AWS 設計の ベストプラクティスで最低限 知っておくべき 10 (+1) のこと

アマゾン ウェブ サービス ジャパン株式会社
ソリューションアーキテクト
大場　崇令 (Takanori Ohba)
2020年6月17日

Who am I ?

@takaohba

- 大場 崇令（オオバ タカノリ）
  - Partner Solutions Architect @Amazon Web Services Japan K.K. (Joined 2015/12)

- Background
  - AWS テクニカルトレーナー@AWSJ K.K.
  - Web サービスのインフラエンジニア
  - 国内クラウドベンダーにてテクニカルサポート

- 好きな AWS サービス
  - AWS Well-Architected Tool
  - AWS Systems Manager
  - AWS Service Catalog

aws certified Solutions Architect Professional
aws certified DevOps Engineer Professional
aws certified Solutions Architect Associate
aws certified SysOps Administrator Associate
aws certified Developer Associate
aws certified Cloud Practitioner

Well-Architected Lead

アジェンダ

- 本セッションの目的
- AWS 設計のベストプラクティスで最低限知っておくべき 10 (+1) のこと
    1. スケーラビリティを確保する
    2. 環境を自動化する
    3. 使い捨て可能なリソースを使用する
    4. コンポーネントを疎結合にする
    5. サーバーではなくサービスで設計する
    6. 適切なデータベースソリューションを選択する
    7. 単一障害点を排除する
    8. コストを最適化する
    9. キャッシュを使用する
    10. すべてのレイヤーでセキュリティを確保する
    11. 増加するデータの管理
- まとめ

本セッションの目的

対象者

- クラウドの導入を検討されている方
- オンプレミスからクラウドへの移行を検討されている方
- クラウドを使い始めたものの、正しくベストプラクティスを適用できているか悩まれている方

目的

- AWS における設計のベストプラクティスを理解する

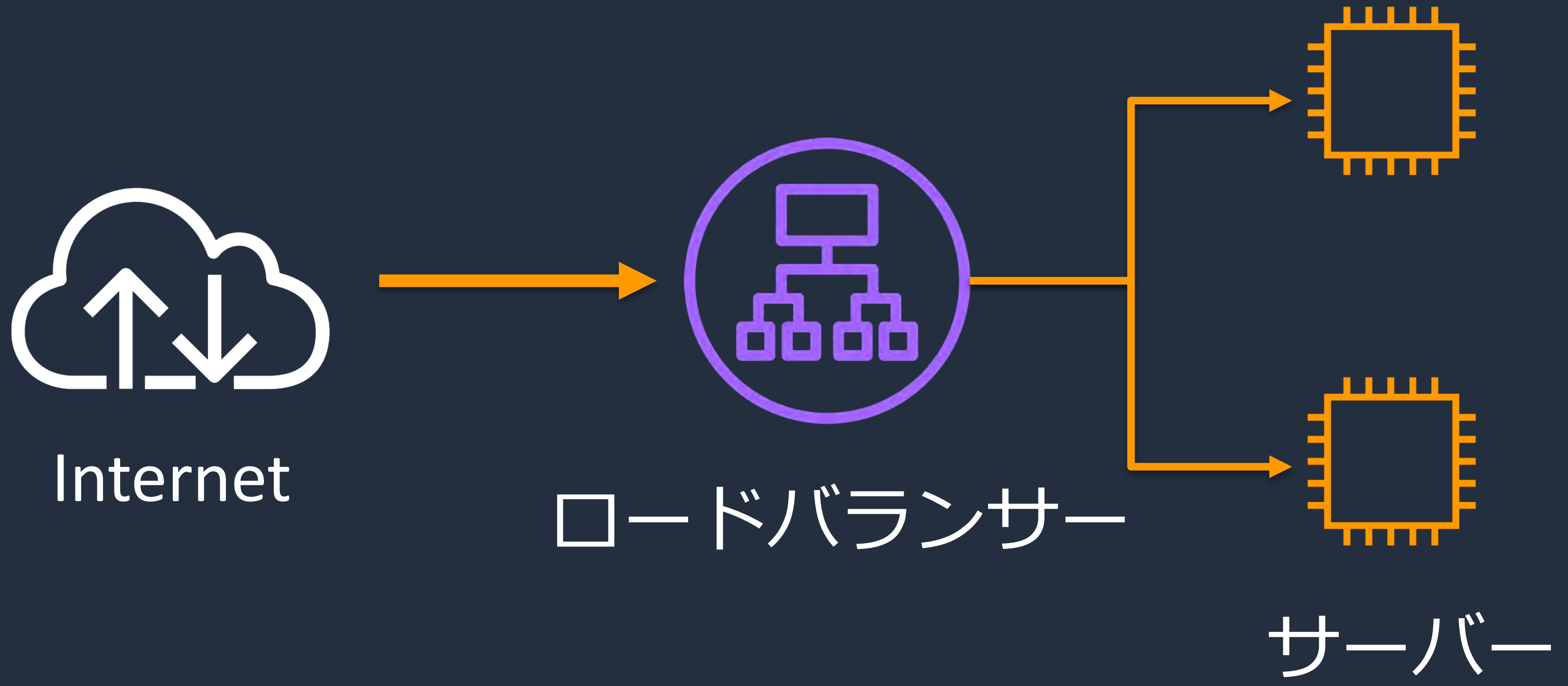
AWS を使ったことがない、使い始めたばかりの方へ

- ➢ 多くは今までオンプレミスで検討すべき
  課題に関することで、オンプレミスで蓄積したナレッジの多くが引き続き役立ちます
- ➢ クラウドならでは、
  クラウドだから可能になる利点、クラウドだから考慮した方がよい点も含まれます

1.スケーラビリティを確保する

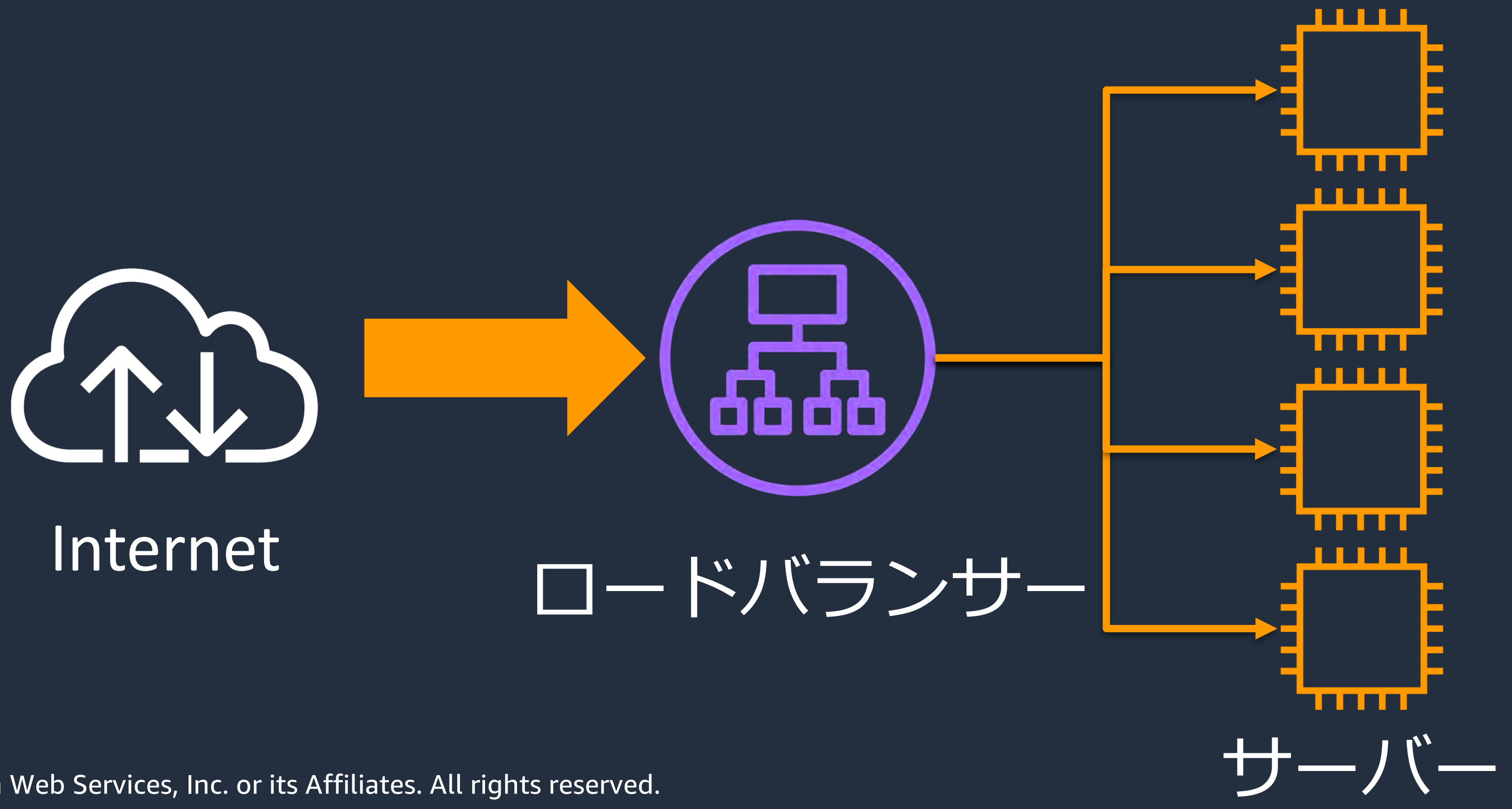
スケーラビリティを確保する

- スケーラビリティとは・・・
  - ➢ ソフトウェアやシステムなどの拡張性、拡張可能性を示す言葉
  - ➢ システムの利用や負荷の増大、用途の拡大などに応じて、どれだけ柔軟に性能や機能を向上、拡張できるかを表している



スケーラビリティを確保する

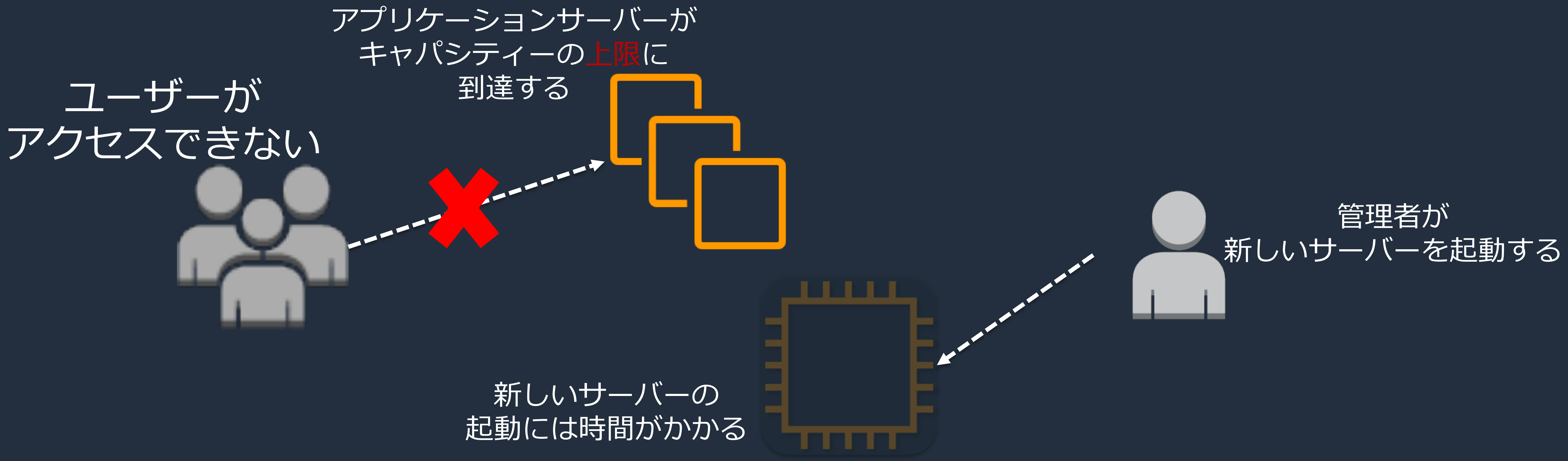
- スケーラビリティとは・・・
  - ➢ ソフトウェアやシステムなどの拡張性、拡張可能性を示す言葉
  - ➢ システムの利用や負荷の増大、用途の拡大などに応じて、どれだけ柔軟に性能や機能を向上、拡張できるかを表している



スケーラビリティを確保する：アンチパターン

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

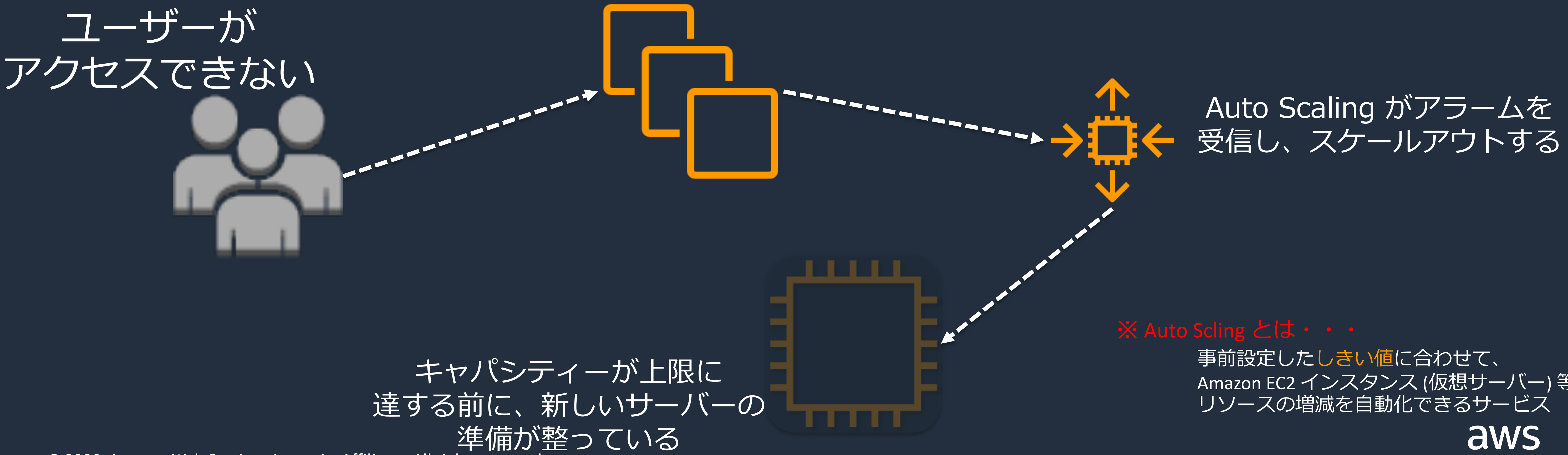
自分のアーキテクチャで需要の変化に対応できるかどうかを確認する



スケーラビリティを確保する：ベストプラクティス

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

自分のアーキテクチャで需要の変化に対応できるかどうかを確認する



※ Auto Scling とは・・・

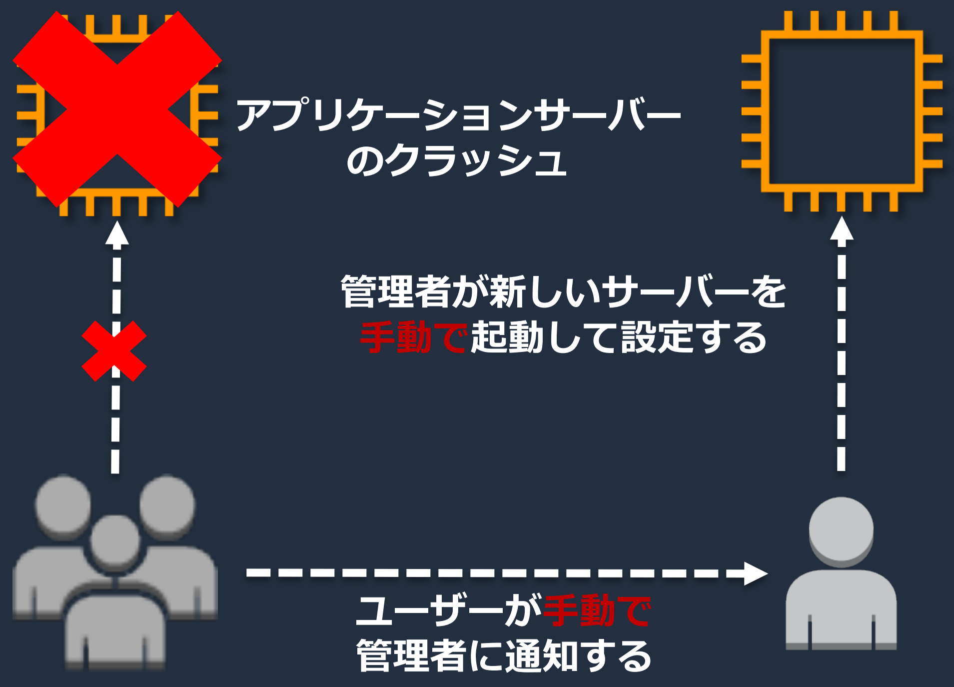
事前設定したしきい値に合わせて、
Amazon EC2 インスタンス (仮想サーバー) 等の
リソースの増減を自動化できるサービス

