AWS Summit

Tokyo

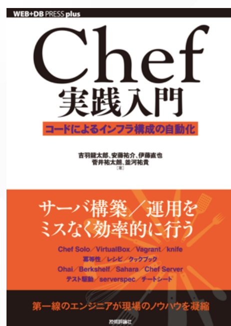
amazon web services

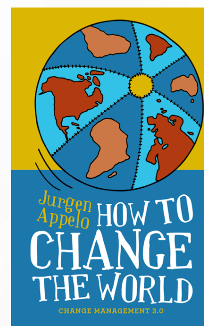
開発生産性をあげるための デプロイ戦略

アマゾン データ サービス ジャパン株式会社
プロフェッショナルサービス本部/部長
吉羽 龍太郎

自己紹介

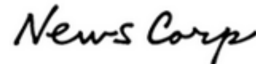
- 吉羽龍太郎
  - アマゾン データ サービス ジャパン株式会社
  - プロフェッショナルサービス本部 部長
  - エンタープライズ企業のクラウド導入やアーキテクティングに関する支援を提供





AWS プロフェッショナルサービスとは？

- お客様のクラウド導入をご支援・加速するための有償コンサルティング/アドバイザリーサービス
- エンタープライズ、政府機関、それらのお客様に従事するSI/ISV様にご提供
- AWSの技術領域に高度に特化
- プロジェクトベースでご支援。期間は短期～1年以上
- タイムアンドマテリアル型で毎月稼働時間分をご請求



デプロイとは？

- あらゆる環境へのコードやバイナリ、アセットなどの配布を**デプロイ**と定義します。
- インフラストラクチャーの構築は**プロビジョニング**と定義します。

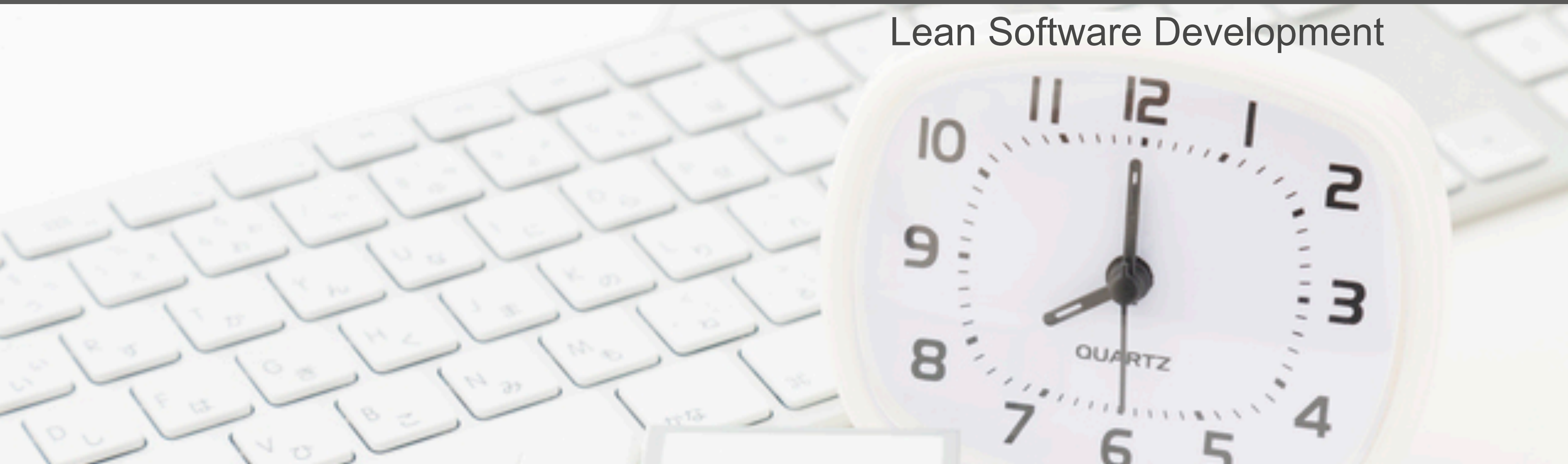
本セッションでは、デプロイにフォーカスしてご説明いたします

amazon web services

会場の皆様に確認

How long does it take to deploy one line of code to production?

Lean Software Development



なぜデプロイに注目する必要があるのか？

amazon web services

AWSのイノベーションのペース

- 合計 1000以上の 新サービス、新機能をリリース
  - 機能追加はAWSが実施
  - 仮想化基盤への適用作業が不要
  - バージョンアップ費用が不要

フィードバックループ

- 顧客とビジネスの間のフィードバックループを加速する
- 顧客の期待に応え続ける

amazon web services

AMAZON.COM
におけるデプロイ

11.6秒	1,079	10,000	30,000
平日の デプロイ間隔	1時間あたりの最高 デプロイ回数	1回のデプロイで 同時に変更をうけ る平均ホスト数	1回のデプロイで 同時に変更をうけ る最高ホスト数

amazon
web services

Two pizza ルール

amazon web services

Two pizza ルール

You build it, you run it.

amazon web services

クロスファンクショナルチーム

	AWS	RSpec	HTML5	Bootstrap	jQuery	Fluentd	Elasticsearch	MongoDB	Ruby	Chef	CircleCI	Go	CoffeeScript
吉羽	○	◎				・		・	◎	△	△		
高田	◎	◎	○	○	・			・	☆	◎	◎	・	
佐藤	○						◎	☆	◎			・	
鈴木	・		◎	◎	◎								◎
高橋	◎	◎				◎	◎	◎		○	○		・
田中	☆	○		△	△				○			◎	・

ポイント

- 多くのチームが非同期にデプロイする必要
- APIによるチーム間の疎結合化
- デプロイ自動化と全チームでの利用

amazon web services

従来型デプロイにおける問題点

問題1：手順書がメンテナンスされない

手順書

Name

リリース手順書_20090306.xlsx
リリース手順書_20100822.xlsx
リリース手順書_20101224_クリスマス対応.xlsx
リリース手順書_20101231_正月対応_org.xlsx
リリース手順書_20101231_正月対応_tanaka.xlsx
リリース手順書_20101231_正月対応.xlsx
リリース手順書_20110503_大規模リリース_レビュー済.xlsx
リリース手順書_20110503_大規模リリース.xlsx

amazon web services

問題2：手作業で間違う

- いままで何度も何度もデプロイ/プロビジョニングしているはず
- 何度か間違ったことありますよね？

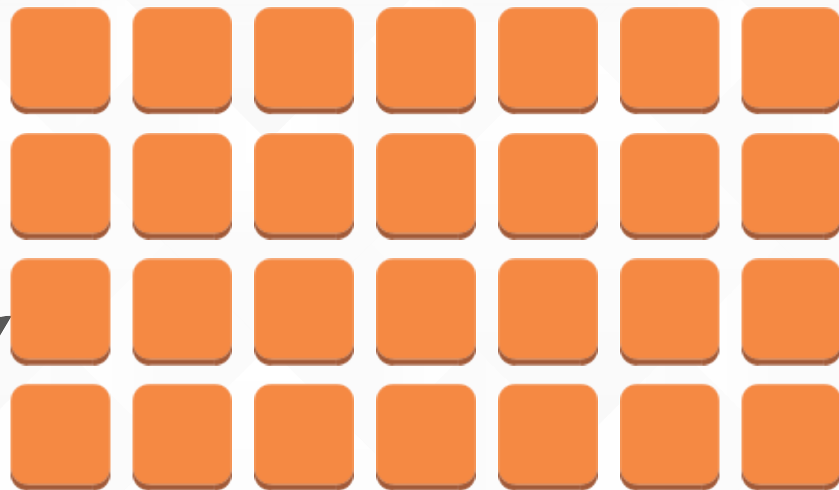
問題3：職人芸への依存

問題4：リリース失敗・戻し失敗

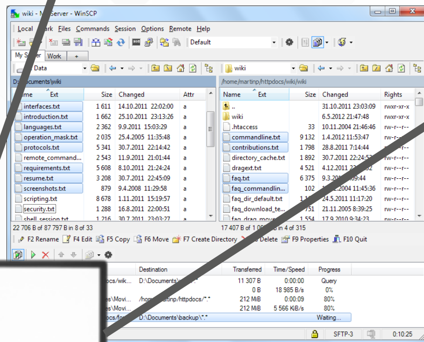
- 手作業で大変なので慎重にレビューが必要
- なので、複数をまとめてリリース
- =ビッグバンリリース
- さらに失敗しやすく
- しかも戻せない

amazon web services

問題5：長時間作業と失敗確率増大



(^_^;)



