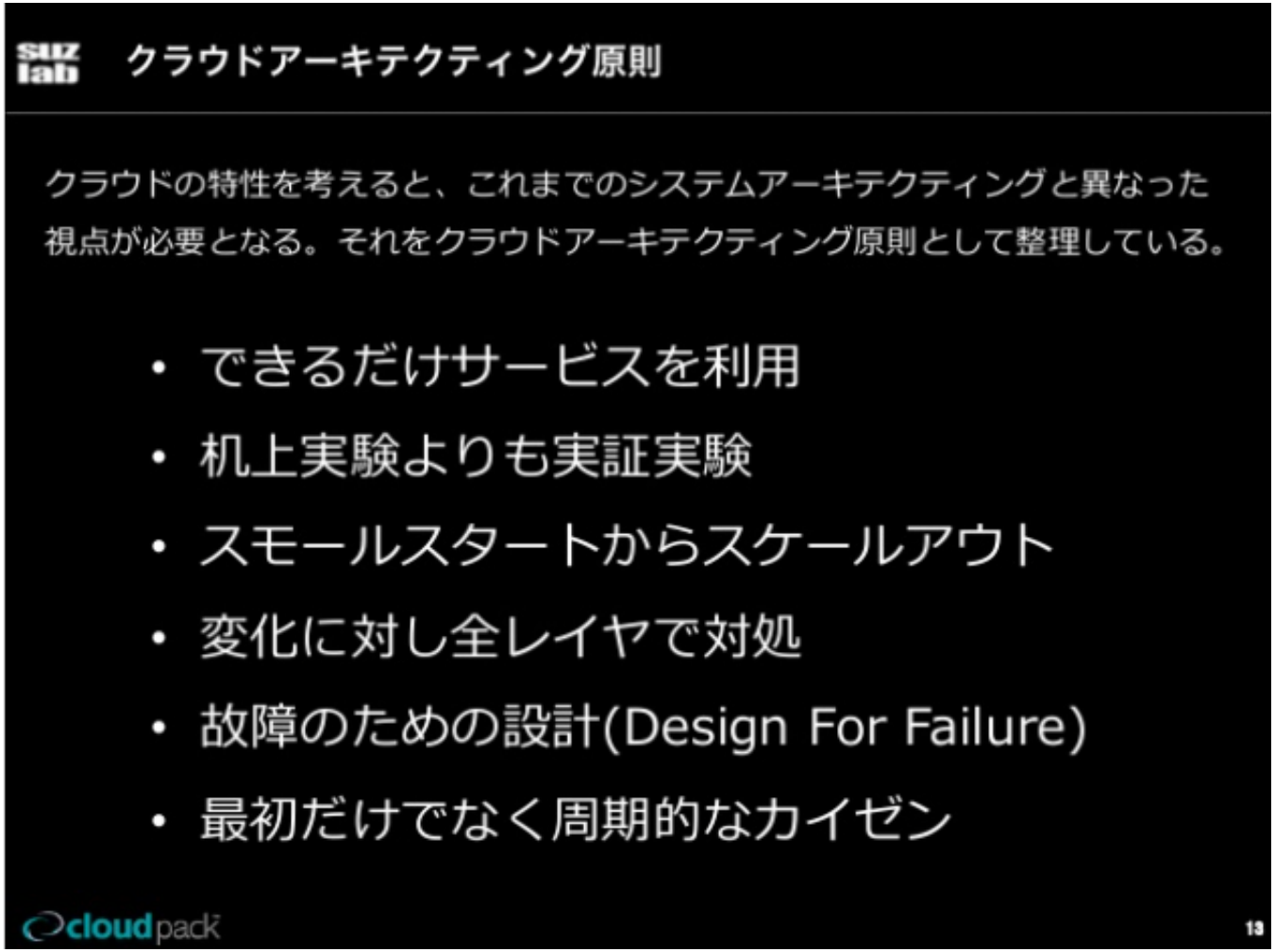


On-demand Bastion (On demand Activationの一つ)