2.環境を自動化する

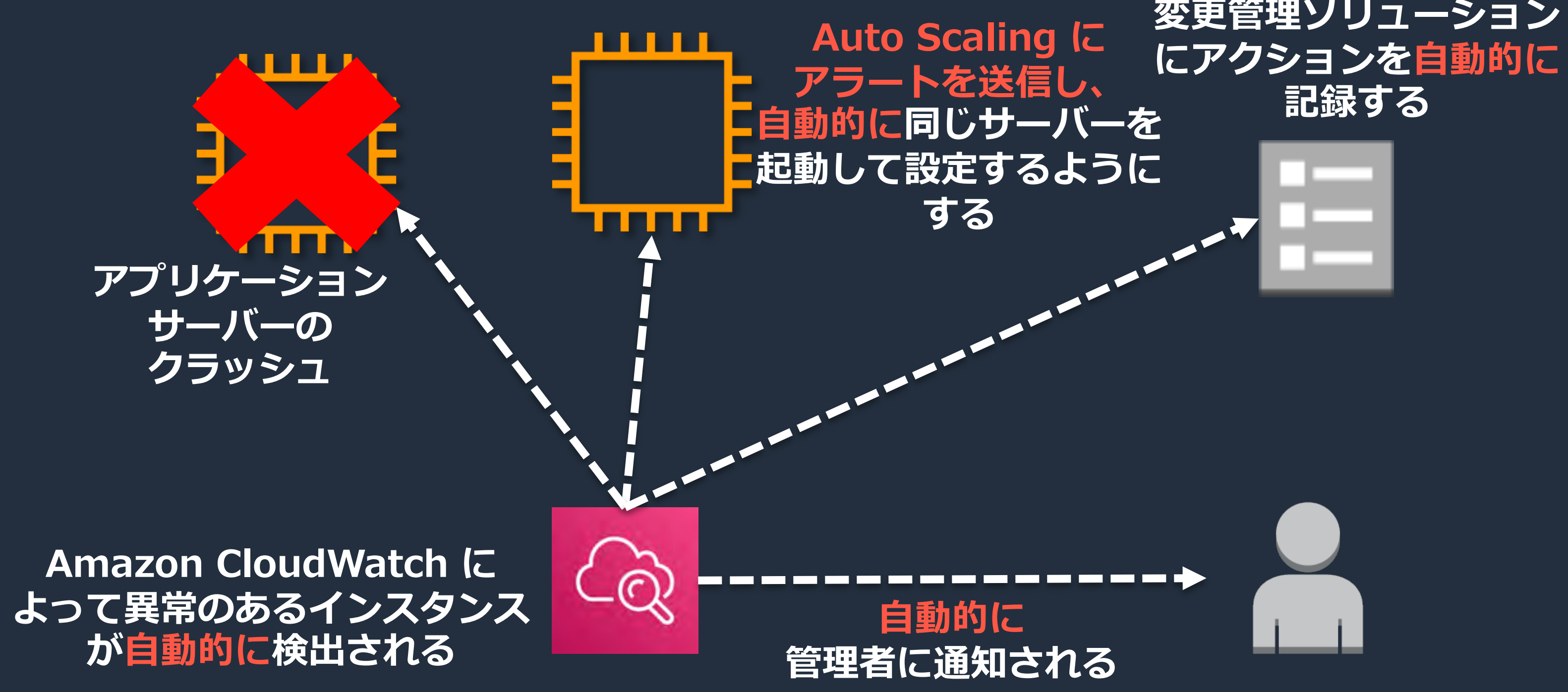
環境を自動化する

リソースの作成 (プロビジョニング)、終了、設定は可能な限り自動化する

アンチパターン



ベストプラクティス



※プロビジョニングとは・・・
プロビジョニングとは、システムやサービスへの需要に対して、
資源の割り当てや設定を行い、利用や運用が可能な状態にすること

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

3. 使い捨て可能なリソースを使用する

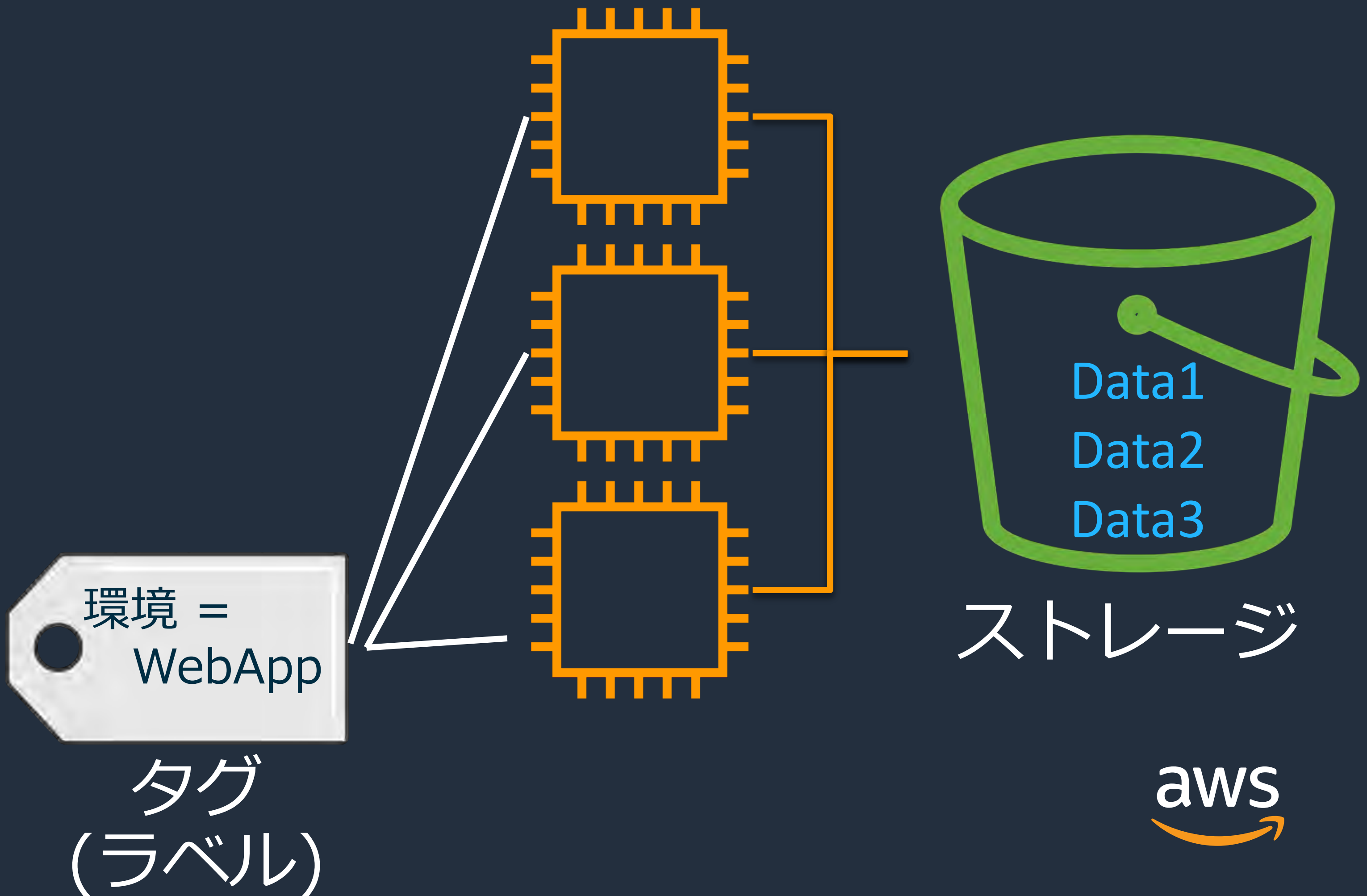
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

使い捨て可能なリソースを使用する

- クラウドではいつでも捨てられるようにリソースを準備しておく
- クラウドではリソース１つ１つに名前は付けず、ラベル (タグ) で管理する
- サーバーには状態 (ステート) を持たせない (ステートレス)
- オンプレミス (ペット) とクラウド (家畜) の特性を理解する

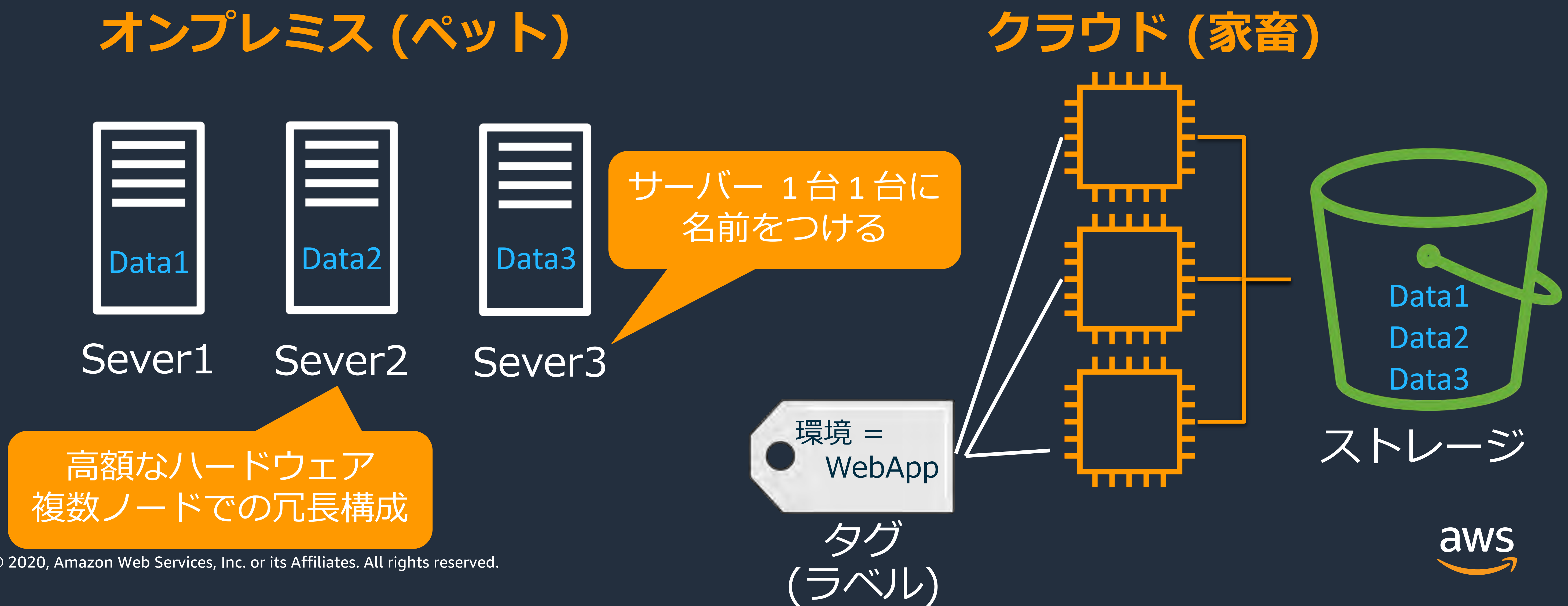
オンプレミス (ペット)

クラウド (家畜)



使い捨て可能なリソースを使用する

- クラウドではいつでも捨てられるようにリソースを準備しておく
- クラウドではリソース１つ１つに名前は付けず、ラベル (タグ) で管理する
- サーバーには状態 (ステート) を持たせない (ステートレス)
- オンプレミス (ペット) とクラウド (家畜) の特性を理解する

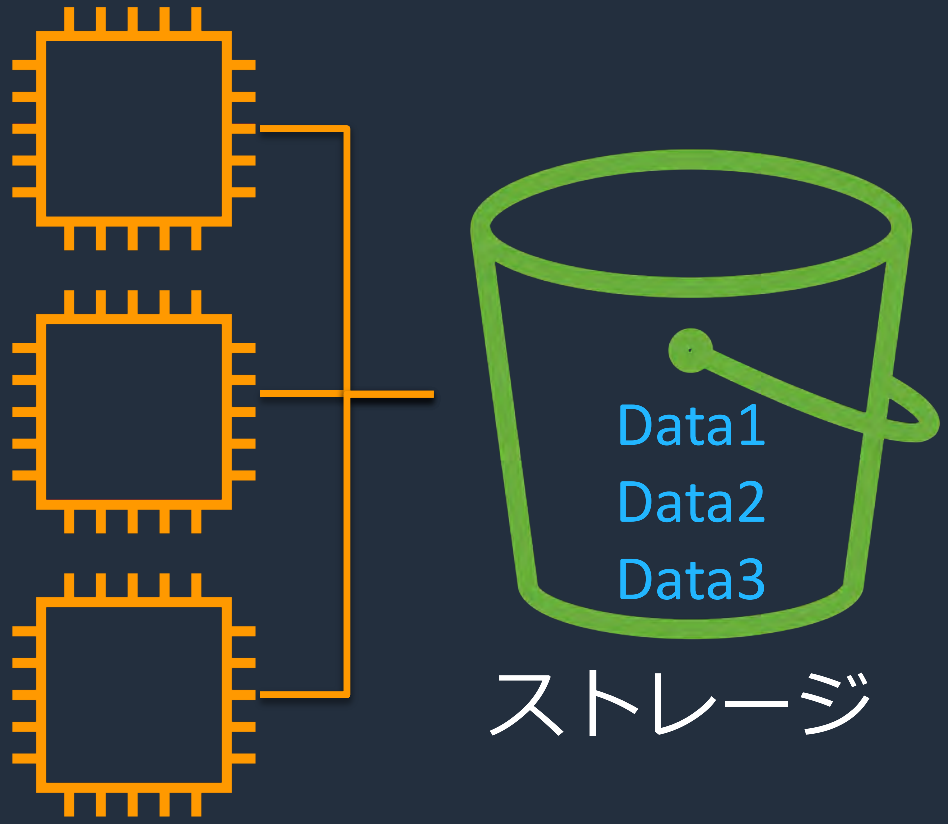


使い捨て可能なリソースを使用する

- 例えば負荷が増大したとき

オンプレミス (ペット)

クラウド (家畜)

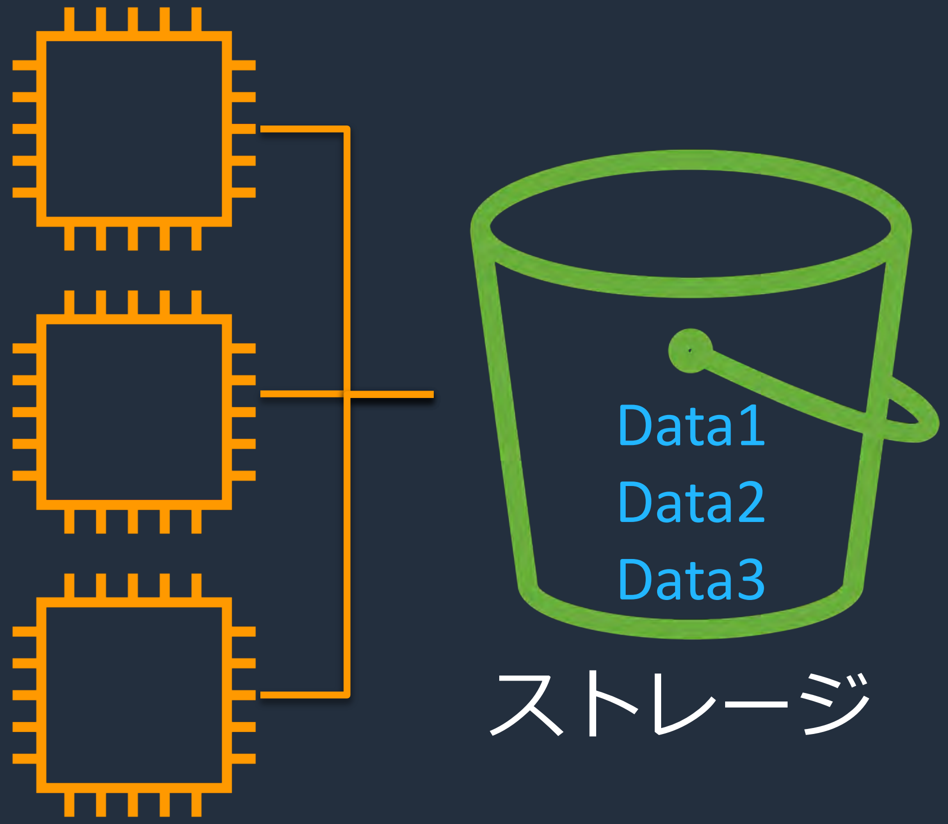


使い捨て可能なリソースを使用する

- 例えば負荷が増大したとき

オンプレミス (ペット)

クラウド (家畜)

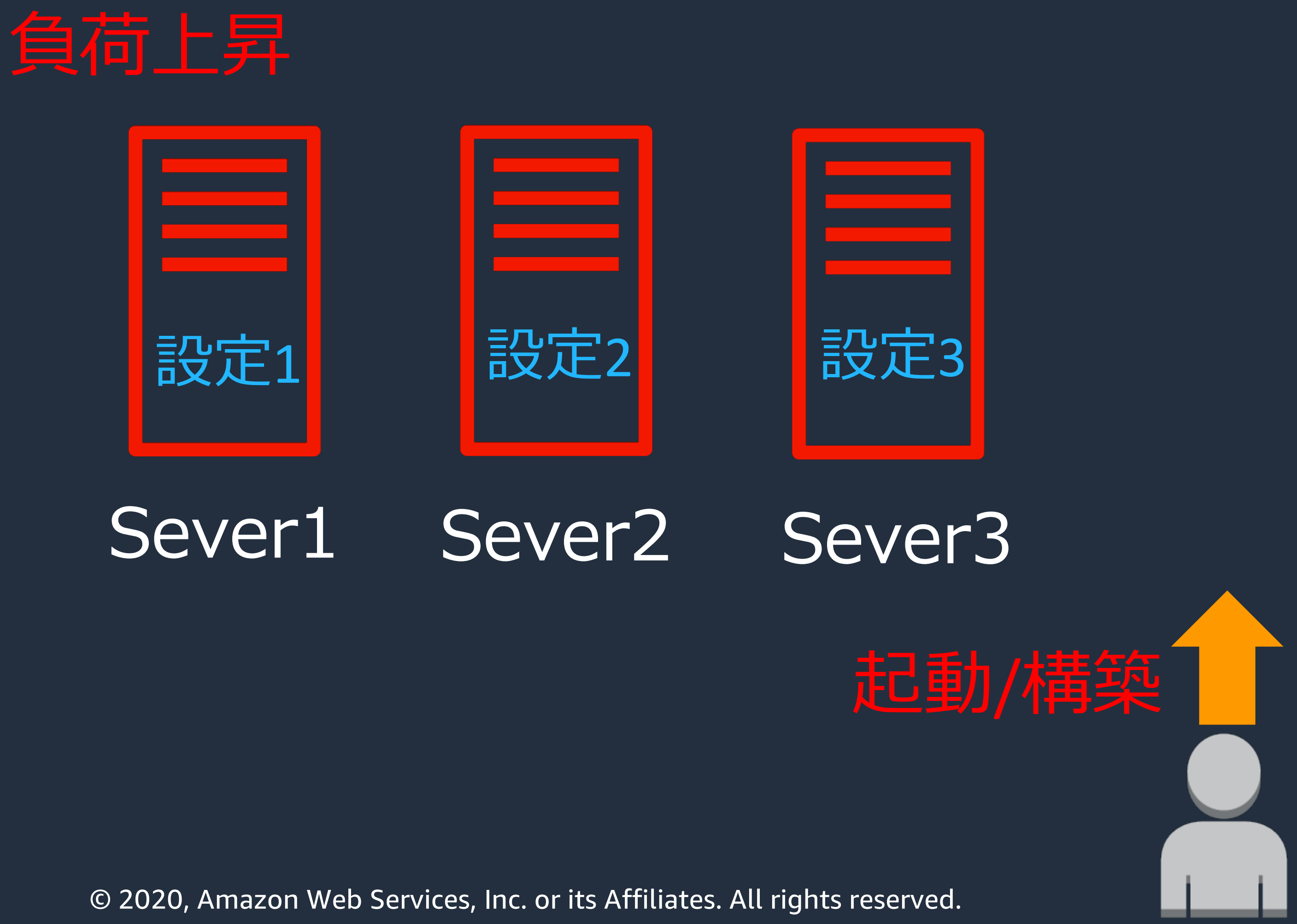


使い捨て可能なリソースを使用する

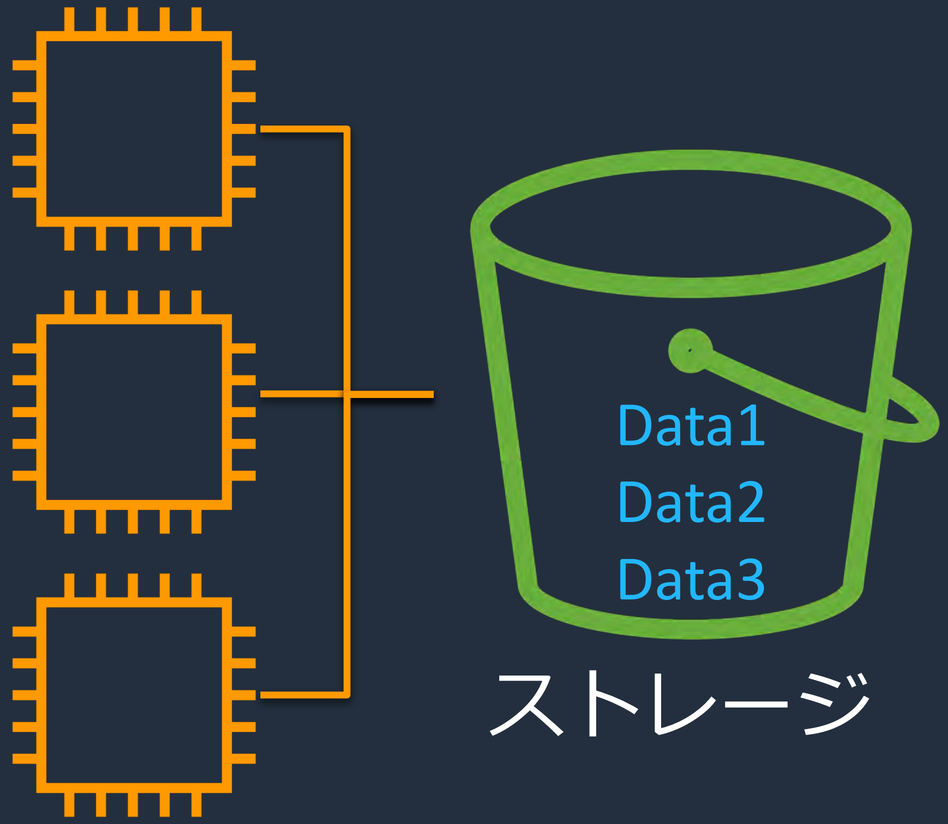
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

- 例えば負荷が増大したとき

オンプレミス (ペット)



クラウド (家畜)