(ﾟ皿ﾟ)ｷｰｯ!!

amazon web services

問題6：結果としてコスト増大・流出

問題7: リリースにいつも自信を持てない
• 清水の舞台から飛び降りる…
• (そして失敗することも…)

問題8：どんどんプロセスが重くなる

- チェックリストが増える
- トラブルごとに増える
- 二重チェック、三重チェック…
- 一時的に問題が発生しなくなるかもしれないが…
- 持続できない…

TPSの側面から見ると

- Just in Time
- かんばん
- ムダ
- 平準化
- アンドン
- ポカヨケ
- 自働化
- 改善
- 見える化

- **作りすぎのムダ**
- 手待ちのムダ
- **運搬のムダ**
- 加工のムダ
- 在庫のムダ
- **動作のムダ**
- **不良をつくるムダ**

amazon web services

TPSの側面から見ると

- Just in Time
- かんばん
- 自働化
- 改善
- 見える化

- 作りすぎのムダ
- 手待ちのムダ
- 動作のムダ
- 不良をつくるムダ

競争力の源泉となる箇所にフォーカスし ムダを排除する必要がある

amazon web services

デプロイの戦略

デプロイの戦略検討ポイント

いつ？	誰が？	要件は？	ﾌﾟﾛｾｽは？
毎日/週1回/月1回…? 日中/夜間/いつでも?	デプロイ担当者/運用部門/チームの誰でも?	ダウンタイム許容度/グレースフルの必要性/デプロイリスク許容度	承認の必要性は?報告はどのように行うか？

チームサイズ・プロダクト規模・アーキテクチャー

amazon web services

デプロイ自動化の基本原則

- 完全自動化の原則
- 変更量最小化の原則
- 高速完了の原則
- 不可逆変更回避の原則
- 成功・失敗自動判定の原則
- 失敗時ロールバックの原則
- デプロイパターン集約の原則