- 解決したい課題：AWS内部に構築した機密サーバ群に対して，踏み台サーバ（Bastion）のセキュリティを高くしたい
- クラウドでの解決：保守・管理作業の間のみ踏み台サーバを起動．利用後に停止する
- 実装：承認プロセスと連動しEC2でBastionのAMIを起動．一定時間後，停止する



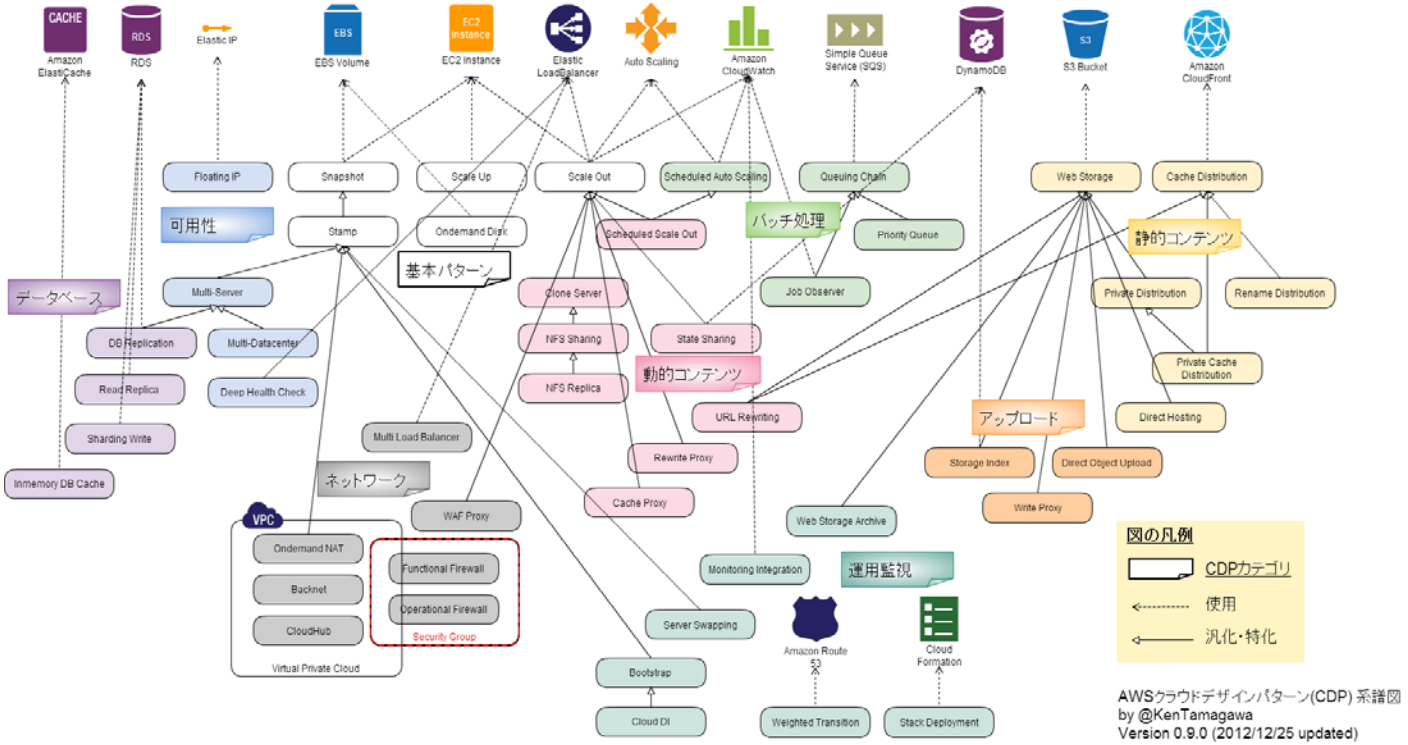
クラウドアーキテクティング原則



鈴木宏康，“実践！AWSクラウドデザインパターン”，
https://www.slideshare.net/suzlab/cdp-28868858

第2部のまとめ

- クラウドデザインパターン
  - AWSシステム開発における典型的なシステム・アーキテクチャのパターンとしてカタログにしたもの
  - クラウドならではの利点や注意点を形式知として学び，共有できる