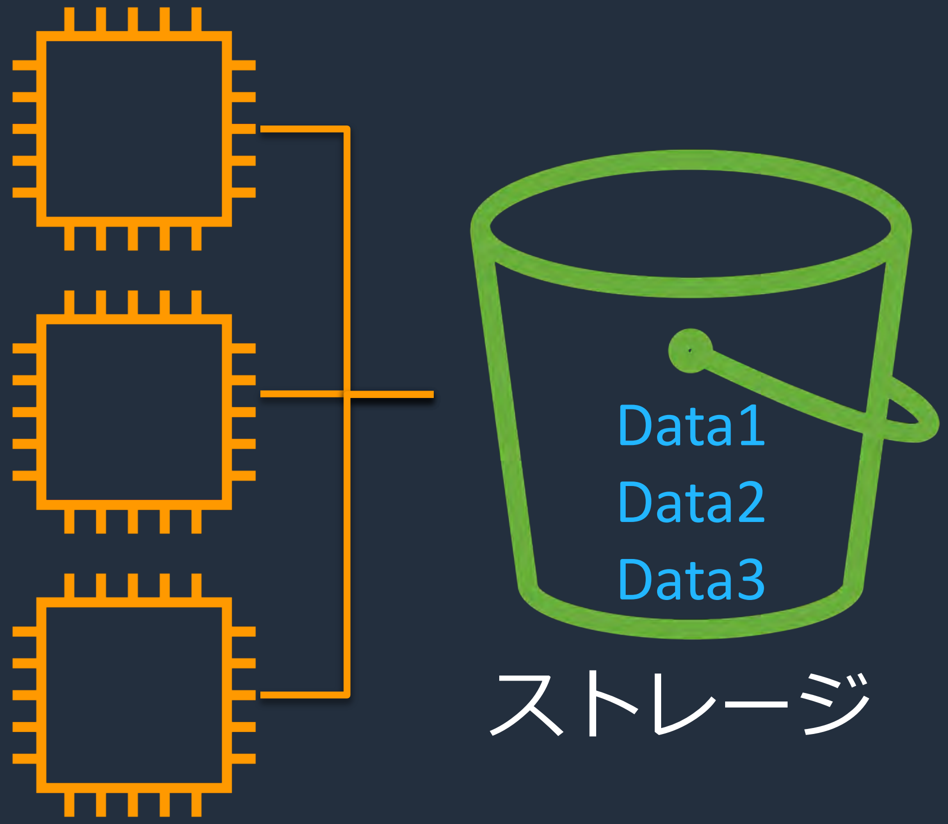
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

使い捨て可能なリソースを使用する

- 例えば負荷が増大したとき

オンプレミス (ペット)

クラウド (家畜)



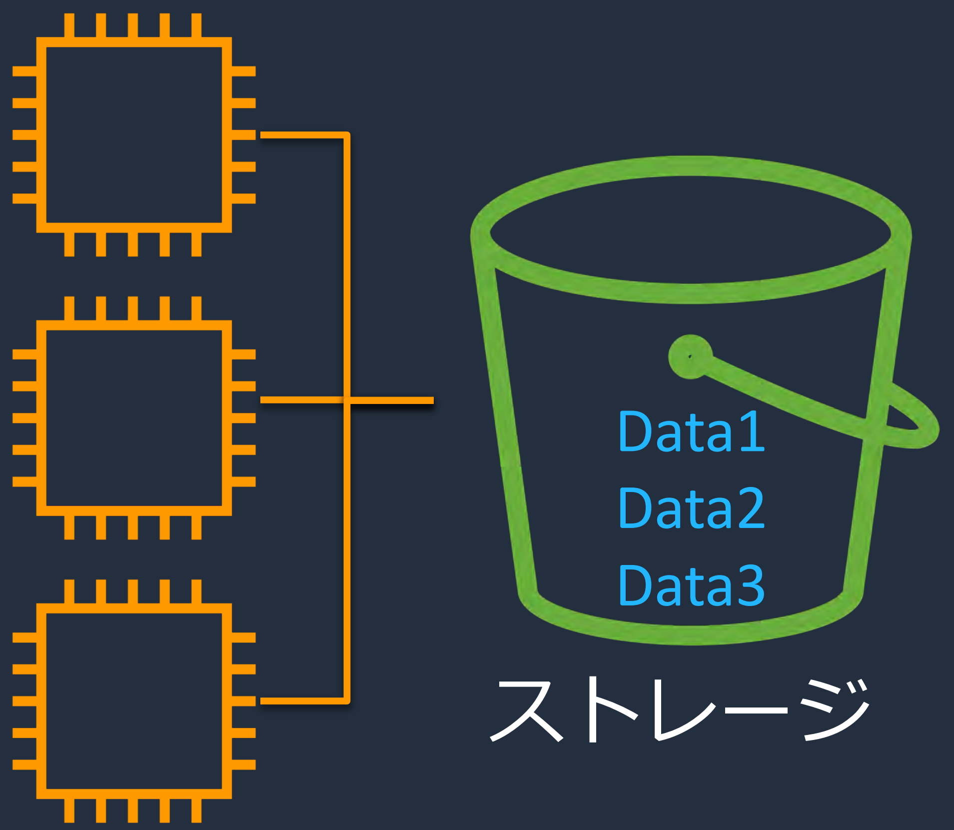
使い捨て可能なリソースを使用する

ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)



使い捨て可能なリソースを使用する

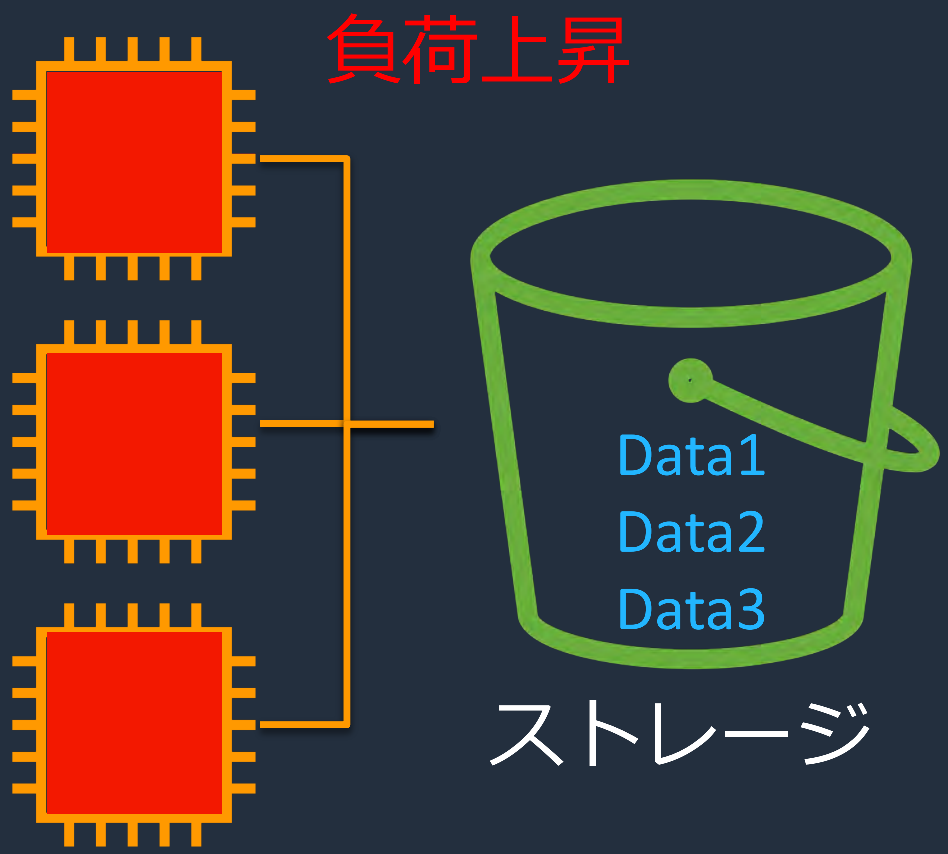
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)



使い捨て可能なリソースを使用する

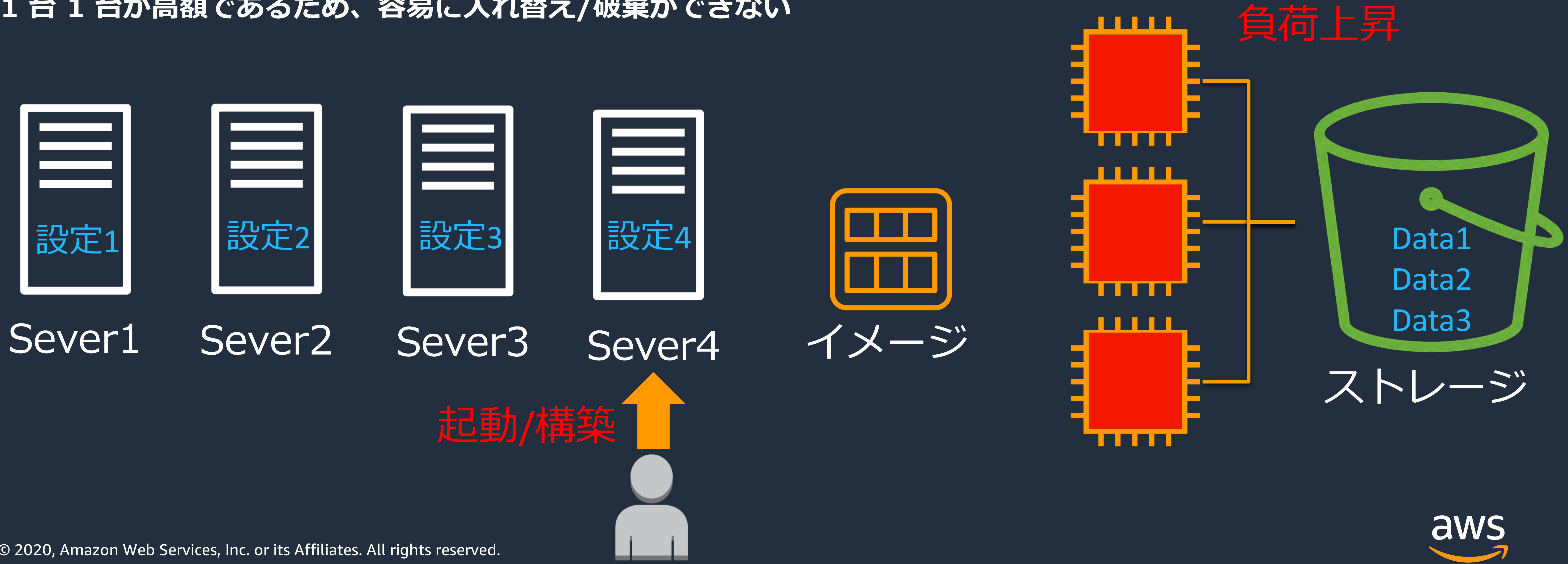
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- サーバー毎に設定が異なる可能性がある
- 負荷の上昇要因を調査し、必要に応じて追加の投資を検討する
- 1 台 1 台が高額であるため、容易に入れ替え/破棄ができない

クラウド (家畜)



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

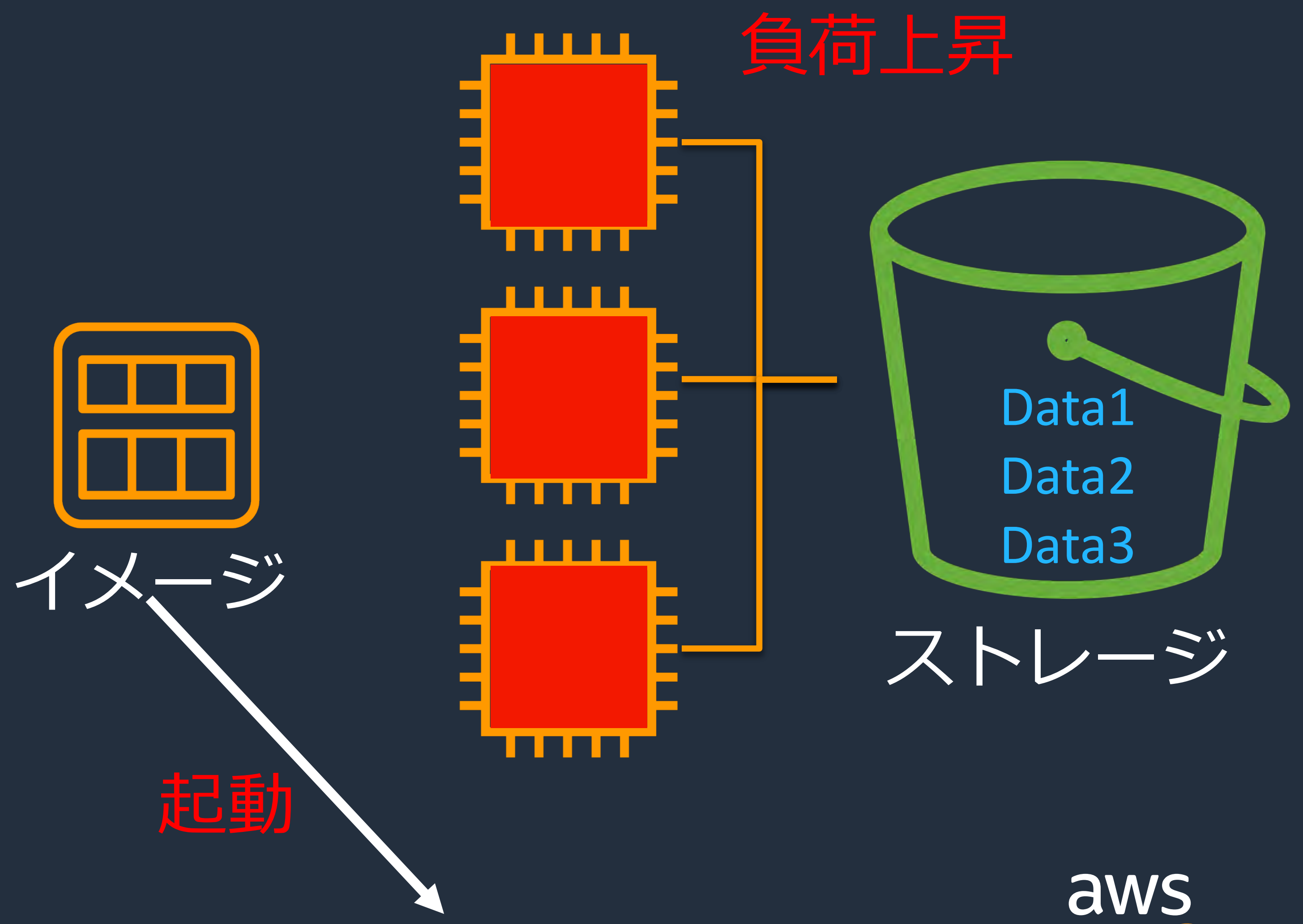
使い捨て可能なリソースを使用する

ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- サーバー毎に設定が異なる可能性がある
- 負荷の上昇要因を調査し、必要に応じて追加の投資を検討する
- 1 台 1 台が高額であるため、容易に入れ替え/破棄ができない

クラウド (家畜)



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

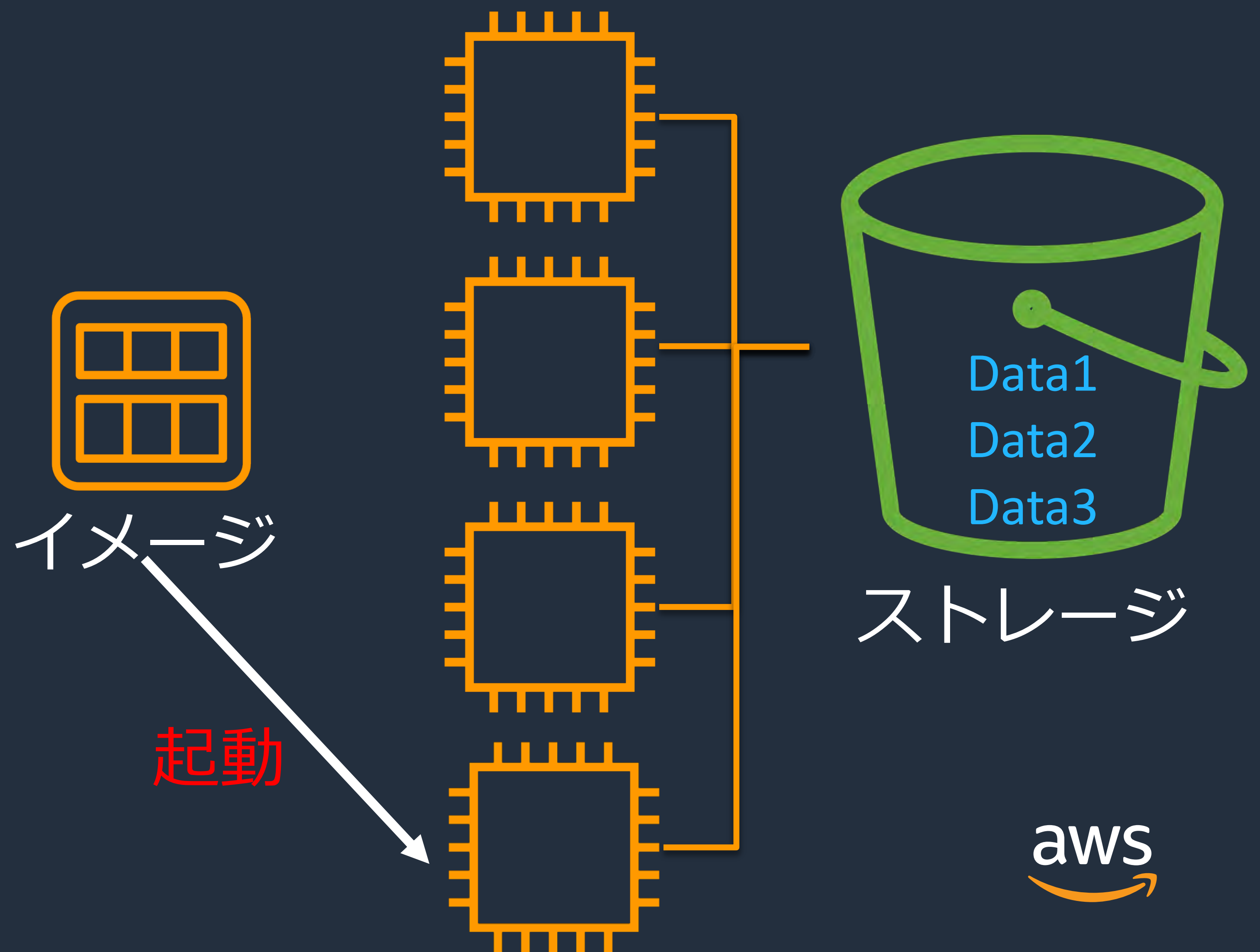
使い捨て可能なリソースを使用する

ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

使い捨て可能なリソースを使用する

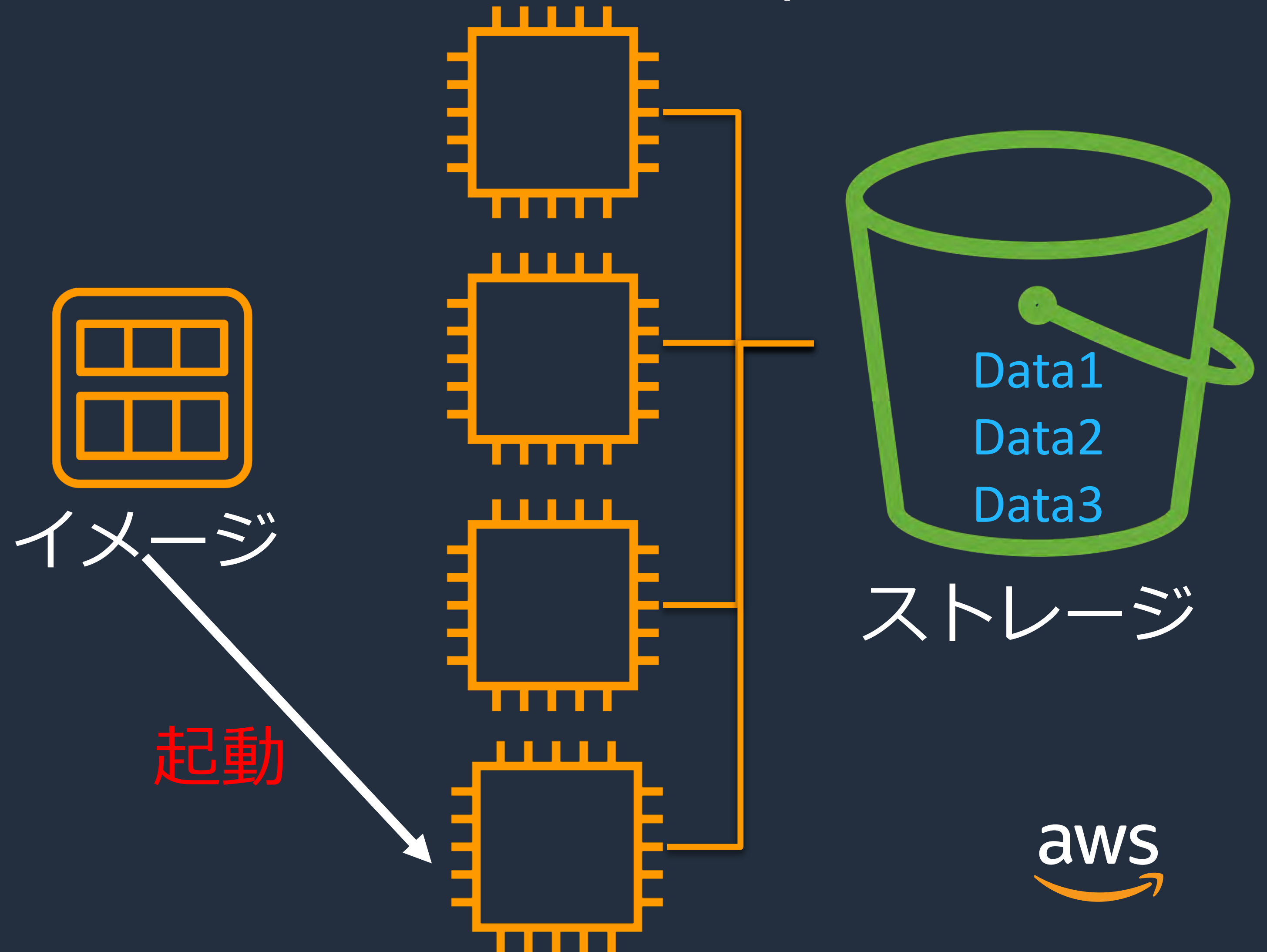
ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)