amazon web services

ベストプラクティス

バージョン管理

テスト自動化

継続的インテグレーション

amazon web services

ベストプラクティス

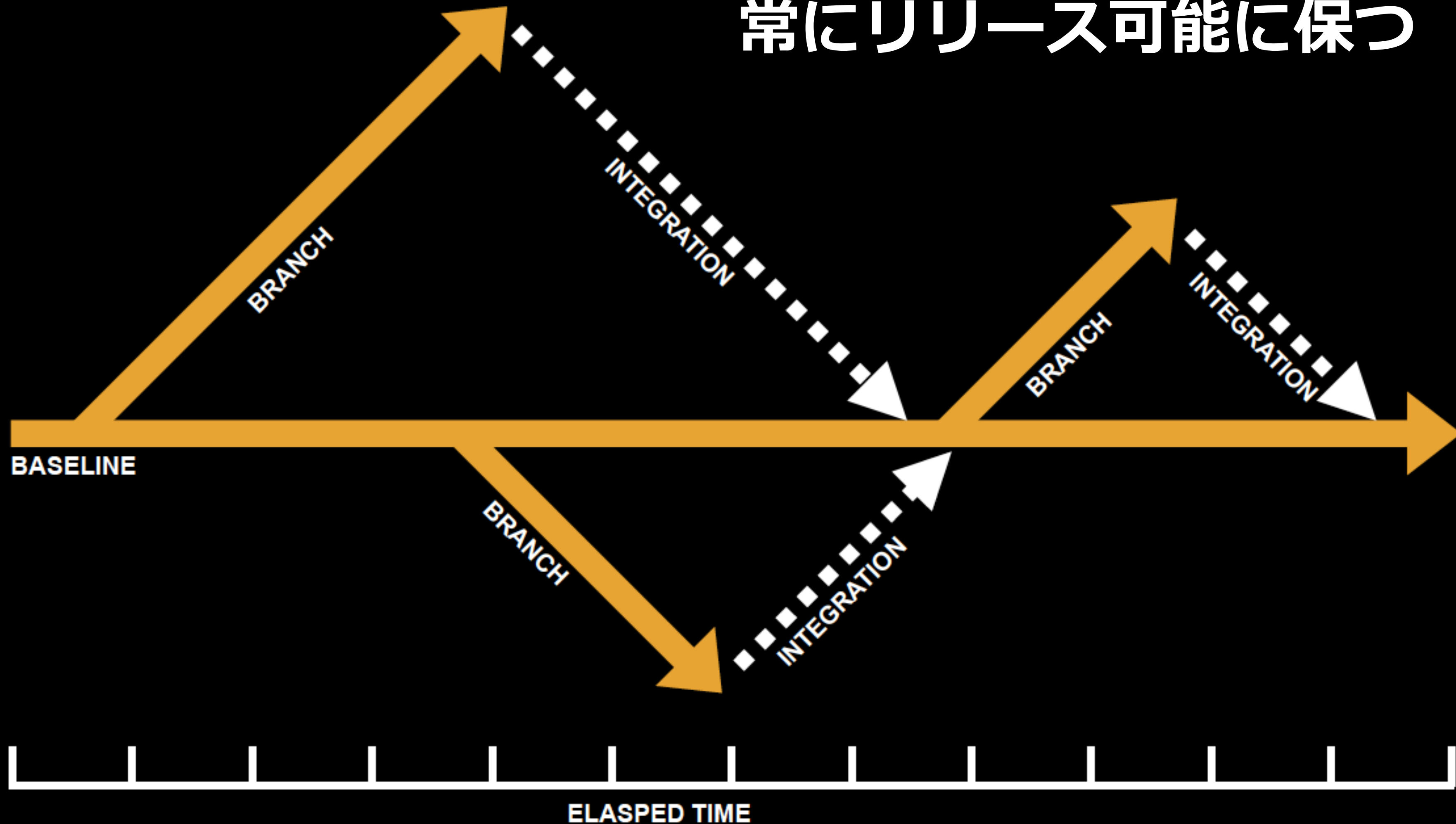
バージョン管理　テスト自動化　継続的インテグレーション

安全な開発のためにプロジェクト初期から

nose

is nicer testing for python

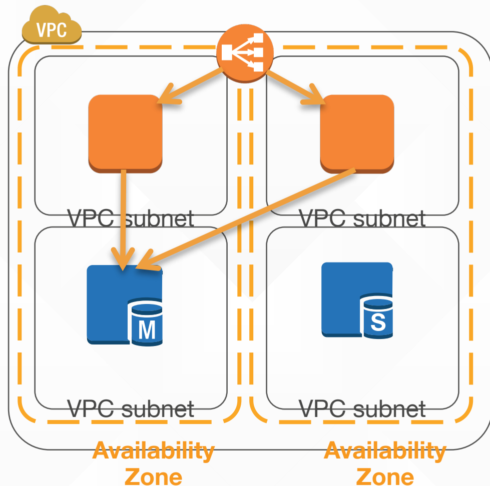
常にリリース可能に保つ



デプロイ自動化の費用対効果

amazon web services

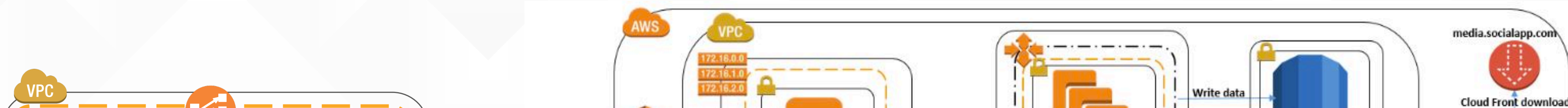
そもそもソフトウェアの複雑化が進行



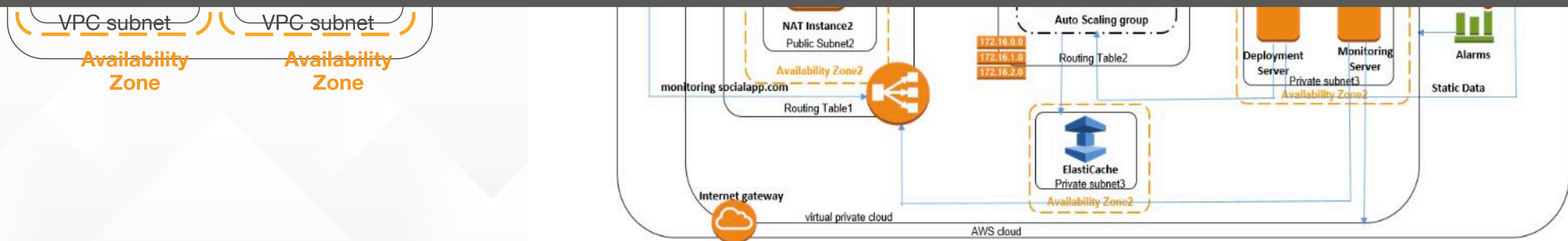
AWS
VPC
172.16.0.0
172.16.1.0
172.16.2.0
users
Hosted DNS Service
NAT Instance1
Public Subnet1
Availability Zone1
socialapp.com
Front-End Load Balancer
Administrators
Bastion/VPN Instance
NAT Instance2
Public Subnet2
Availability Zone2
monitoring socialapp.com
Routing Table1
Internet gateway
virtual private cloud
Web Server Instances
Private subnet1
Availability Zone1
Web Server Instances
Private subnet2
Availability Zone2
Auto Scaling group
Routing Table2
ElastiCache
Private subnet3
Availability Zone2
AWS cloud
Write data
Read Data
RDS Database
Multi-AZ
Replication
RDS Database
Read-Replica
Private DB Subnet Group
Routing Table3
Deployment Server
Monitoring Server
Private subnet3
Availability Zone2
media.socialapp.com
Cloud Front download Distribution
SES Email
Cloud Watch
Alarms
Static Data

amazon web services

そもそもソフトウェアの複雑化が進行



手作業で同時に複数箇所を間違えずに操作することが困難に



amazon web services

デプロイ回数はそもそもとても多いという現実

個人開発環境	共用開発環境	CI環境	スモーク テスト環境
ステージング	性能 テスト環境	セキュリティ テスト環境	受け入れ テスト環境
本番環境	負荷試験環境	パッチ 検証環境	デグレ 検証環境

amazon web services

例えば…

開発チームが5人いて、1日に1回共用開発環境にデプロイ、2日に1回ステージングにデプロイし、チームで5日に1回本番にデプロイするとします。一回のデプロイは手作業で20分かかるとします。

一ヶ月20営業日とすると、

5人*(20回*20分+10回*20分)+4回*20分=約6人日

自動化すればすぐに損益分岐点を交差

amazon web services

例えば…

開発チームが5人いて、1日に1回共用開発環境にデプロイ、
2日に1回ステージングにデプロイし、チームで5日に1回本

本番環境へのデプロイだけでなく
プロセス全体をみて考える

5人*(20回*20分+10回*20分)+4回*20分=約6人日

自動化すればすぐに損益分岐点を交差

amazon web services

その他の潜在的コスト削減

新人が加わった際のデプロイ手順の教育コスト削減

手作業でデプロイを行ない失敗した結果発生する損失の削減

その他の潜在的コスト削減

新人が加わった際のデプロイ手順の教育コスト削減

手作業でデプロイを行ない失敗した結果発生する損失の削減

もしデプロイ自動化に反対されるなら メトリクスを集める

デプロイのパターン

クラウドネイティブアーキテクチャの原則

故障のための設計

- あらゆるものはいつでも故障する前提で設計。単一障害点を避ける
- サーバコピーを取得しいつでも起動可能に
- スナップショットでのバックアップ
- 障害時の自動復旧のためにAuto Scalingを活用
- 複数アベイラビリティゾーンによる冗長化

疎結合なコンポーネント

- コンポーネントを独立させ、各コンポーネントはブラックボックスとして設計
- コンポーネント間を疎結合に
- アプリケーションをステートレスに。この実現のためにELBを活用し、スティッキーセッションを避ける
- セッションデータは各サーバではなくデータベース(RDS)に保存

弾力性の実装

- コンポーネントの健全性や配置場所を決めつけない
- いつでもリブート可能になるように設計
- 動的なコンフィグレーション(起動時に自動でアプリケーションやライブラリ等をインストールする)の活用
- 設定済みAMIの活用による時間短縮

全レイヤでセキュリティ担保

- AWSのセキュリティは責任分担モデル。OS以上の層やセキュリティグループの設定、データ暗号化、秘密鍵の管理等は利用者側の責任
- OSの層とアプリの層、ネットワークの層など各所でセキュリティを担保すること。必要に応じてアプリケーションのペネトレーション試験も推奨

並列処理の実装

- AWSの特性は時間単位でリソースを使用可能な点。これを活かしてアプリケーションやバッチ処理の並列化、マルチスレッド化を検討する
- ELBやAutoScalingを活用して並列処理、分散処理を実装

異なるストレージの使い分け

- EBS(仮想マシンで利用するディスク)や、データベース、S3(インターネットストレージ)等データを保存するストレージには多種多様なものがある
- 読み書きの特性(読み書きの比率や頻度)や、データ喪失の可能性、ステートレスの実現可否を踏まえて適切なストレージを選択する