全体のまとめ

- AWSハンズオンを通して，クラウドコンピューティングの意義や効果を体得した
  - ハンズオン1：EC2でWebサイトを構築
  - ハンズオン2：ロードバランサーでスケールアウト
- AWSを活用した実用的なシステム開発について概観した
  - システムアーキテクチャ，マネージドサービス，開発事例
  - クラウドデザインパターン

- ポイント
  - AWSを活用すれば迅速なサービス開発・公開・運用が可能
    - 小さくやって当たれば大きく育てる．当たらなければやめる
    - 個人レベルでも始められる．高価な設備は不要
  - 付加価値サービスは自分で考える必要がある
    - 技術駆動ではなく目的指向で考える
- つけっぱなしに注意
  - 使わない場合はこまめに止めること

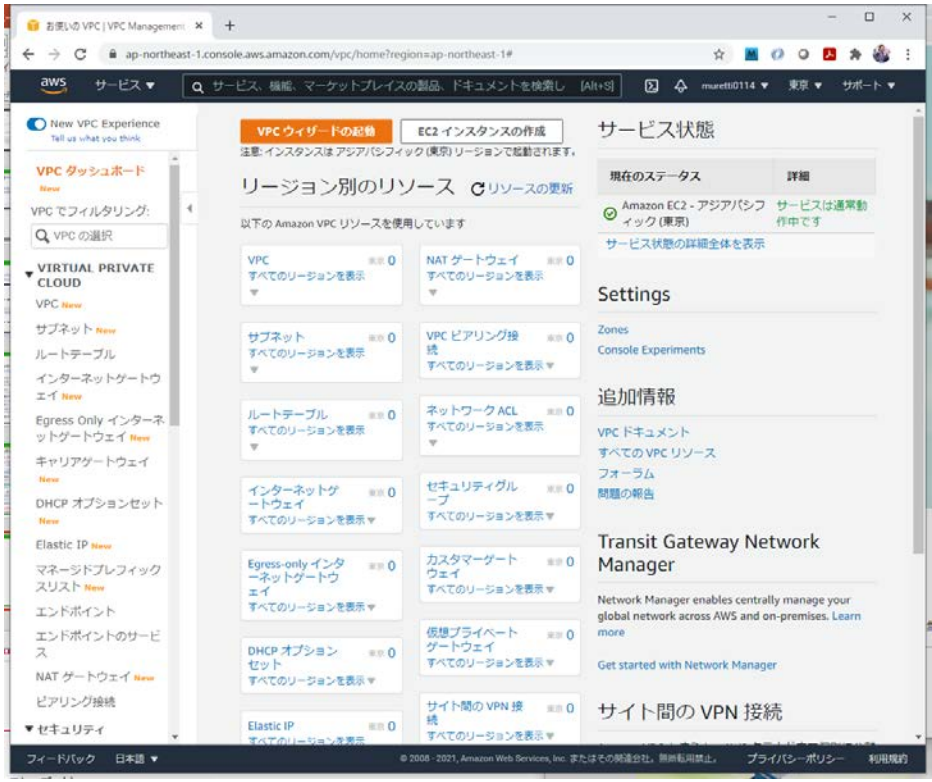
※重要：リソースの後片付けをしよう

- つけっぱなしだと課金されるので以下の手順で片付ける
  - EC2インスタンス：「インスタンスの状態」を変更する
    - 「インスタンスの停止」：サーバを一時停止する．課金がストップする
    - 「インスタンスの終了」：サーバを消去する．再起動不可能になる
  - 停止・終了したらElastic IPを解放（リリース）すること
    - EC2インスタンス実行時は無料．停止・終了した状態では課金される（1時間0.5円）．使わないリソースを占有してしまうためコストが発生する
    - 参考：https://dev.classmethod.jp/articles/cost-of-eip/
  - ロードバランサーは「アクション」→「削除」で削除．一時停止できない
  - VPCは基本的に無料．全リソースを消したい場合は「VPCダッシュボード」から削除
  - スナップショットを削除できない場合には，AMIで使用されていないか確認し，使用されている場合，AMIの登録を解除してから，スナップショットを削除する



ダッシュボードのリソースから確認する

- EC2ダッシュボードおよびVPCダッシュボードからリソースが片付けられたかを確認する



- お疲れさまでした！！