- **不要になったらいつでも入れ替え/破棄できる**
- **同一イメージから起動できるため、リソース作成/追加を自動化しやすい**



使い捨て可能なリソースを使用する

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

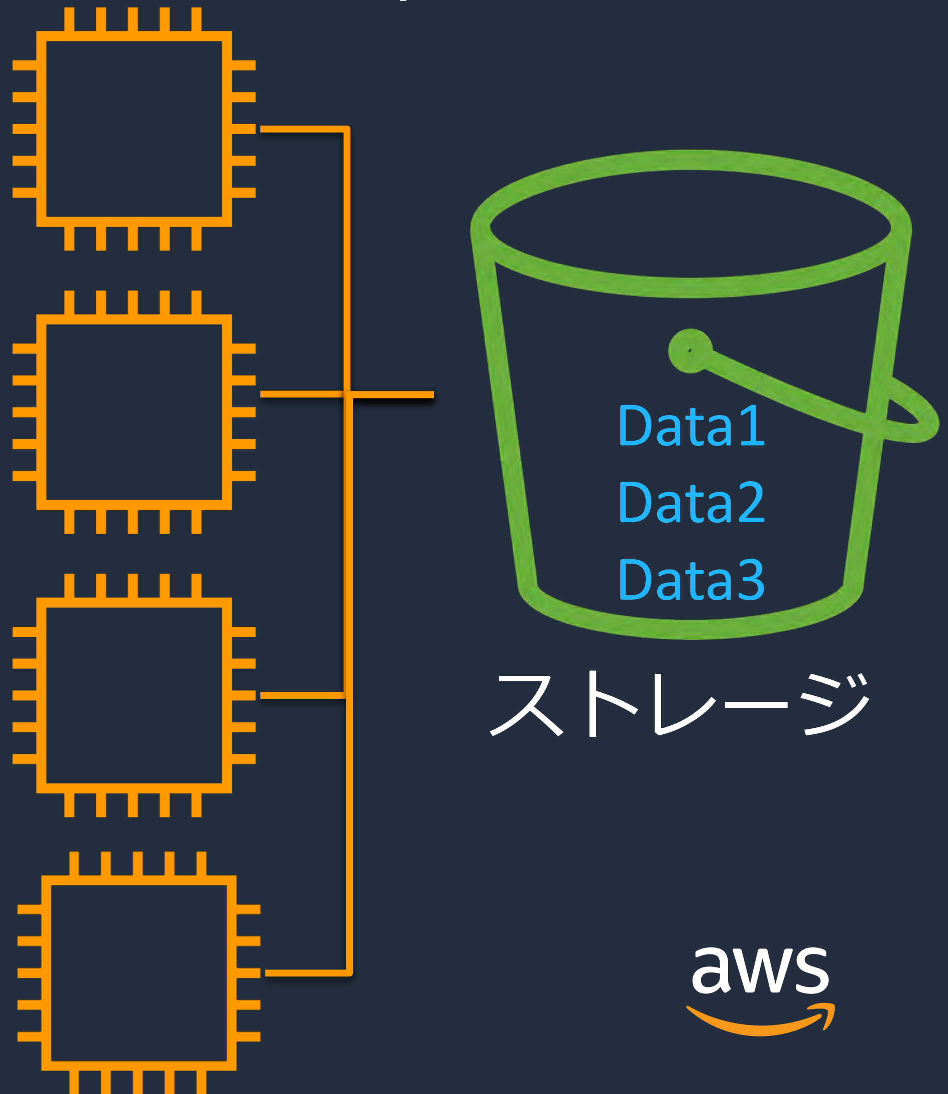
ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- サーバー毎に設定が異なる可能性がある
- 負荷の上昇要因を調査し、必要に応じて追加の投資を検討する
- 1 台 1 台が高額であるため、容易に入れ替え/破棄ができない

クラウド (家畜)

- 不要になったらいつでも入れ替え/破棄できる
- 同一イメージから起動できるため、リソース作成/追加を自動化しやすい



使い捨て可能なリソースを使用する

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

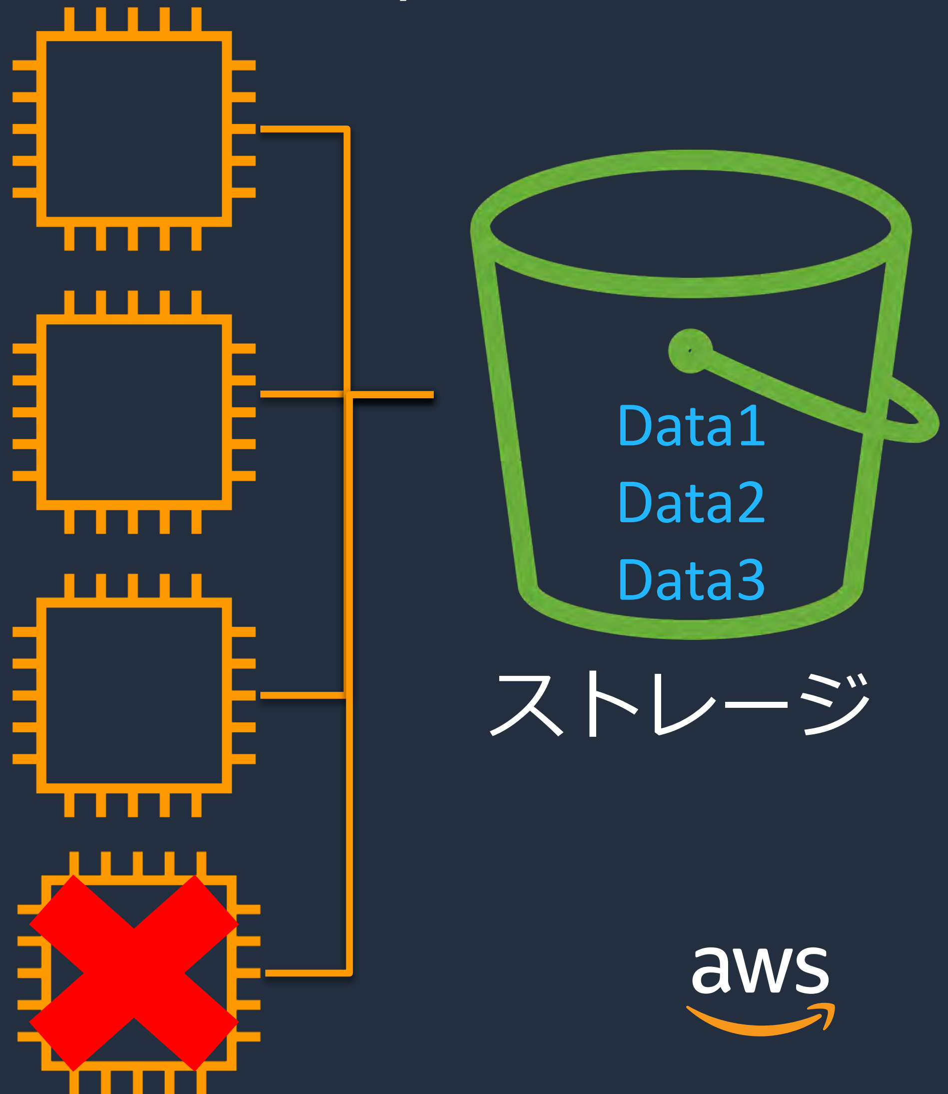
ロ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)

- **不要になったらいつでも入れ替え/破棄できる**
- **同一イメージから
起動できるため、リソース作成/追加を自動化しやすい**



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

使い捨て可能なリソースを使用する

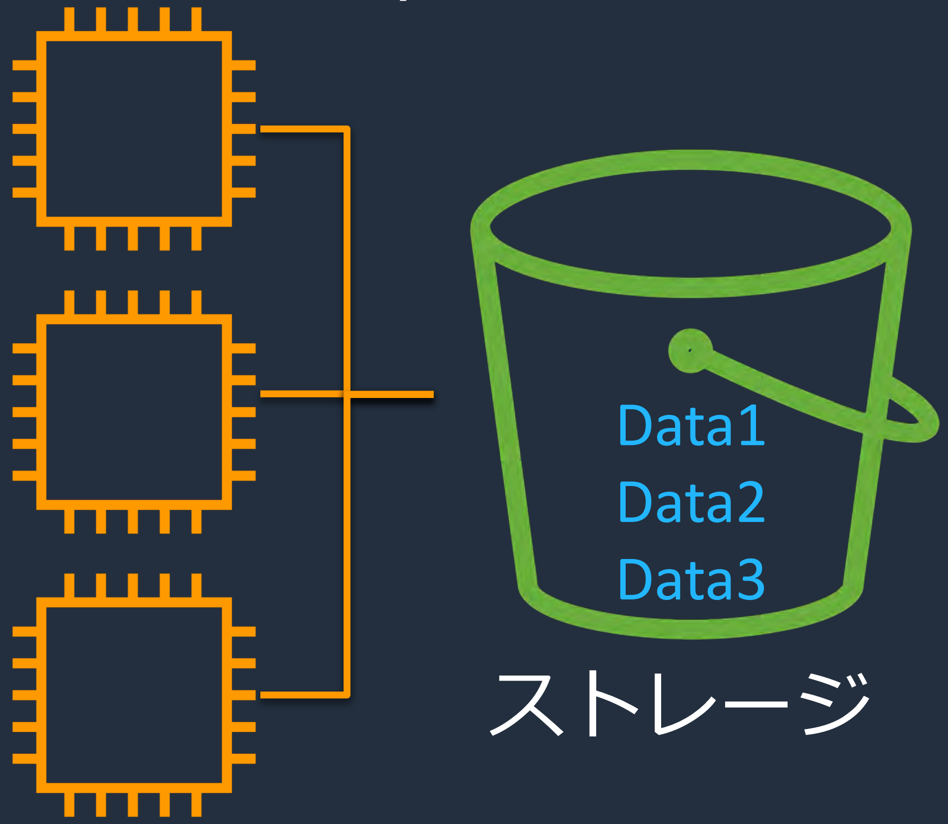
□ 例えば負荷が増大したとき

オンプレミス (ペット)

- **サーバー毎に設定が異なる可能性がある**
- **負荷の上昇要因を調査し、必要に応じて追加の投資を検討する**
- **1 台 1 台が高額であるため、容易に入れ替え/破棄ができない**

クラウド (家畜)

- **不要になったらいつでも入れ替え/破棄できる**
- **同一イメージから起動できるため、リソース作成/追加を自動化しやすい**



(参考) 使い捨て可能なリソースを使用する

動的にプロビジョニングされるクラウドコンピューティングの特性を活用する

アンチパターン

- 長期間の運用で各サーバーの設定に違いが生じる
- 必要でないときにもリソースが稼働している
- IP アドレスの固定により柔軟性が失われている
- 使用中のハードウェアで新しい更新をテストすることが難しく不便である

ベストプラクティス

- 設定が同じ場合、新しいリソースのデプロイを自動化する
- 使用していないリソースは終了する
- 新しい IP アドレスに自動的に切り替える
- 新しいリソースの更新内容をテストしてから、古いリソースを置き換える

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

4. コンポーネントを疎結合にする

疎結合/密結合について

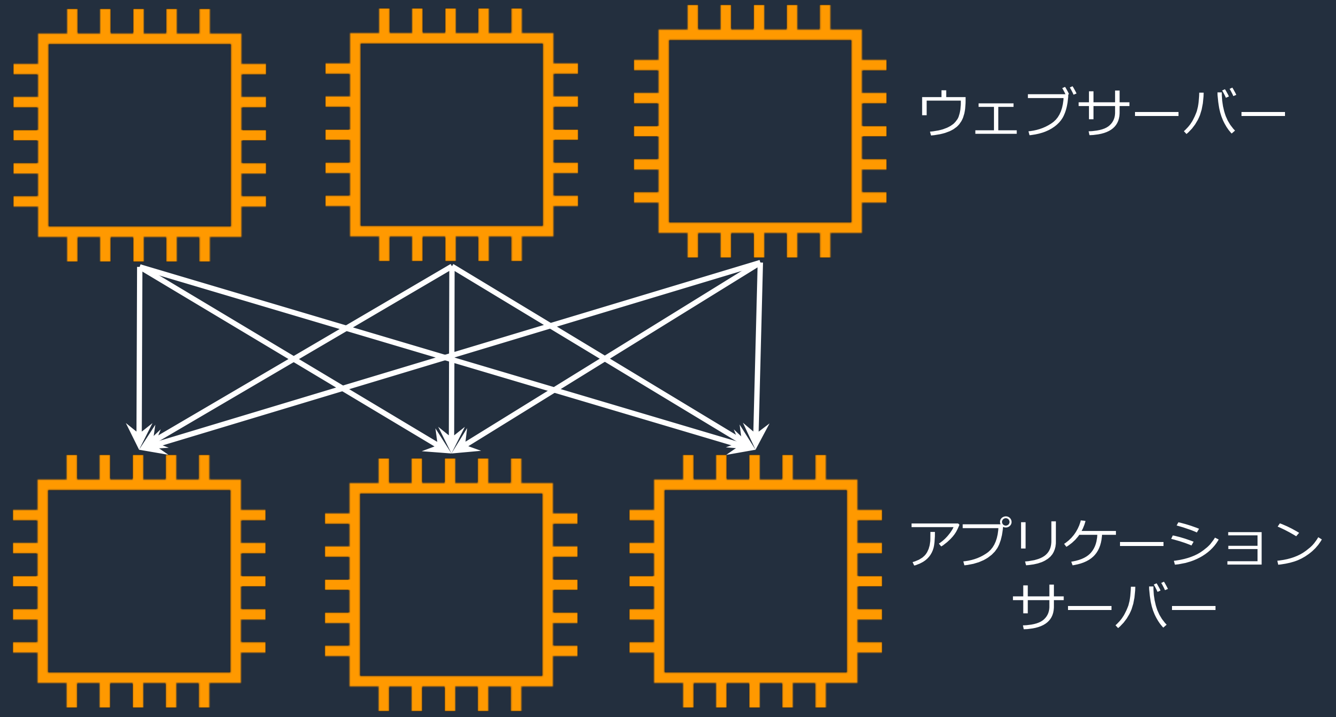
- 疎結合とは・・・
  - システムの構成要素 (コンポーネント) 間の結びつきや互いの依存関係、関連性などが弱く、各々の独立性が高い状態のこと

- 密結合とは・・・
  - システムの構成要素 (コンポーネント) 間の結びつきや互いの依存関係、関連性などが強く、各々の独立性が低い状態のこと

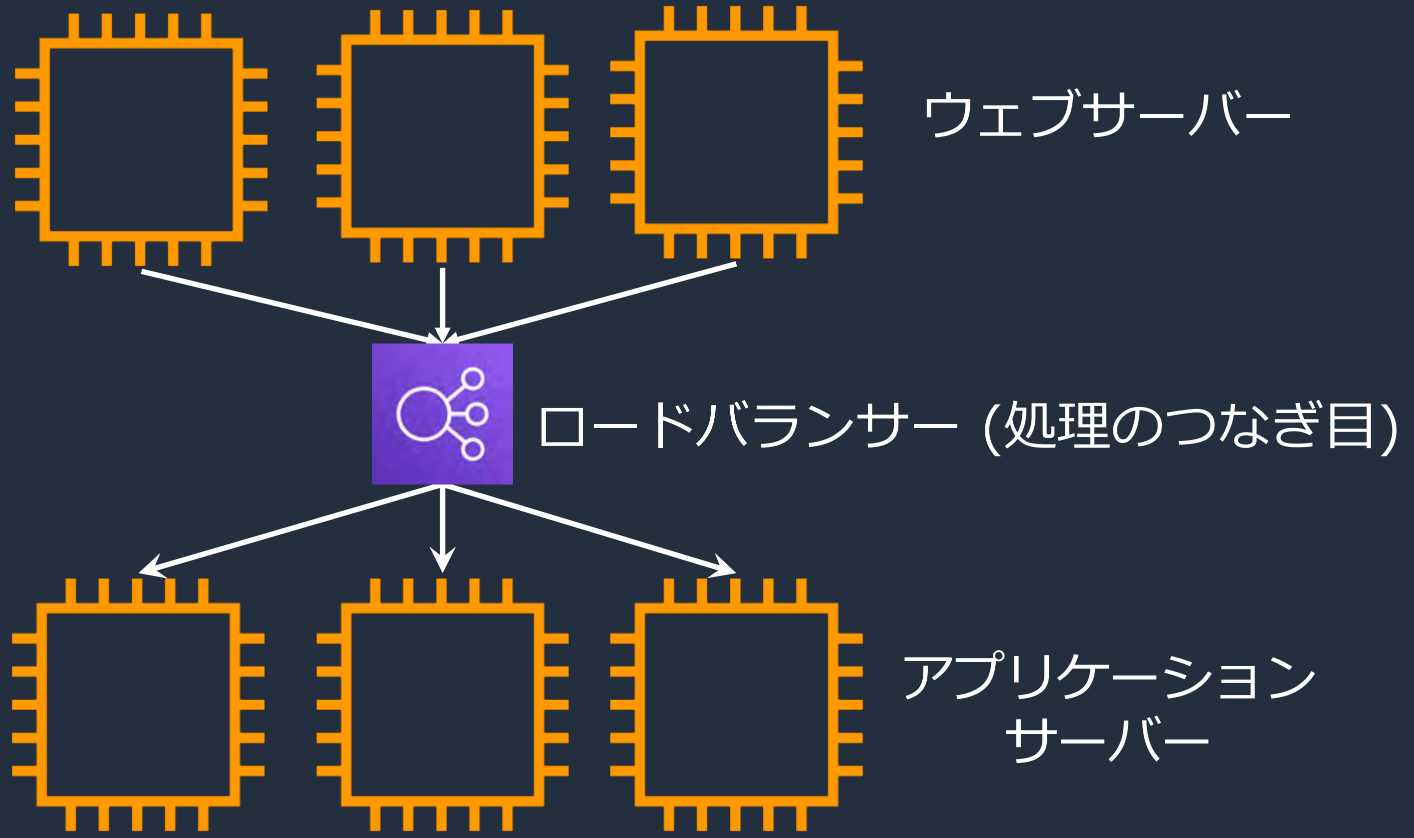
疎結合/密結合例

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

密結合



疎結合



疎結合化 : メリット & アプローチ

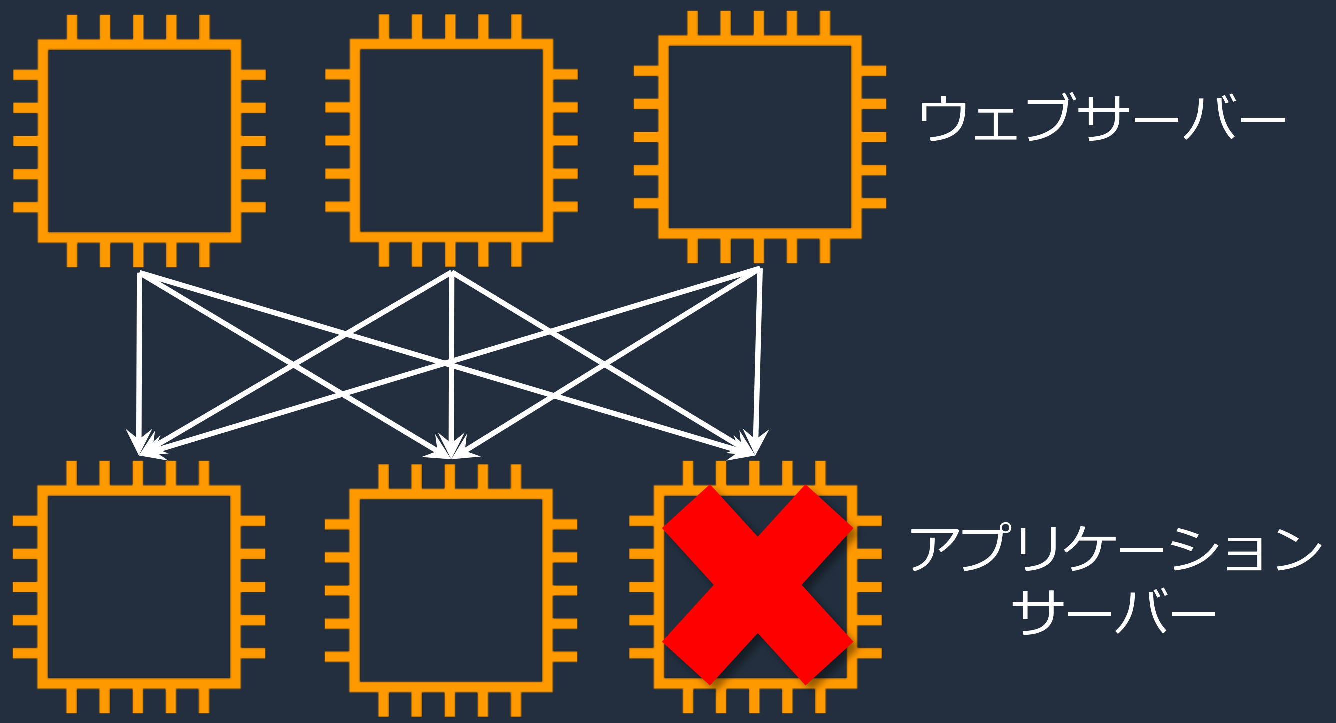
- メリット
  - 障害ポイントが判別しやすい
  - 必要な部分のみに変更（アプリ修正、拡張、縮小など）を適用できる