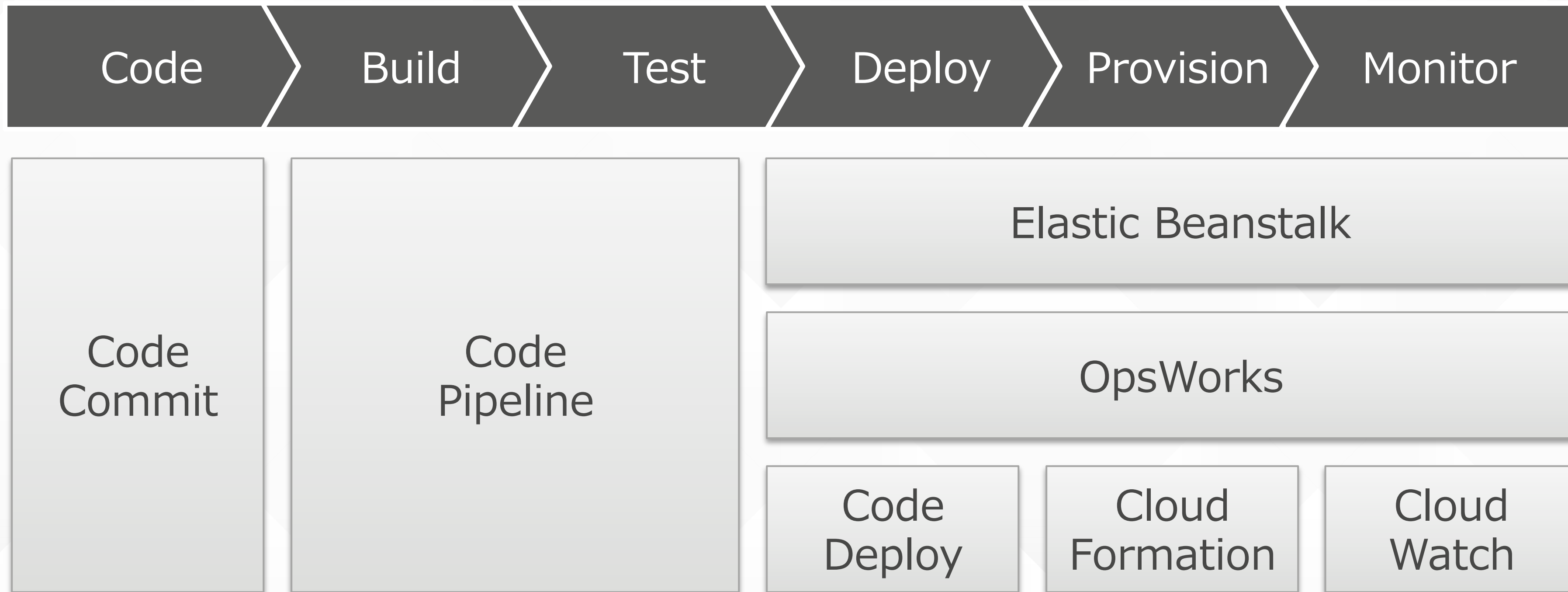
amazon web services

パターン

- 単純デプロイ
- サービス停止＆デプロイ
- 読み取り専用化＆デプロイ
- 機能縮退化＆デプロイ
- 半数切り離し＆デプロイ
- ブルーグリーンデプロイ
- カナリアリリース

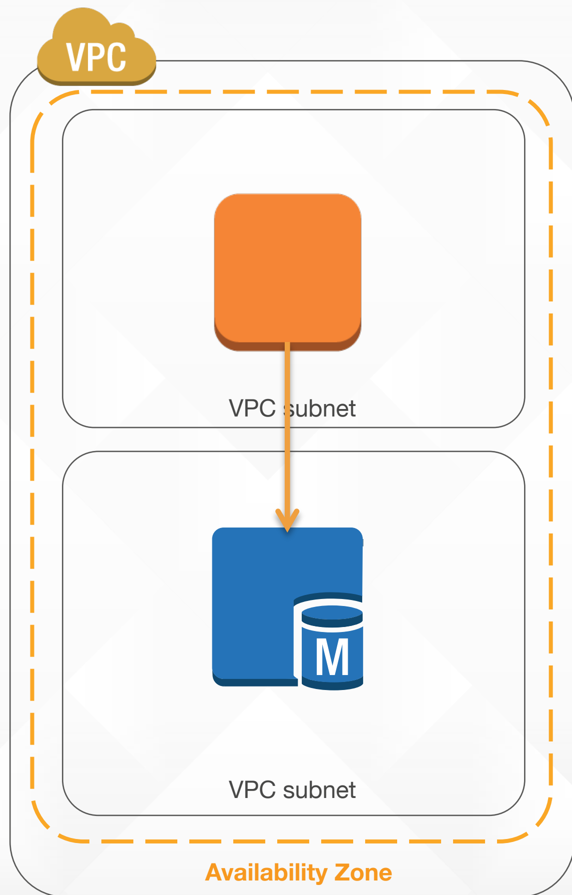
amazon web services

AWSにおける開発系サービス概観



amazon web services

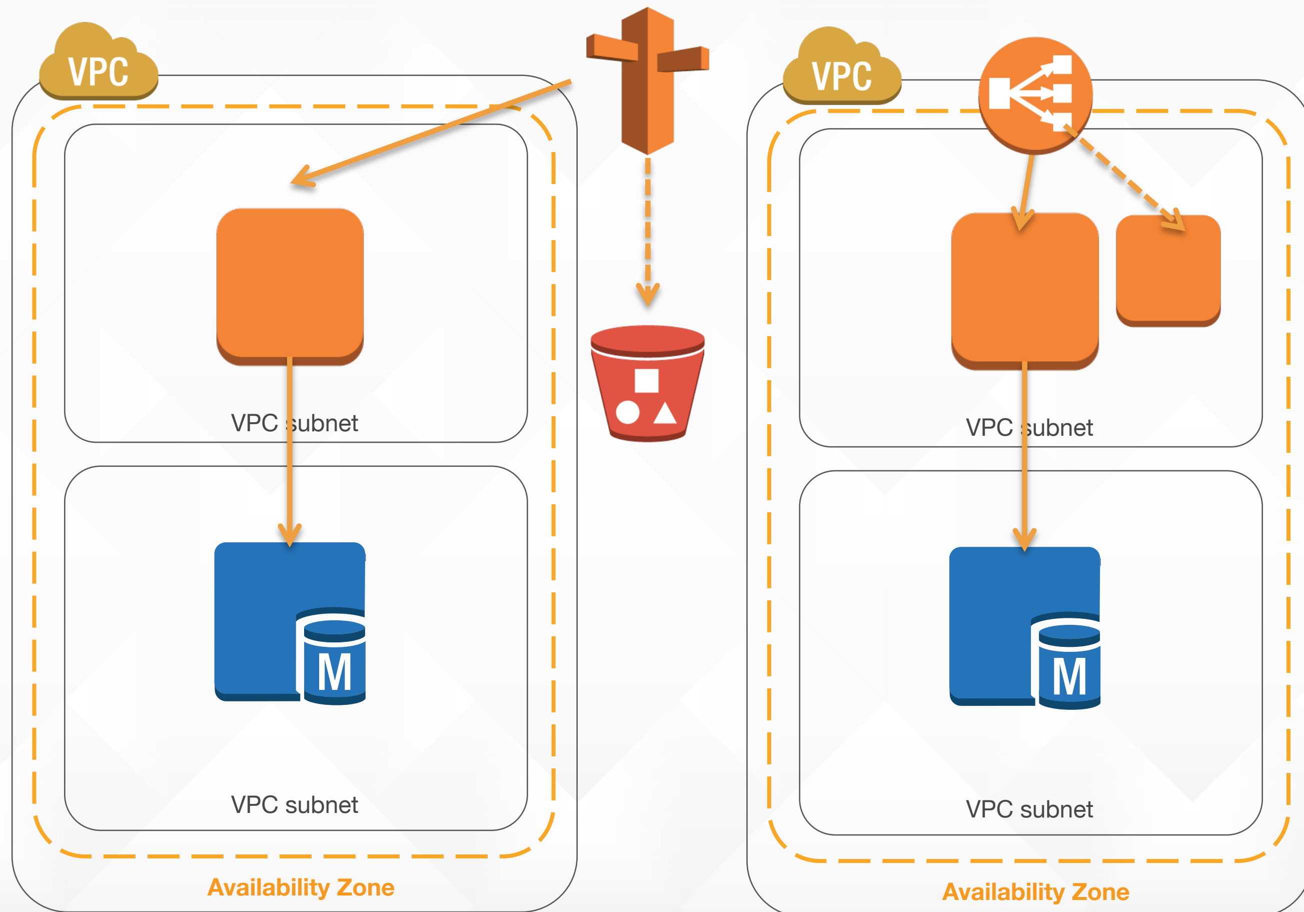
構成1：シンプルなWeb系システム



- 非常にシンプルな構成
- 可用性の考慮なし。デプロイ失敗はサービスの停止
- デプロイの選択肢は多数あり
  - 手作業
  - デプロイツール、rsyncなど
  - CodeDeploy
  - Elastic Beanstalk