- アプローチ
  - 処理種別やデータ特性ごとにコンポーネントを細分化
  - ステート（データ）を持つレイヤと持たないレイヤを分離
  - 「処理のつなぎ目」を検討
  - データ特性ごとにデータストアを選択

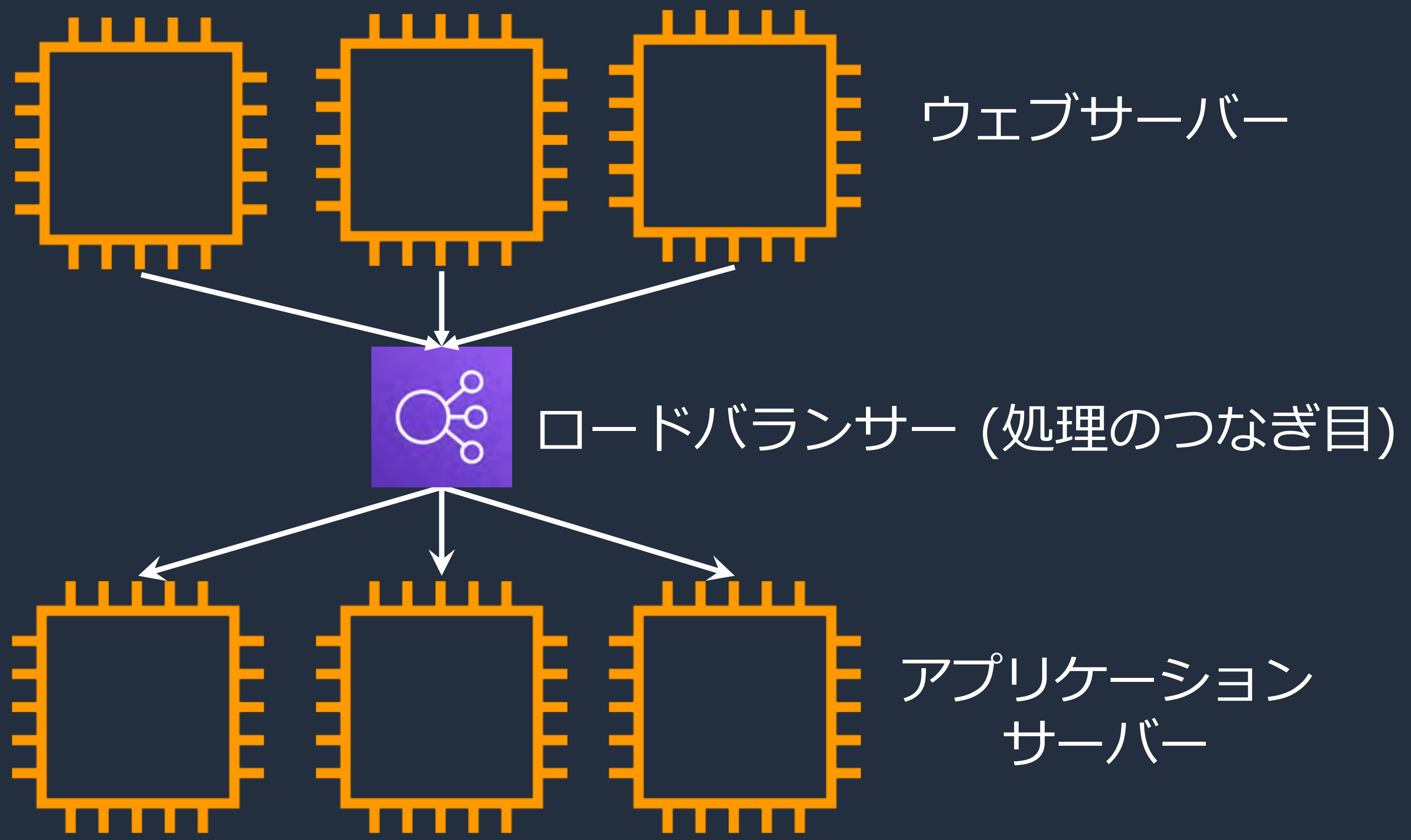
疎結合と密結合の違い：障害時

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

密結合

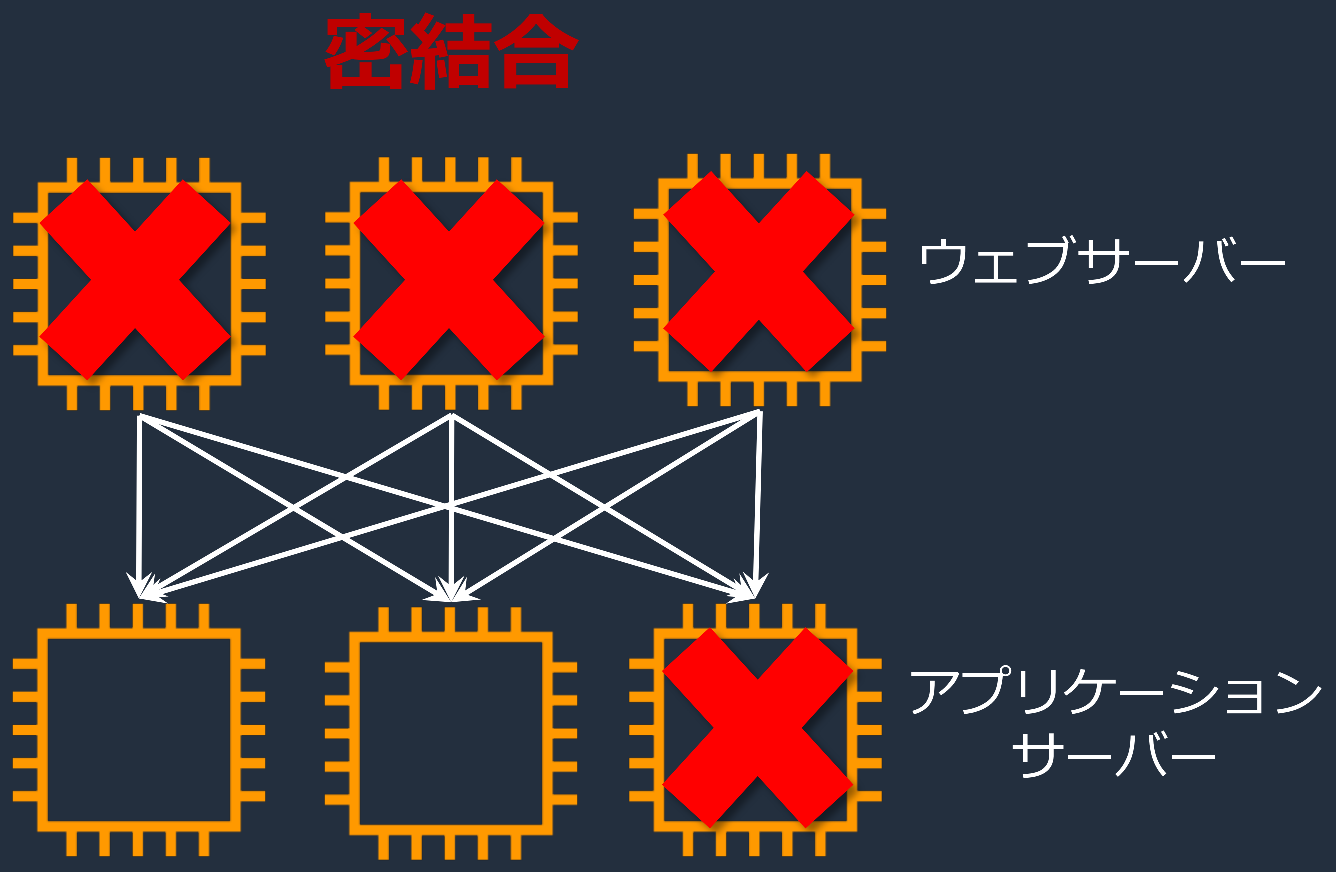


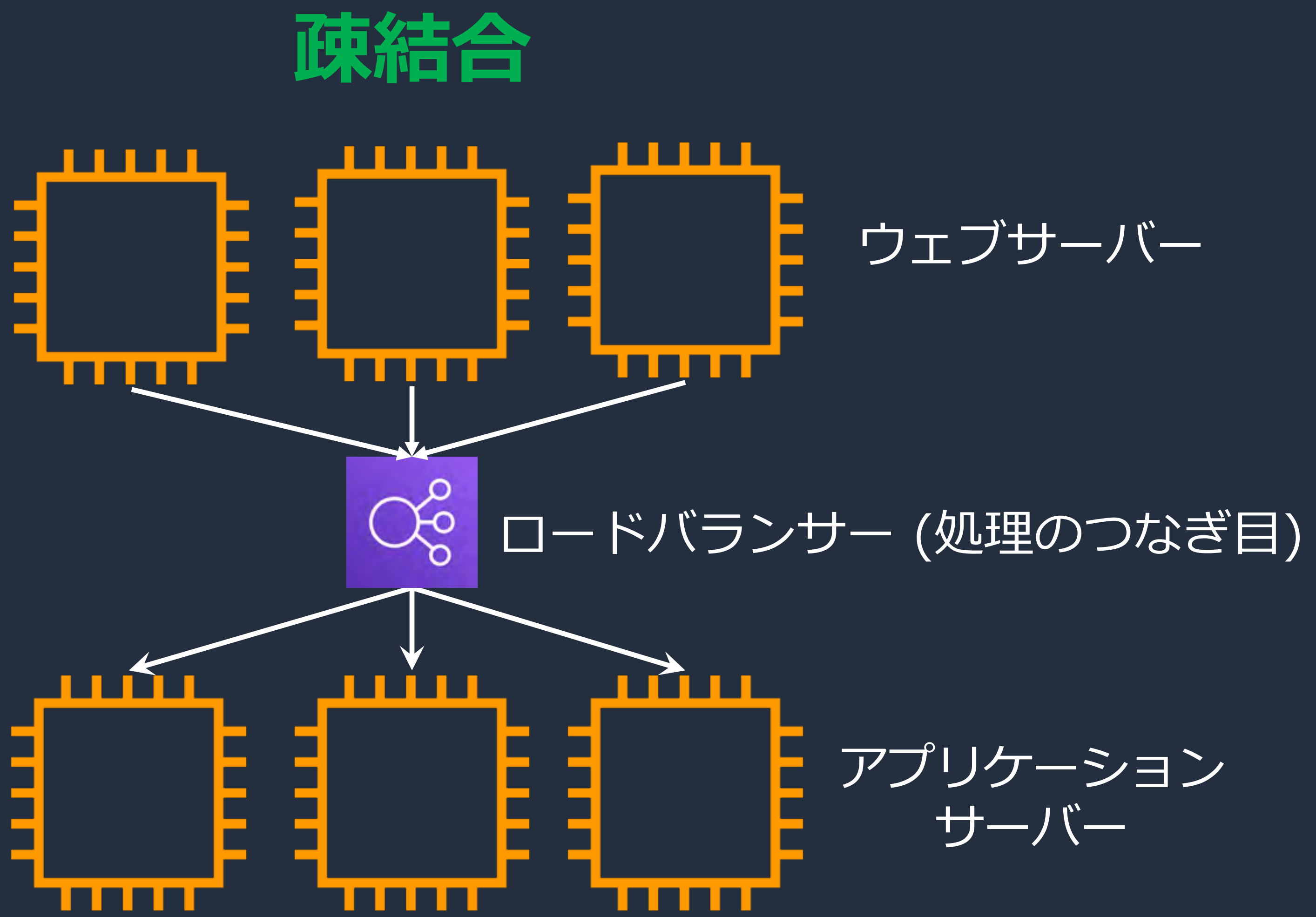
疎結合



疎結合と密結合の違い：障害時

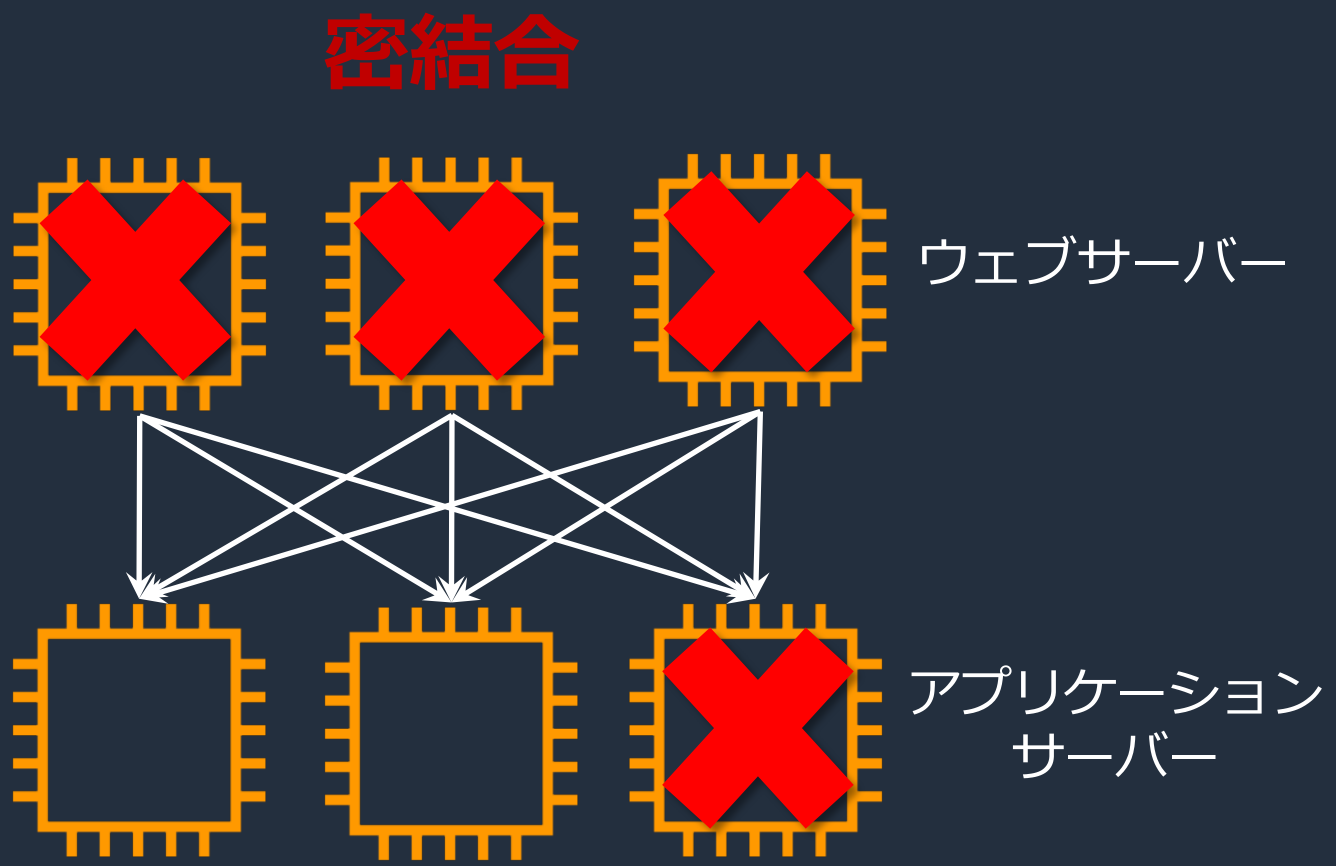
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

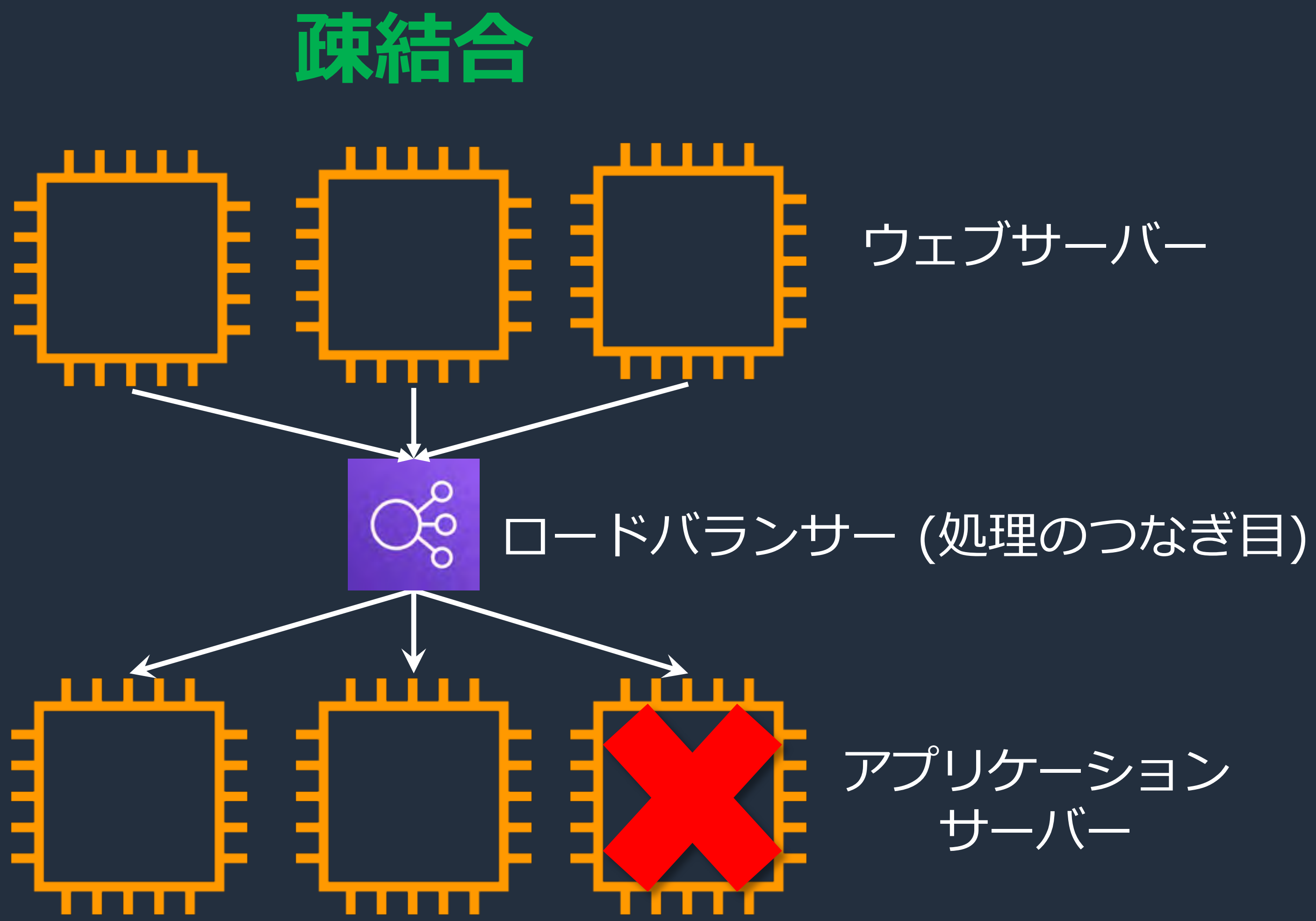




疎結合と密結合の違い：障害時

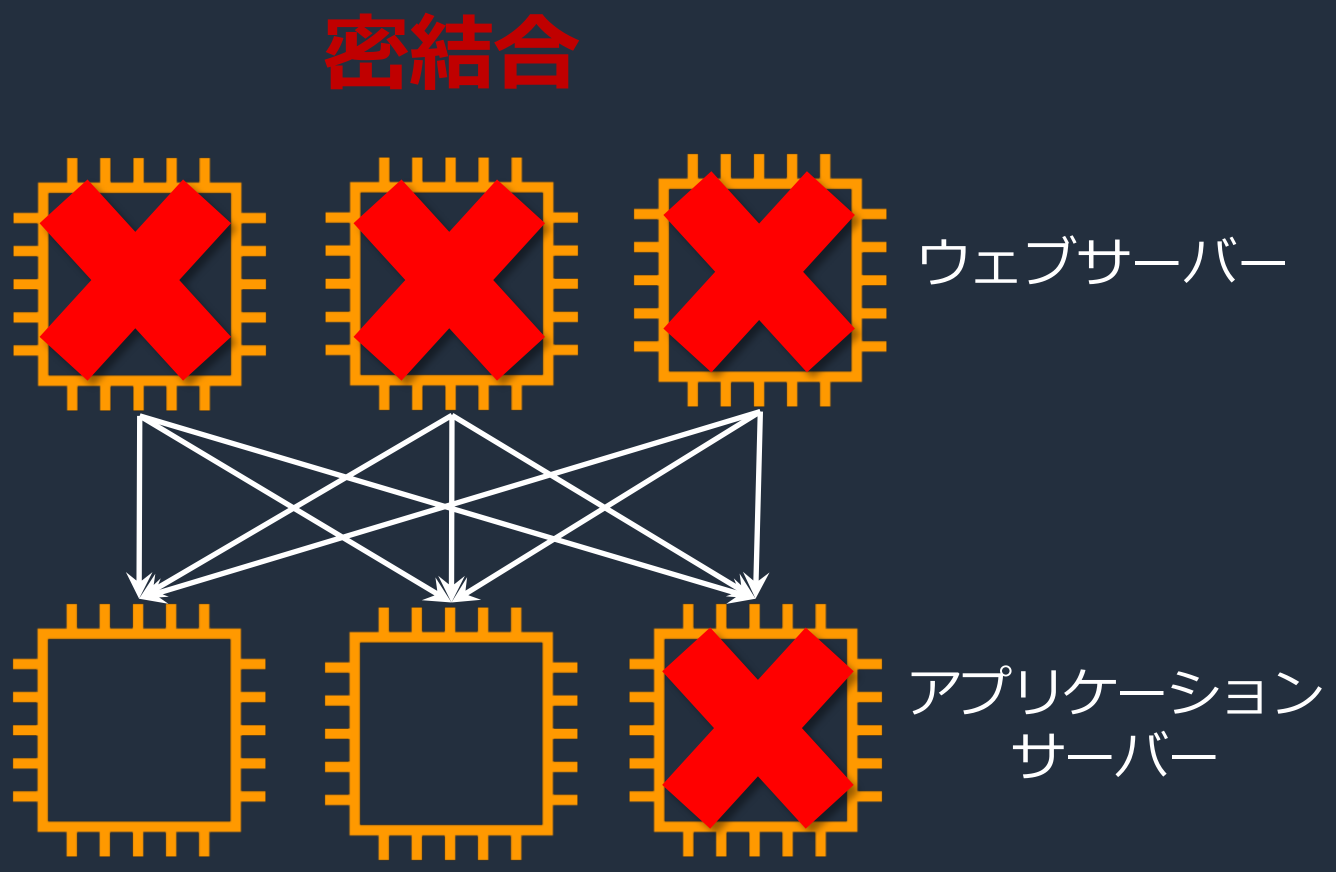
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

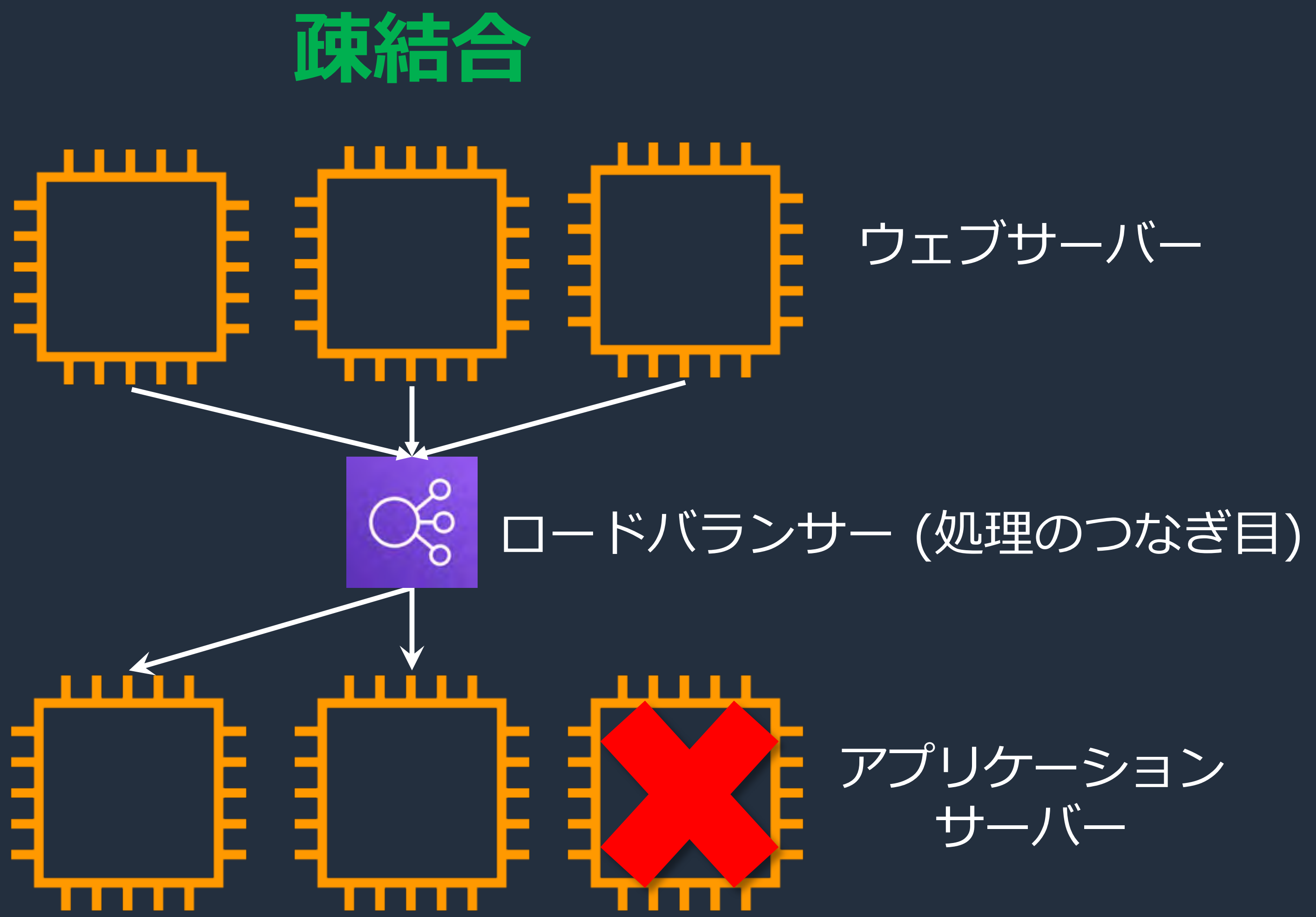




疎結合と密結合の違い：障害時

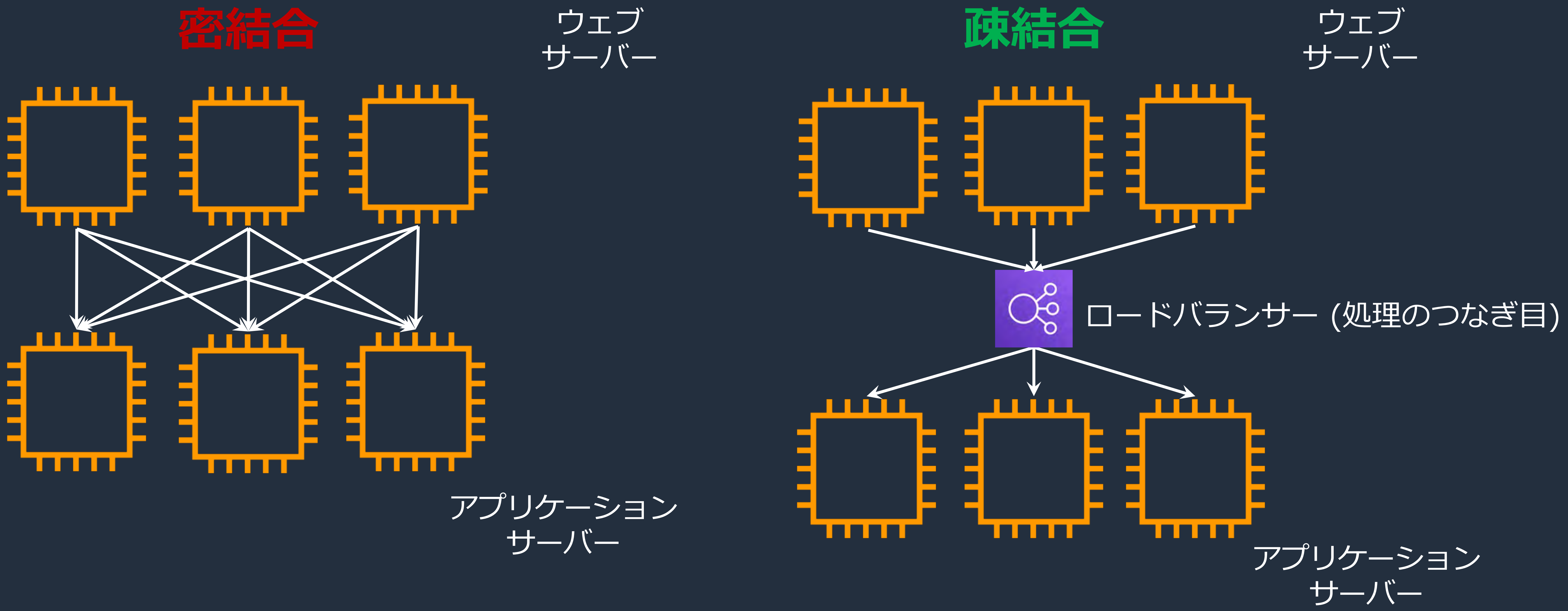
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理





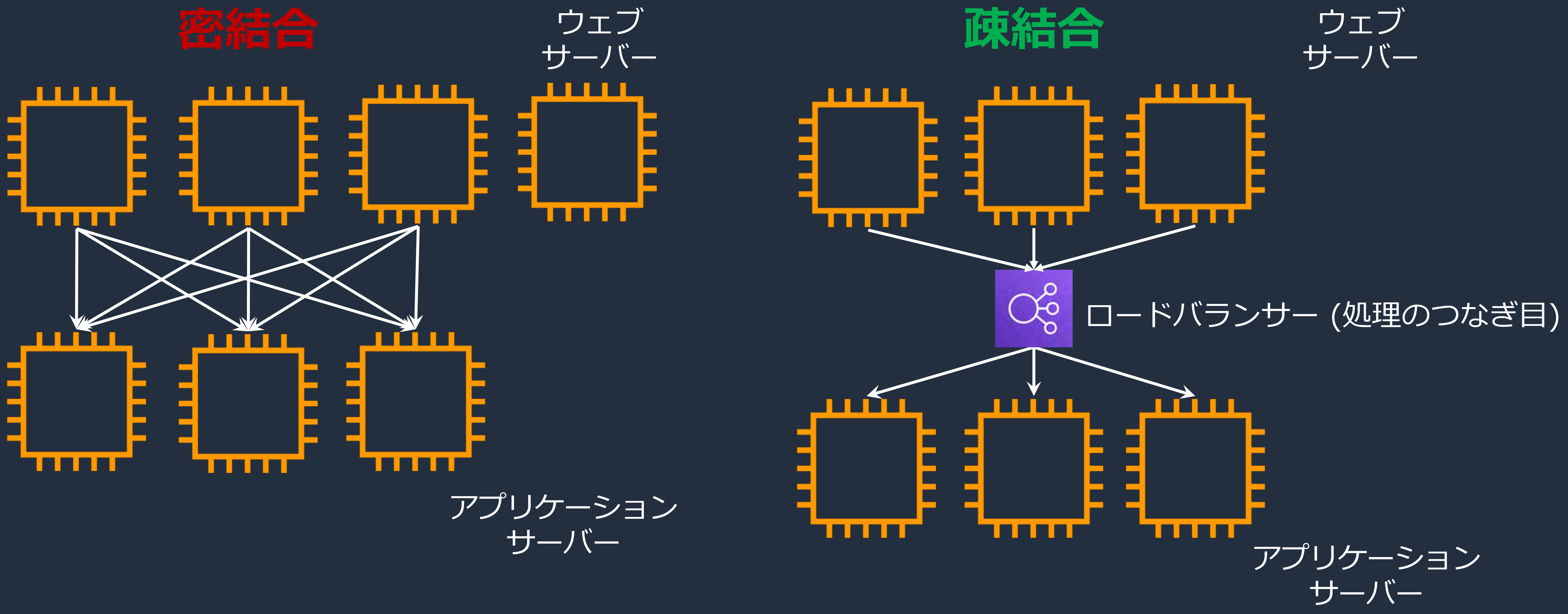
疎結合と密結合の違い：スケーラビリティ

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



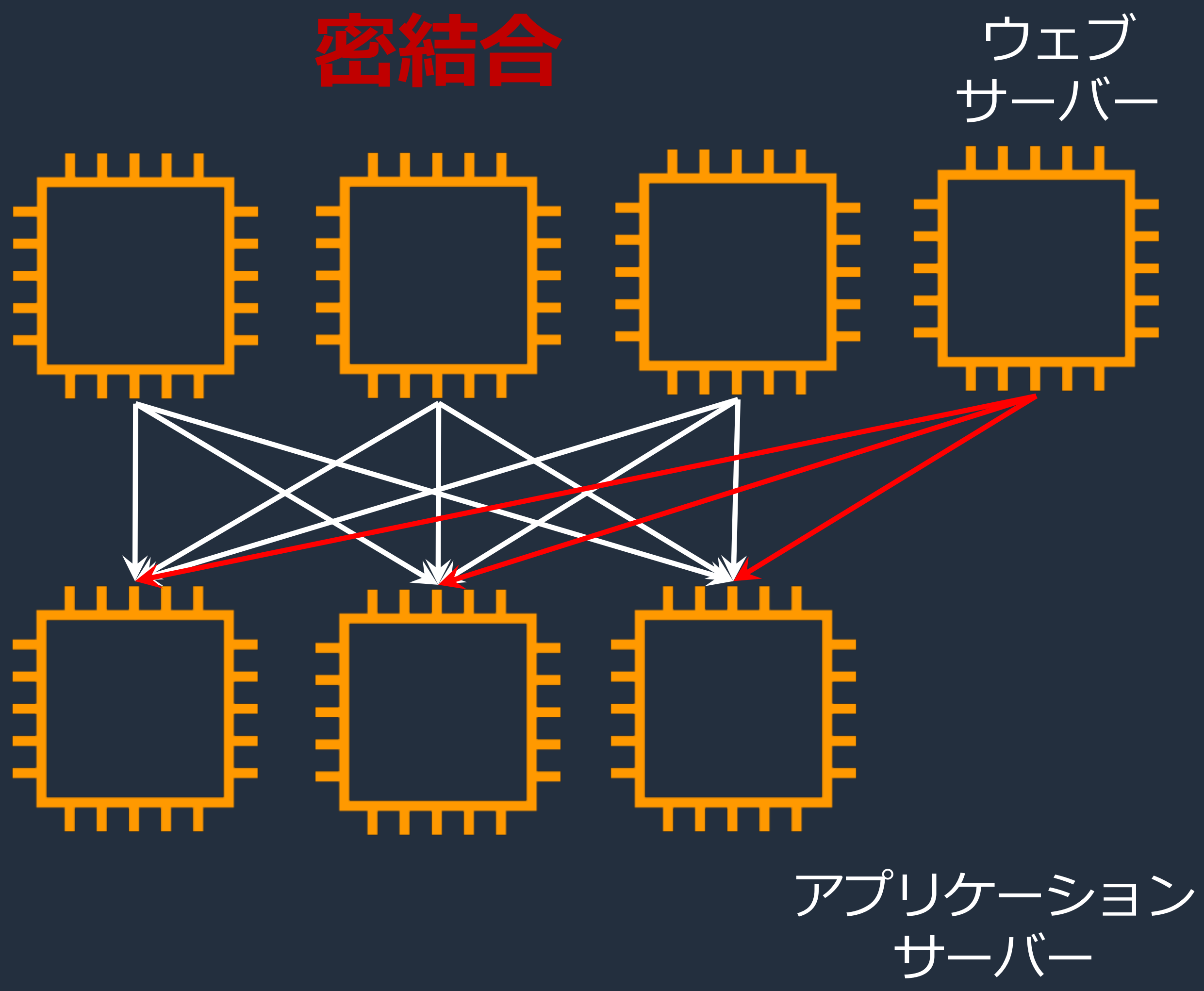
疎結合と密結合の違い：スケーラビリティ

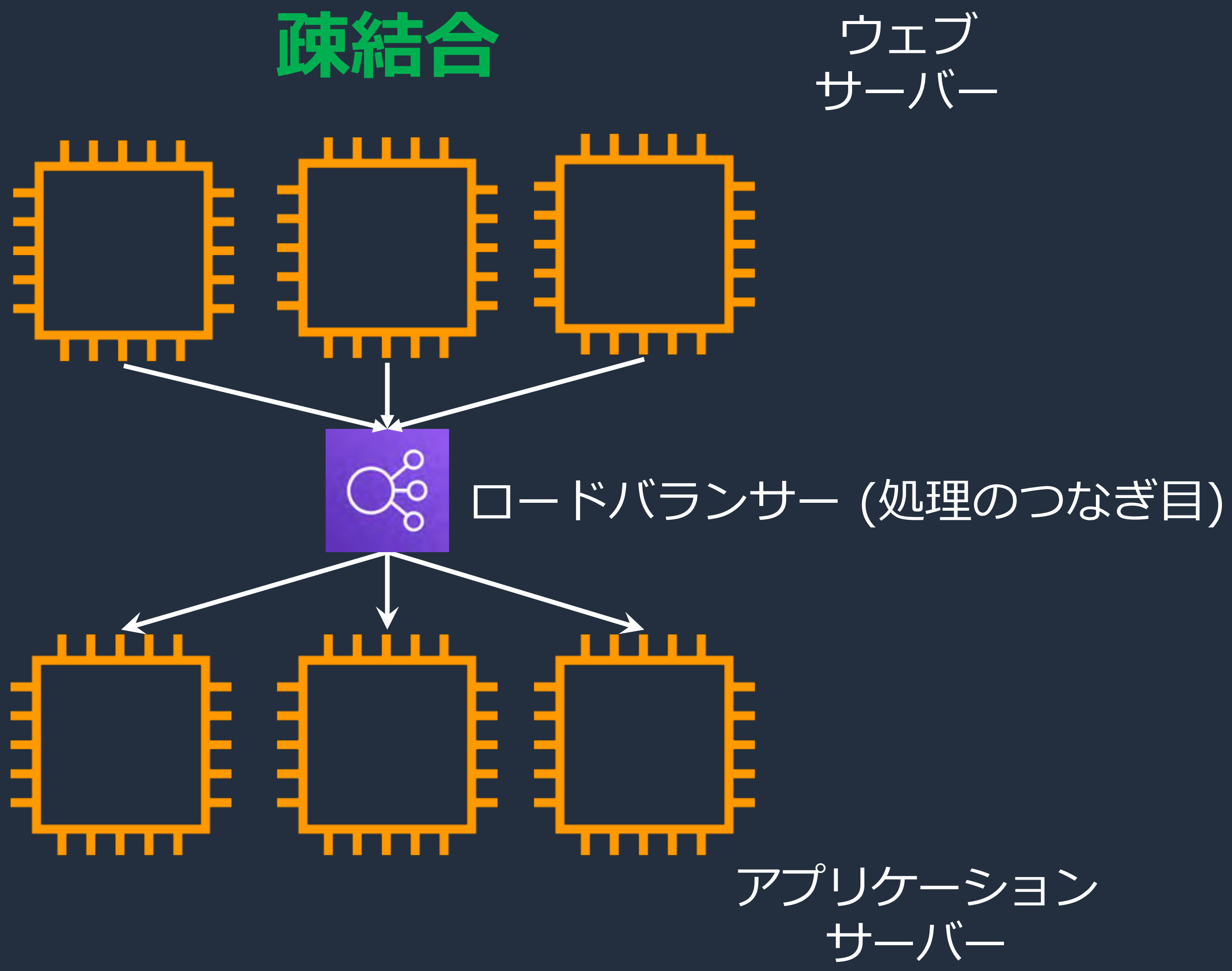
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