amazon web services

(参考) Sorryページへの切り替えパターン

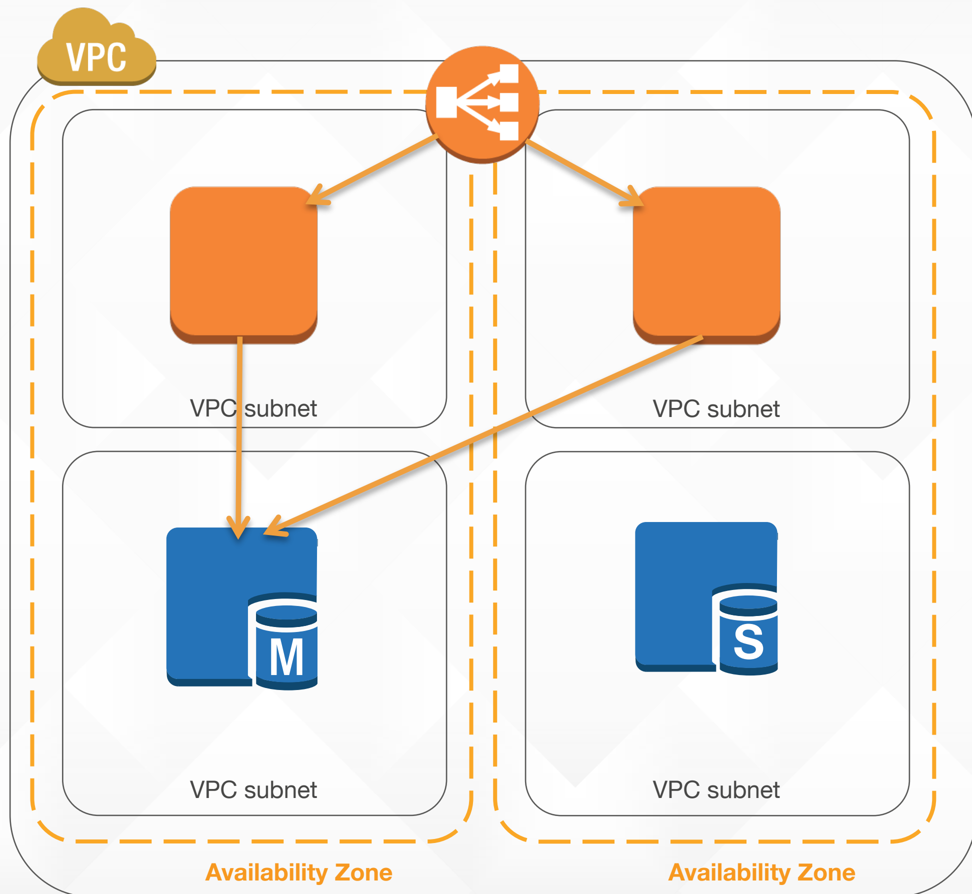


DNSフェイルオーバーまたはSorryサーバへの切り替えによって、一時的にメンテナンスモードに変更してデプロイ。

前者はTTLやDNSキャッシュの問題があるため後者推奨

amazon web services

構成2：冗長化されたWeb系システム



- デプロイの選択肢
  - デプロイツール
  - CodeDeploy
  - Elastic Beanstalk
- 検討ポイント
  - 1台ずつLBから外してデプロイ・確認・投入するか、それとも、一気に反映するか？

amazon web services

(参考)ELBのConnection Draining

バックエンドのEC2インスタンスをELBから登録解除したり、ヘルスチェックが失敗した時に、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。

- 新しく作成したELBではデフォルトで有効、タイムアウト 300秒。
- タイムアウト最大 3600秒

接続が途中で切れることを防げ、シームレスなデプロイが可能に!!

amazon web services

処理フローの互換性の確保

VPC

V1.0

V1.1

VPC subnet

VPC subnet

M

S

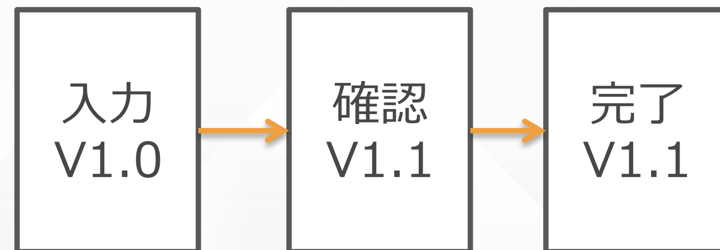
VPC subnet

VPC subnet

Availability Zone

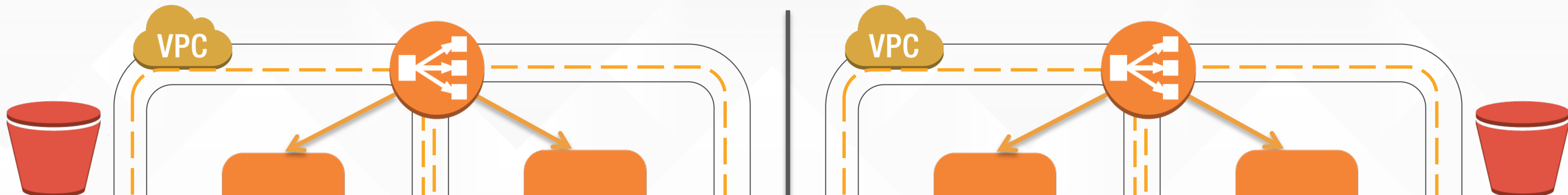
Availability Zone

- 複数のシーケンスからなる一連の処理が存在し、無停止デプロイする場合、アプリケーションデプロイのタイミングで、複数バージョンにまたがる処理が発生する可能性がある。