疎結合と密結合の違い：スケーラビリティ

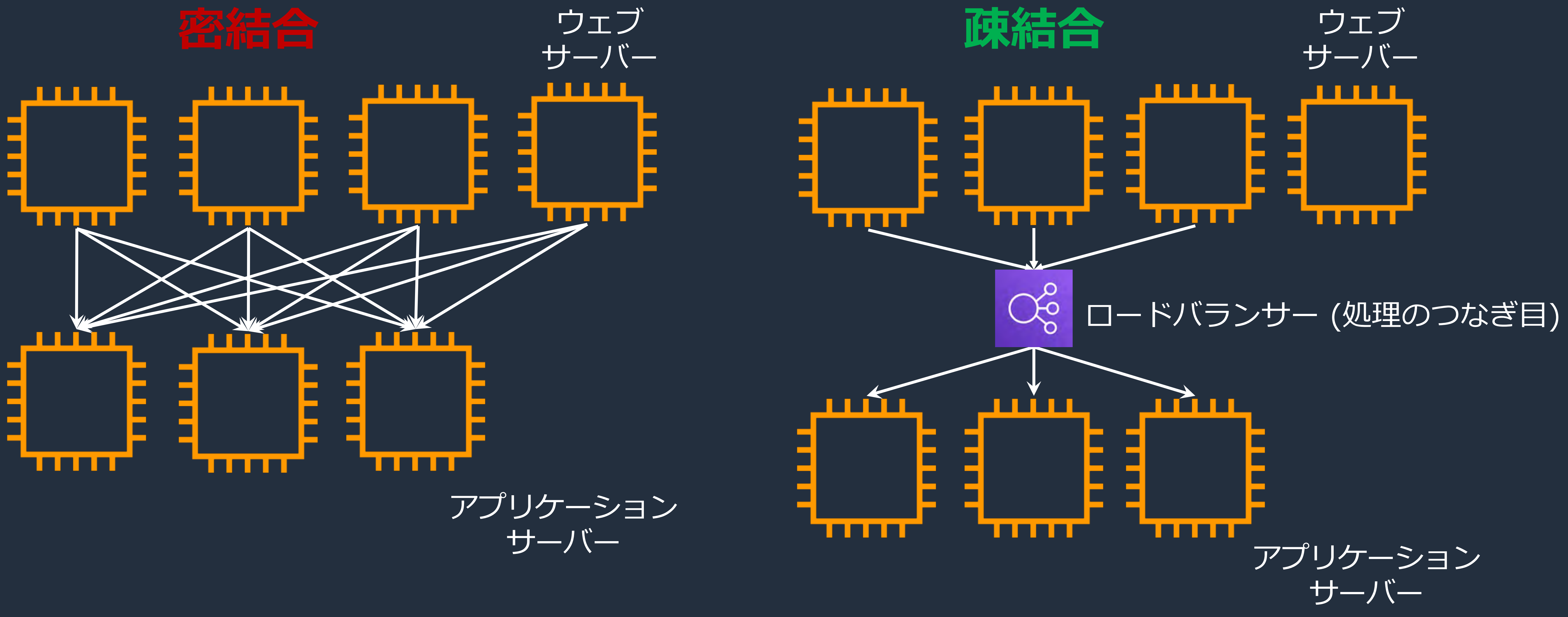
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理





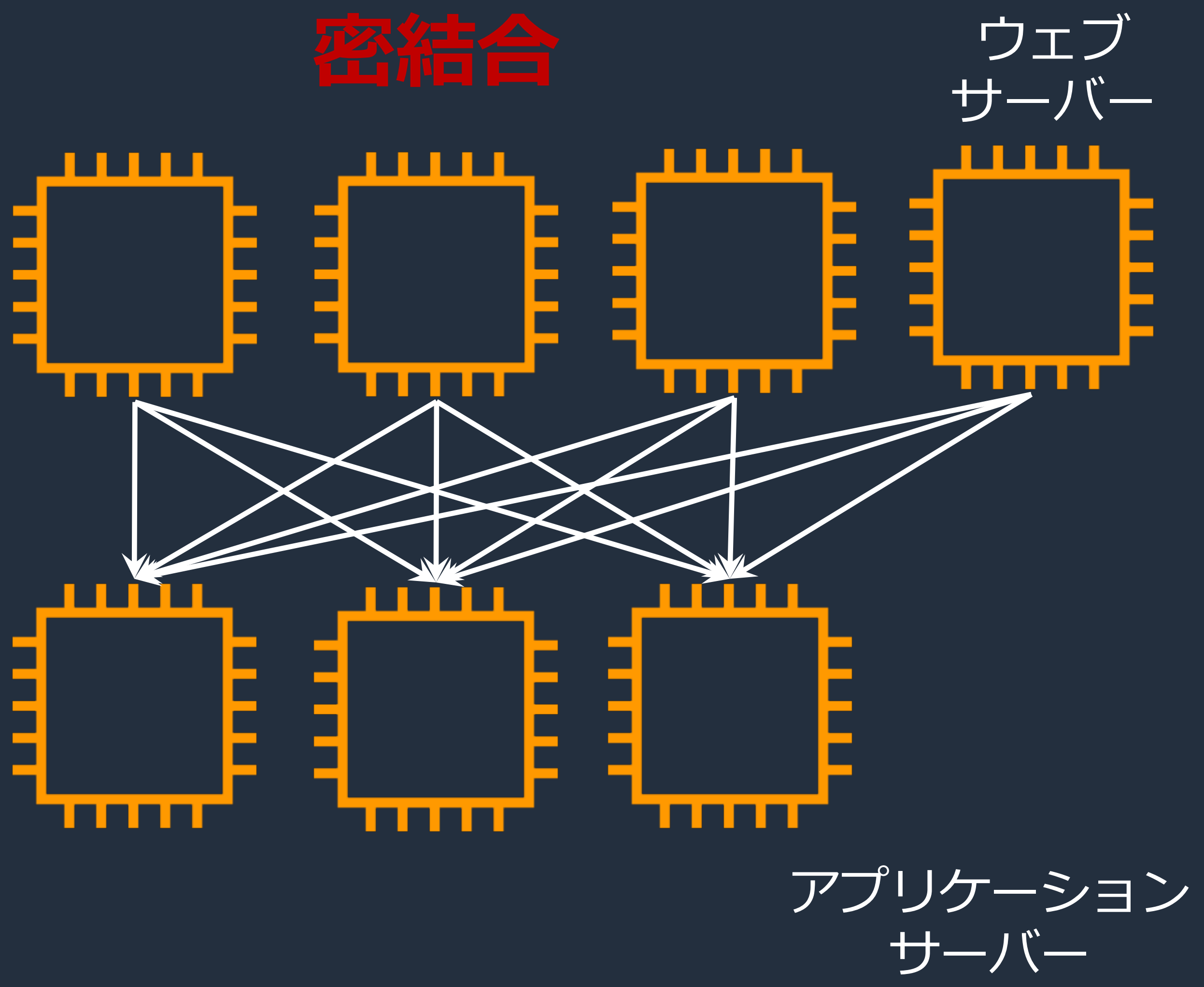
疎結合と密結合の違い：スケーラビリティ

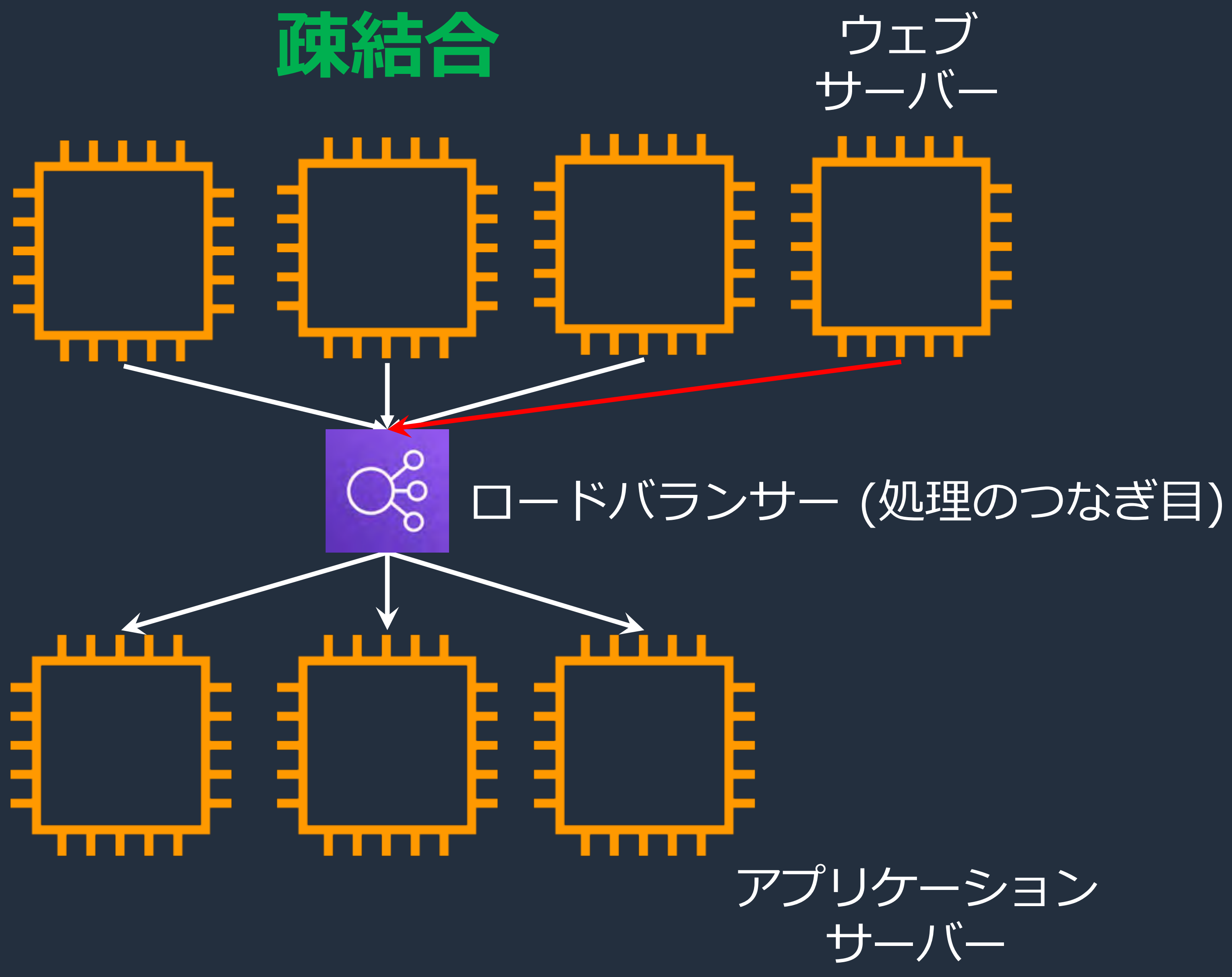
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