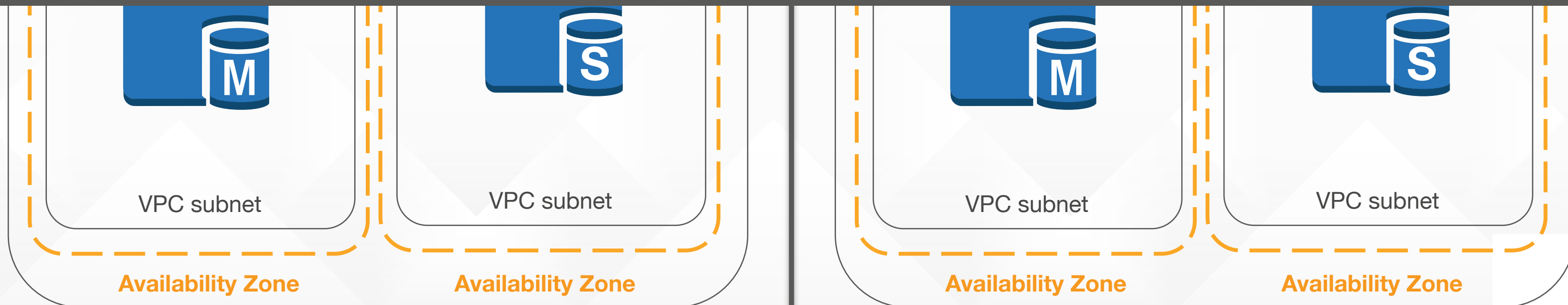


amazon web services

開発環境と本番環境の相違点の管理



相違点をどう管理するかを設計すること



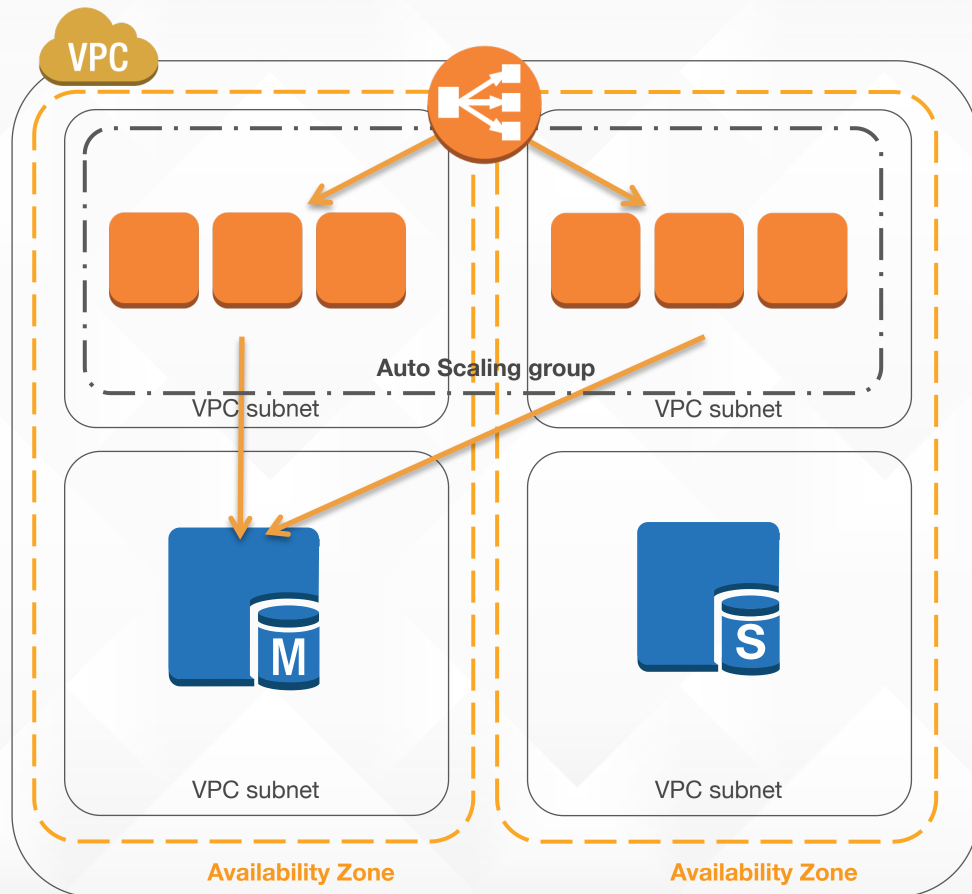
環境別の相違点の管理方法

- 接続先DB、ユーザ名、パスワード、S3のバケット名、DynamoDBやSQSのエンドポイントなどは環境(開発/ステージング/本番など)によって変化する
- 管理の方法
  - ①環境変数
  - ②設定ファイル

ハードコーディング、AMIへの埋め込みは避ける!!

amazon web services

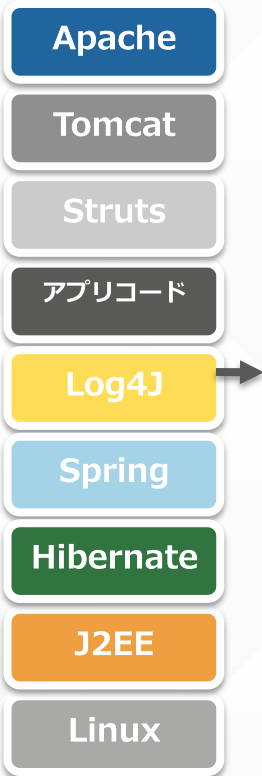
構成3：伸縮するWeb系システム



- Auto Scaling発動時には最新のアプリケーションが自動でデプロイされるようにする必要
- 稼働中のインスタンスへのアプリデプロイをどうするか？
- AMIをどこまで作りこむか？
- インスタンス起動時のcloud-initの活用
- デプロイの選択肢
  - Auto Scaling + cloud-init
  - Elastic Beanstalk
  - CodeDeploy

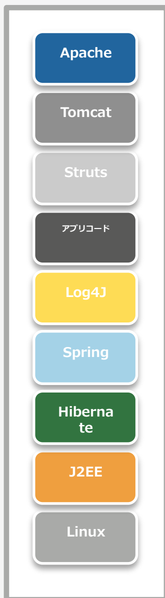
amazon web services

どんなAMIを使うか？



Javaスタック

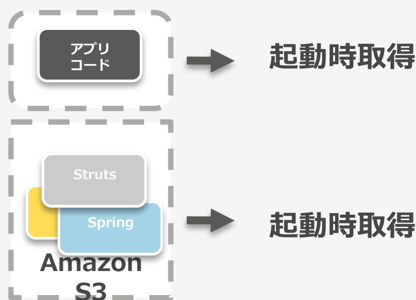
[1]全部入りAMI

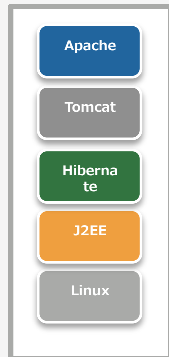


Java AMI

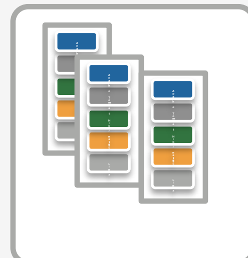
EC2

[2]Golden Image



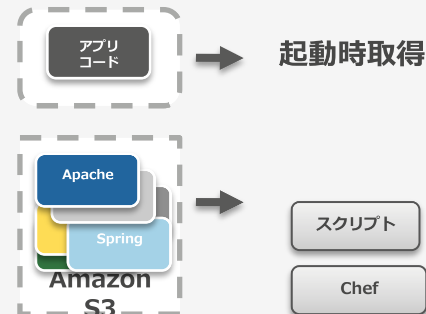


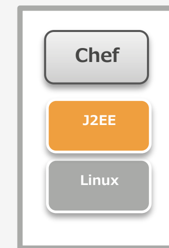
Java AMI



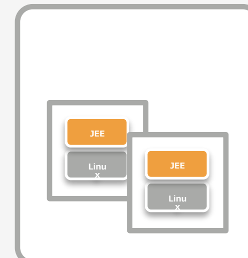
EC2

[3]最小構成AMI





AMI



EC2

AMIによるデプロイのPros/Cons

方式	利点	欠点
[1]全部入りAMI	同一性の確保	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]GoldenImage	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	自由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

amazon web services

AMI自動作成例

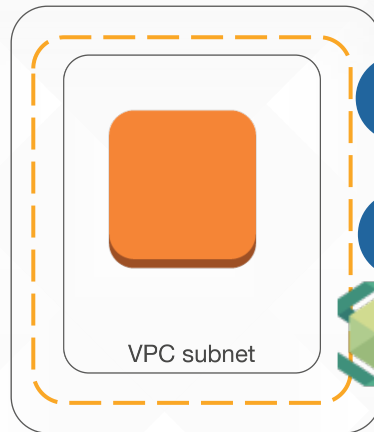
Jenkins

1 チェックアウト

1 チェックアウト

```
{
  "builders": [{
    "type": "amazon-ebs",
    "region": "ap-northeast-1",
    "source_ami": "ami-39b23d38",
    "instance_type": ”m3.large",
    "ssh_username": "ec2-user",
    "ssh_timeout": "5m",
    "ami_name": "amznlinux-{{timestamp}}"
  }],
  "provisioners": [{
    "type": "chef-solo",
    "cookbook_paths": ["./vendor-
cookbooks/"],
    "run_list": ["timezone", "apache2",
"memcached"],
    "json": {"memcached":
{"maxcon":"512","cachesize":"512"}},
    "prevent_sudo": false,
    "skip_install": false
  }]
}
```

VPC



3 EC2インスタンス起動

4 Serverspecでテスト

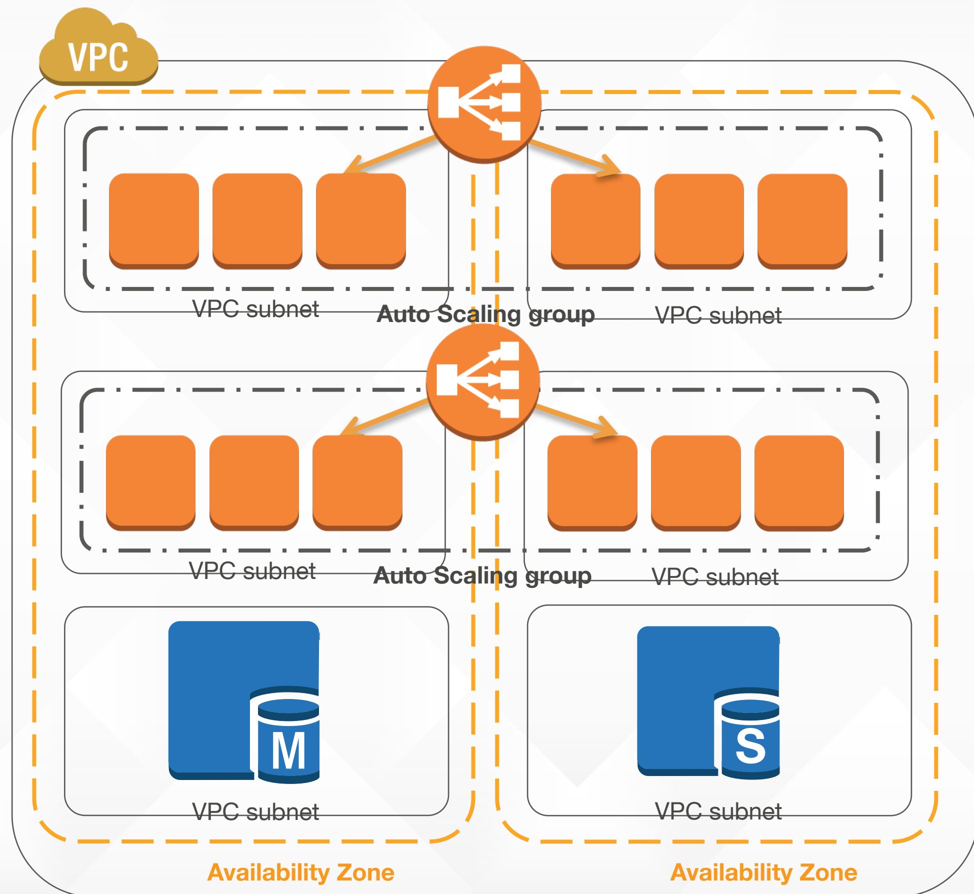
PACKER

2 Packer起動

5 AMI化

amazon web services

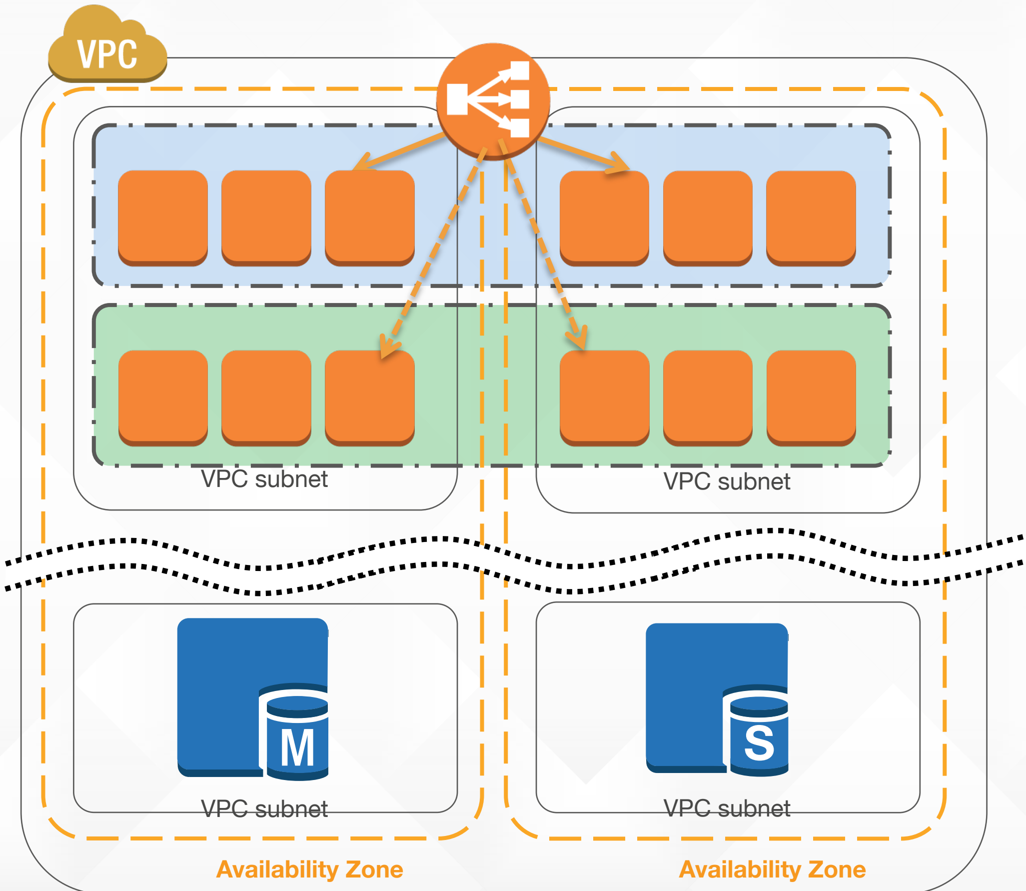
構成4：伸縮+3層Web系システム



- デプロイの選択肢は前ページの構成と大きくは変わらない
- 注意すべき点として、各層のデプロイをどのように同期させるかは、サービス停止が許容できない場合課題になる

amazon web services

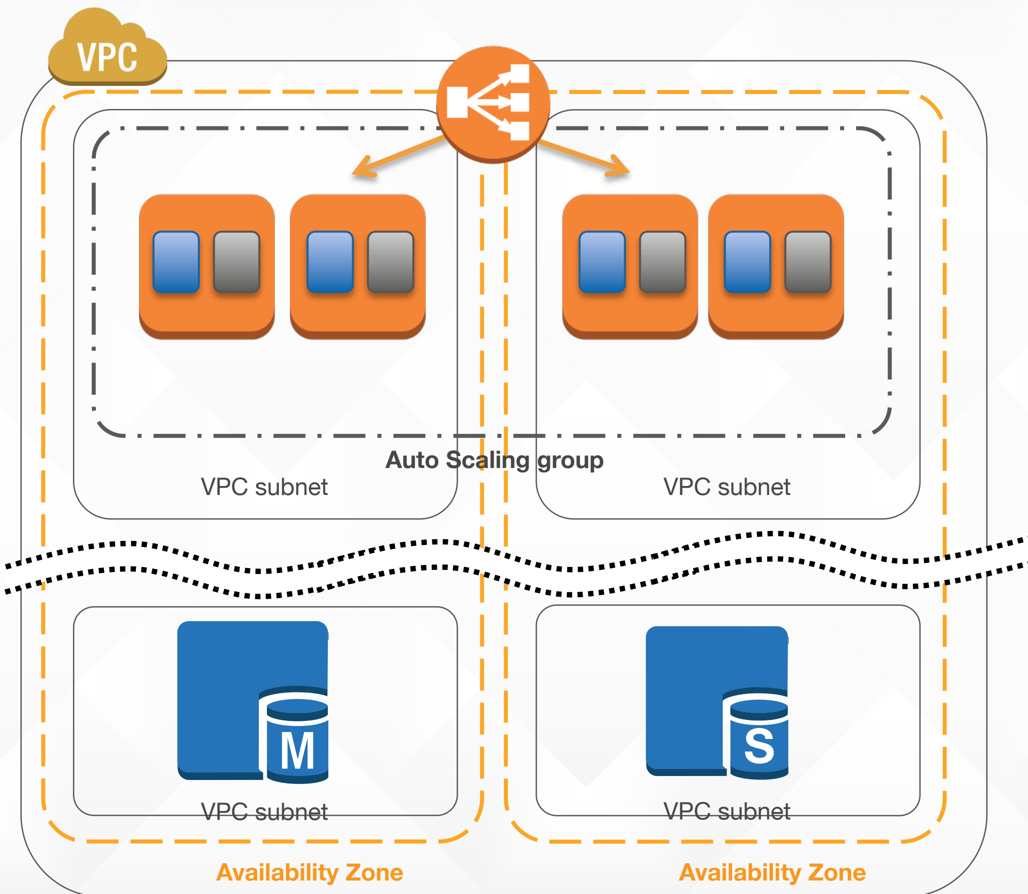
構成5：伸縮+3層Web系システム(ブルーグリーン)



- A面、B面を用意しておき、リリース時にELBの設定を変更してデプロイを完了させる。
- デプロイが完了し問題ないことが確認できれば片面を停止したり破棄することでコストを圧縮できる。

amazon web services

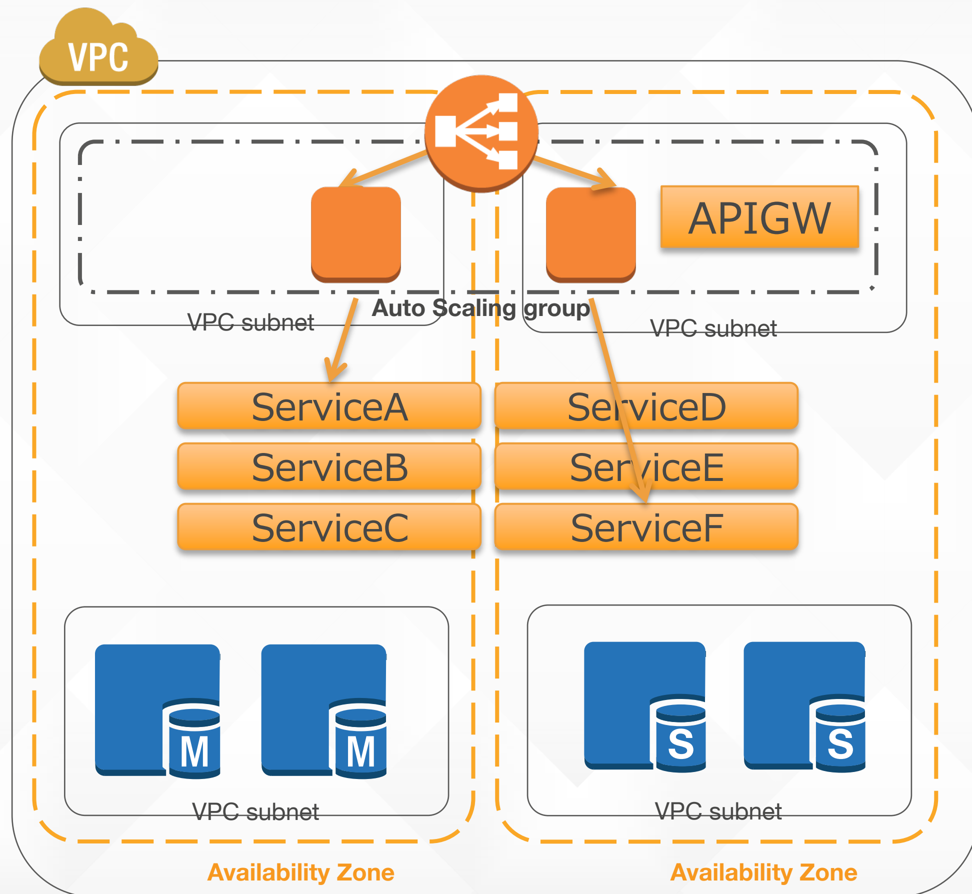
構成6：伸縮+3層Web系システム(コンテナ)



- Dockerを利用したデプロイ
- デプロイの際に、新バージョンのコンテナを動作させ、ポートマッピングを変更することでデプロイする。
- ECSや他のDockerクラスタ化ソリューションを利用することで実現

amazon web services

構成7：マイクロサービス



- モノリシック(一枚板)な単一巨大システムから、単一の役割を持つ複数のサービスに分離する構成
- 各サービスはインターフェイスさえ同一であればブラックボックス化できるため、疎結合
- サービス内の構成はここまで見てきた構成パターンのいずれかになる

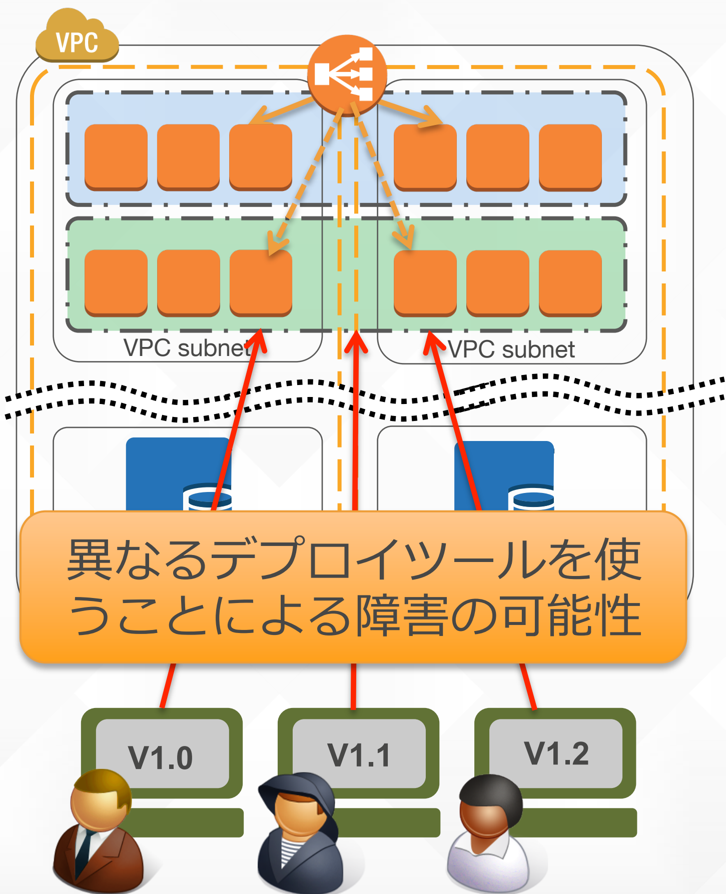
amazon web services

マイクロサービス化の際に注意すべき点

- サービスの後方互換性を確保する
  - かならずしもサービスを同じタイミングでデプロイできるとは限らない
  - すなわちインターフェイスが変わる場合は、新・旧が併存する期間を設ける。例：/api/v1/ と /api/v2/
- 性能やエラーレートの監視とスロットリング
  - いつ・どこから呼ばれるかのコミュニケーションが必要になるとマイクロサービス化の利点が失われる

amazon web services

デプロイツールのデプロイ問題の解消



VPC

VPC subnet

VPC subnet

Chat

CI

M

S

VPC subnet

VPC subnet

Availability Zone

Availability Zone

amazon web services

Thank You

amazon web services