疎結合と密結合の違い：スケーラビリティ

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

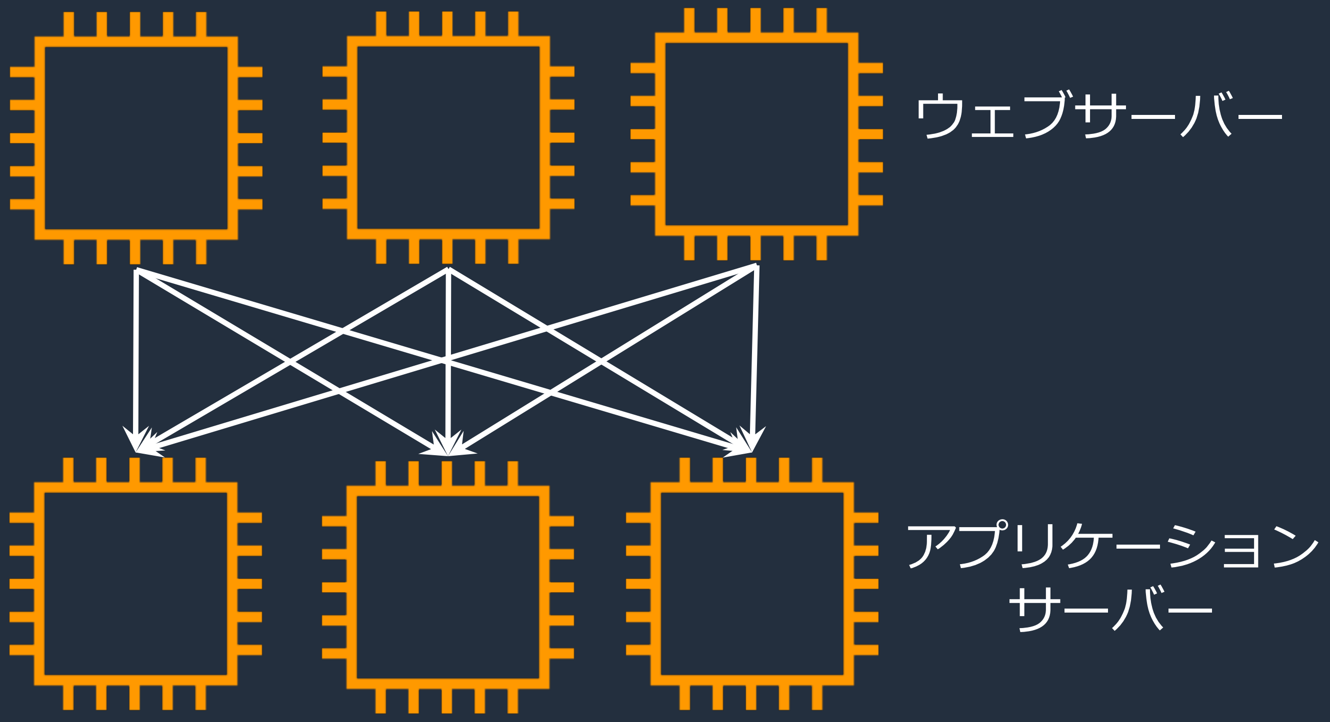




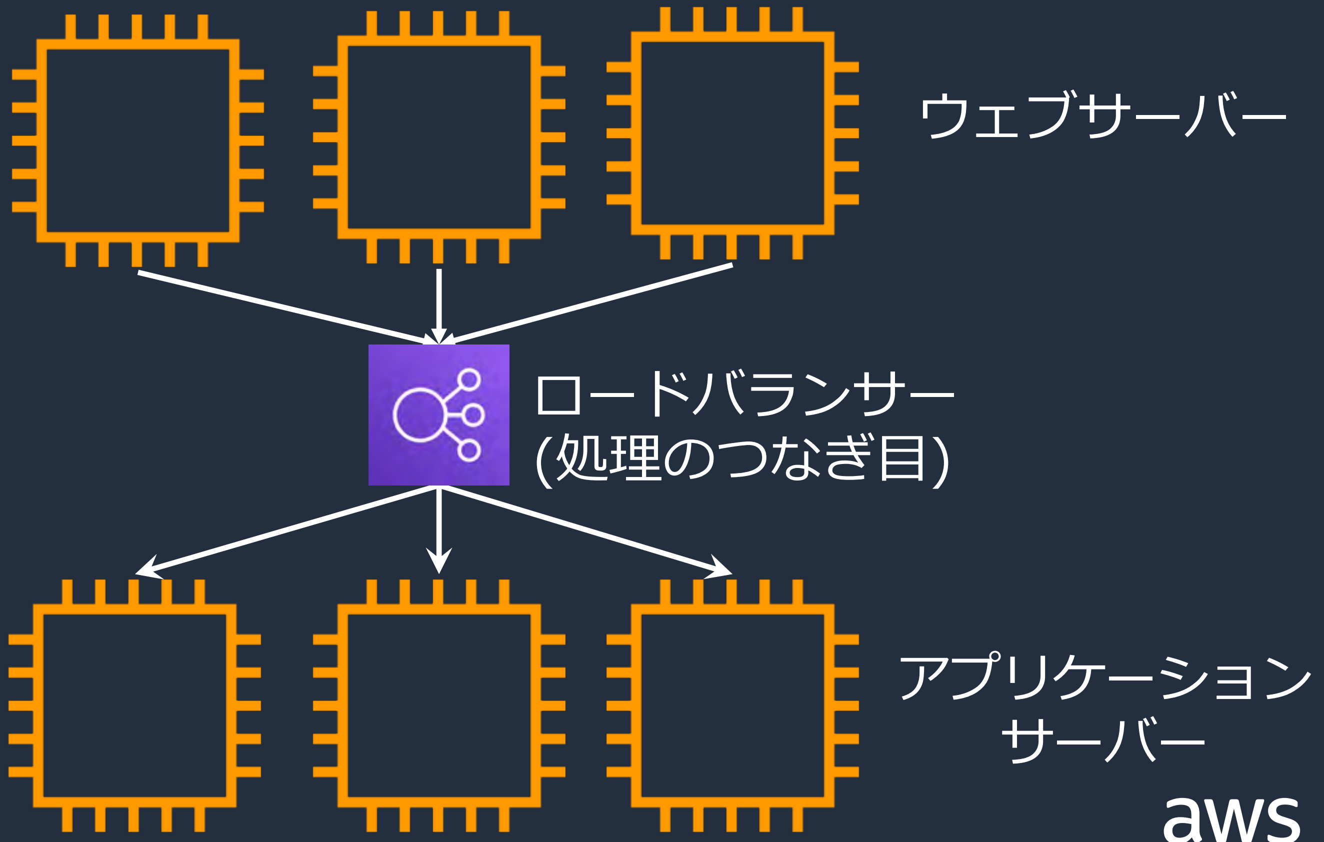
コンポーネントを疎結合にする

互いに独立したコンポーネントを使ってアーキテクチャを設計する

アンチパターン



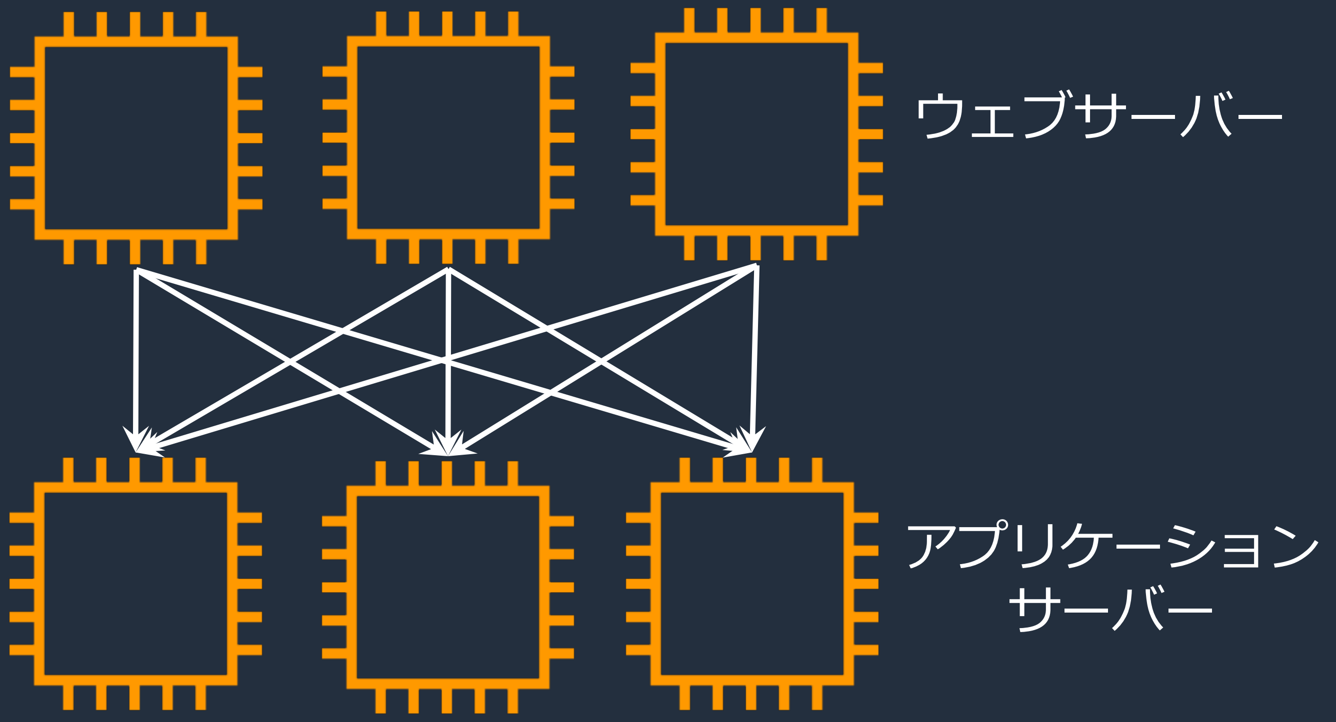
ベストプラクティス



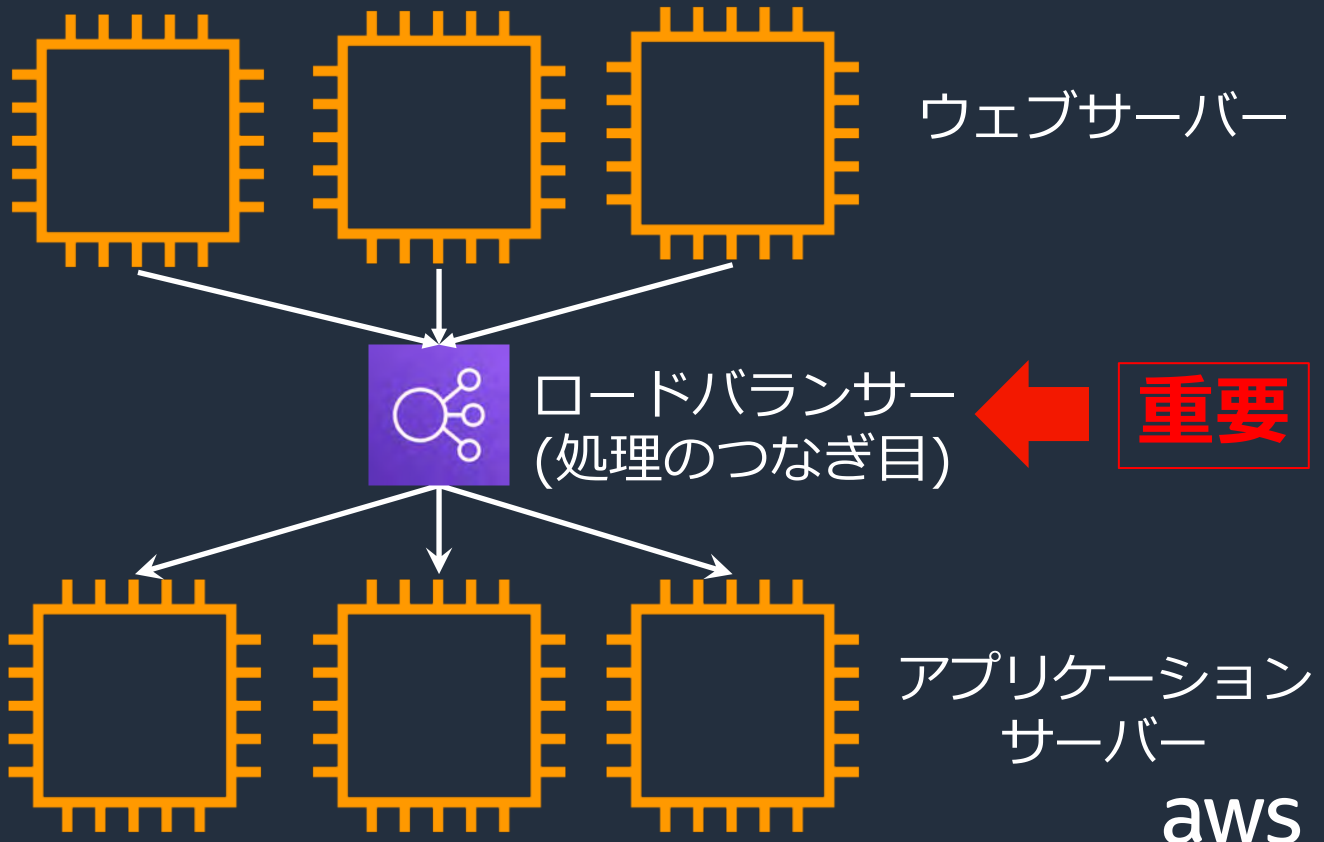
コンポーネントを疎結合にする

互いに独立したコンポーネントを使ってアーキテクチャを設計する

アンチパターン



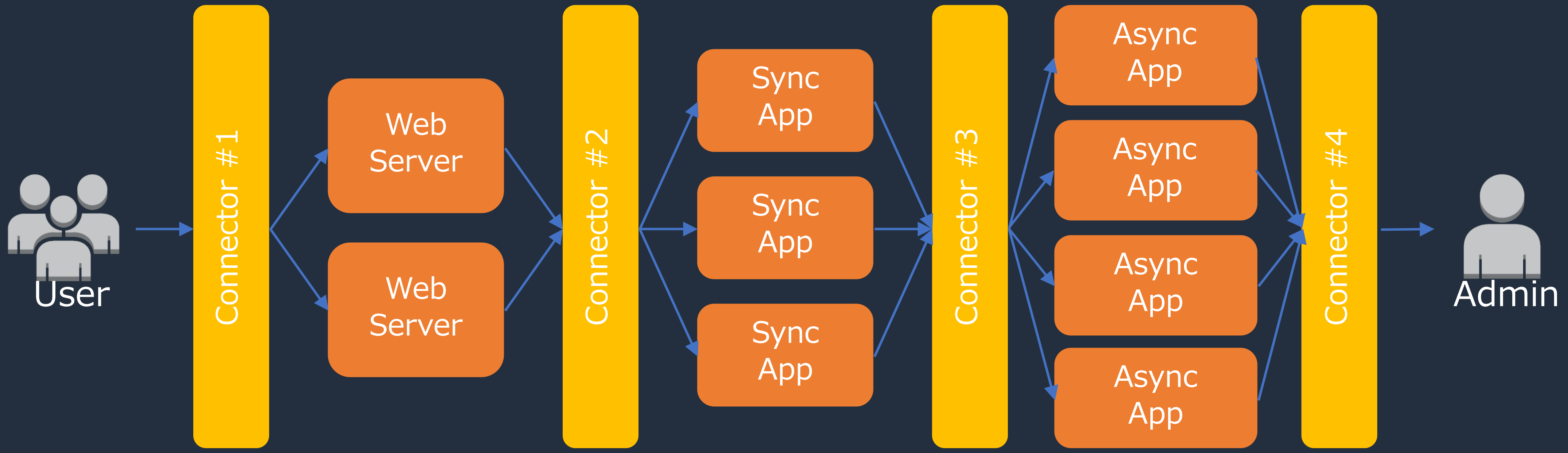
ベストプラクティス



疎結合アプローチ：処理のつなぎ目

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

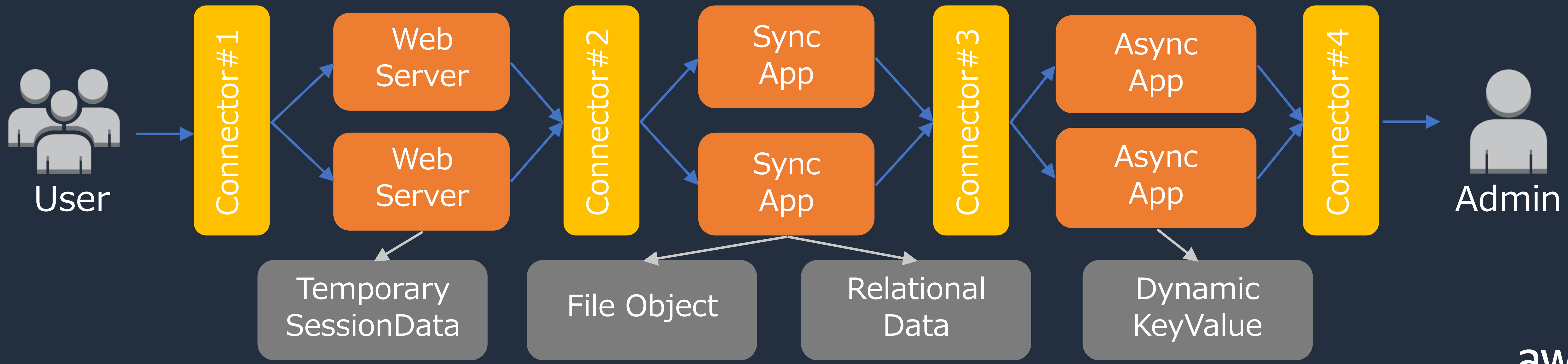
- 処理側の拡張性を考え、Many-to-Many に繋げることが前提
- DNS、ロードバランサー、Message キュー、Event 通知など
- つなぎ目にスケーラビリティ、レジリエンシー（回復性）、 コスト効率が必要



疎結合アプローチ：処理のつなぎ目

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

- データは常にデータストアに格納し、処理側には持たせない
- データ特性ごとに最適なデータストアを選択する
    - データ形式：構造型/ キーバリュー型 / ファイルオブジェクト
    - 増減率・更新頻度：動的 / 静的
    - RW特性：読取重視 / 書込重視
- 耐久性：永続データ / 一時データ
- データストアにスケーラビリティ、レジリエンシー、コスト効率が必要



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

5. サーバーではなくサービスで設計する

マネージドサービスを活用する

- AWS におけるマネージドサービスとは・・・
  - 耐障害性や可用性が
組み込まれており、テクノロジーをサービスとして活用できる
  - 例えば、ロードバランサーが必要な場合、
ロードバランサーのマネージドサービス **Elastic Load Balancing** を
使うと、利用者はリクエストの負荷分散制御に集中でき、
ロードバランサーの運用管理やキャパシティコントロールは
AWS に任せる事ができる

(参考) サーバーではなくサービスで設計する

AWS のサービスの幅広さを活用する。
インフラストラクチャをサーバーに限定しない

アンチパターン

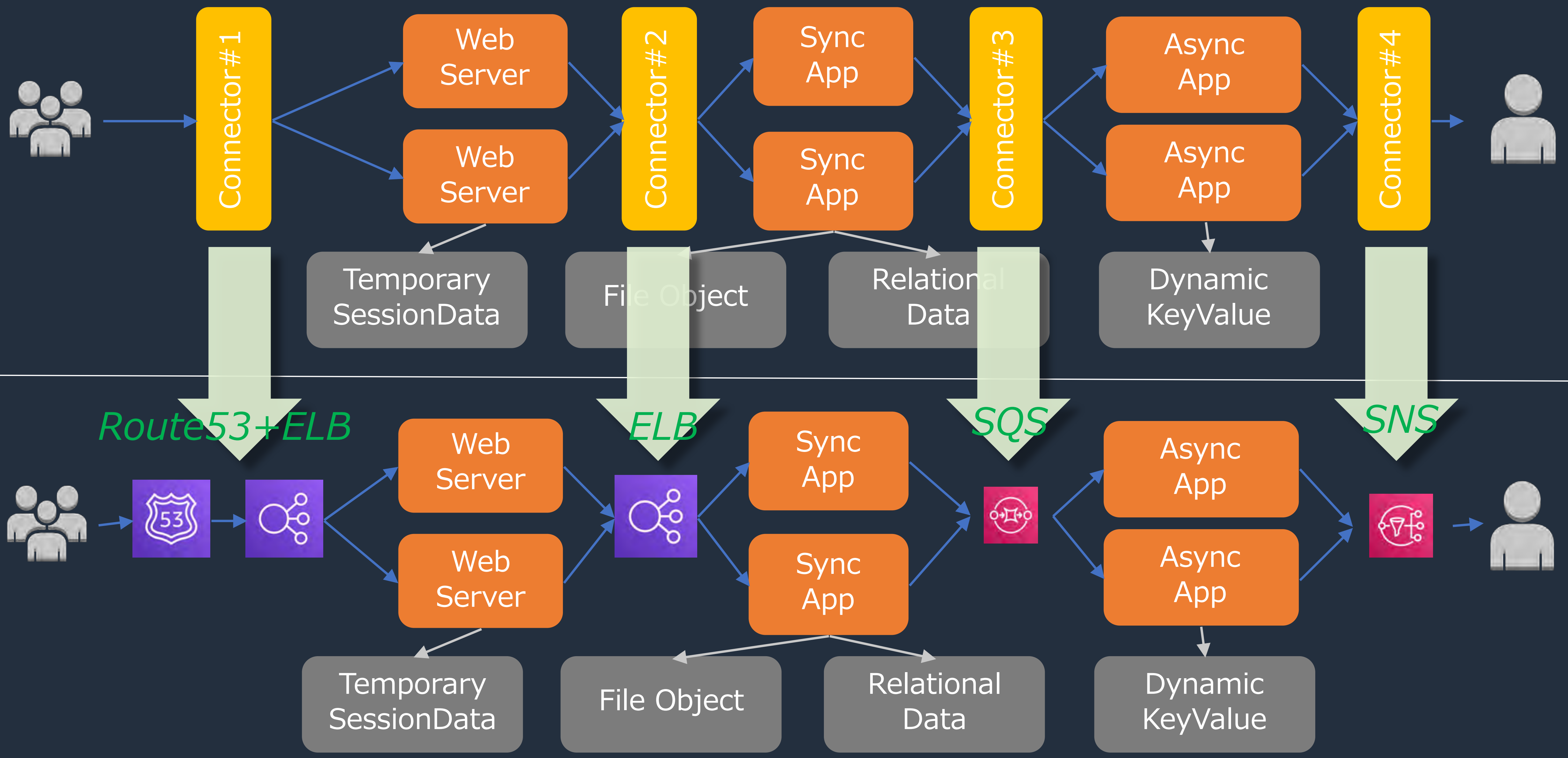
- シンプルなアプリケーションを永続的にサーバー上で実行する
- アプリケーション間で直接通信が行われる
- 静的ウェブアセットはローカルのインスタンスに保存される
- ユーザー認証とユーザー状態の保存は、バックエンドサーバーによって処理される

ベストプラクティス

- 必要に応じてサーバーレスソリューションをプロビジョニングする
- メッセージキューによりアプリケーション間の通信が処理される
- 静的ウェブアセットは Amazon S3 などの外部のデータストアに保存される
- ユーザー認証とユーザー状態の保存には AWS のマネージドサービスを使用する

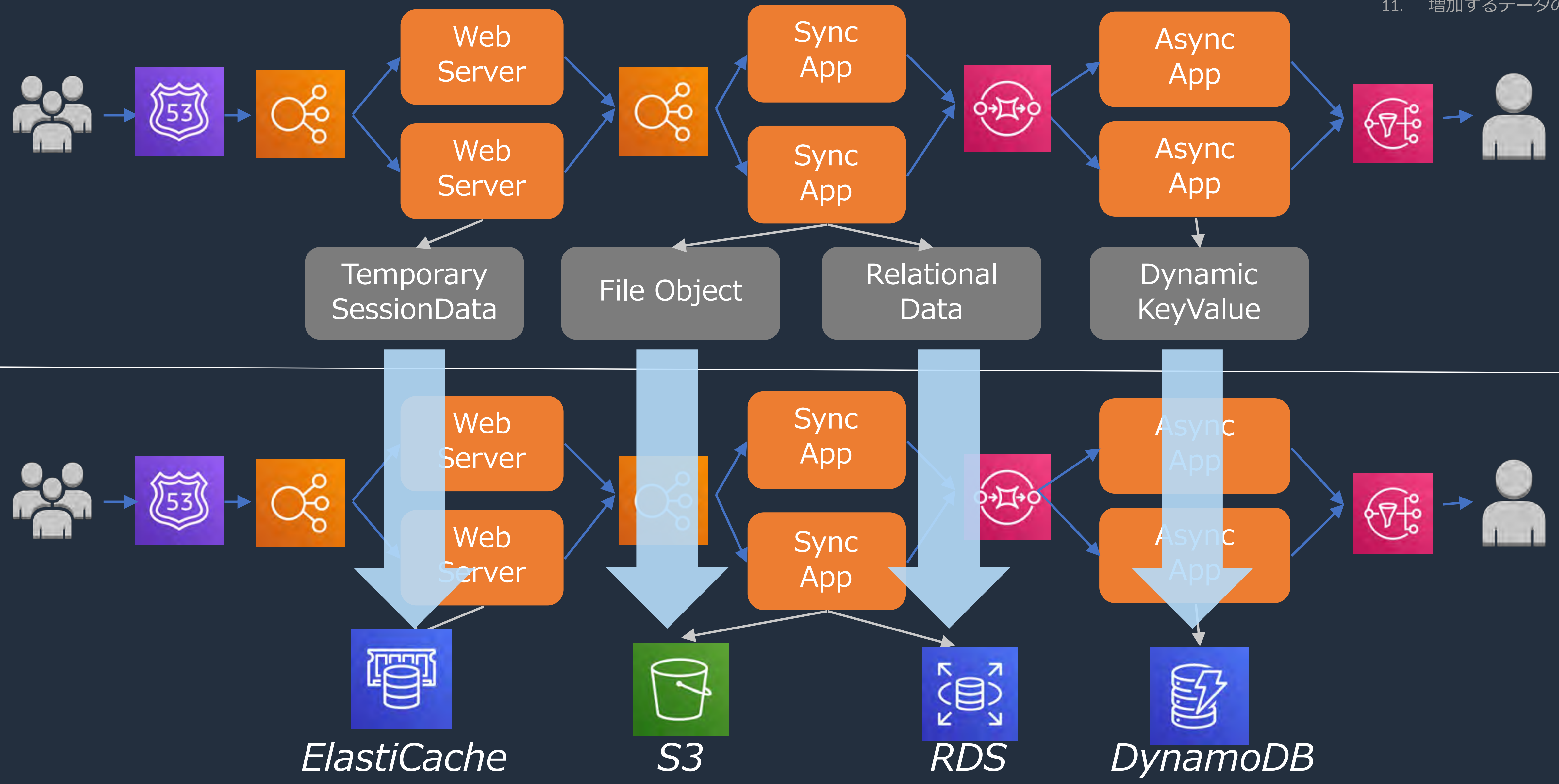
つなぎ目のマネージドサービス化

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



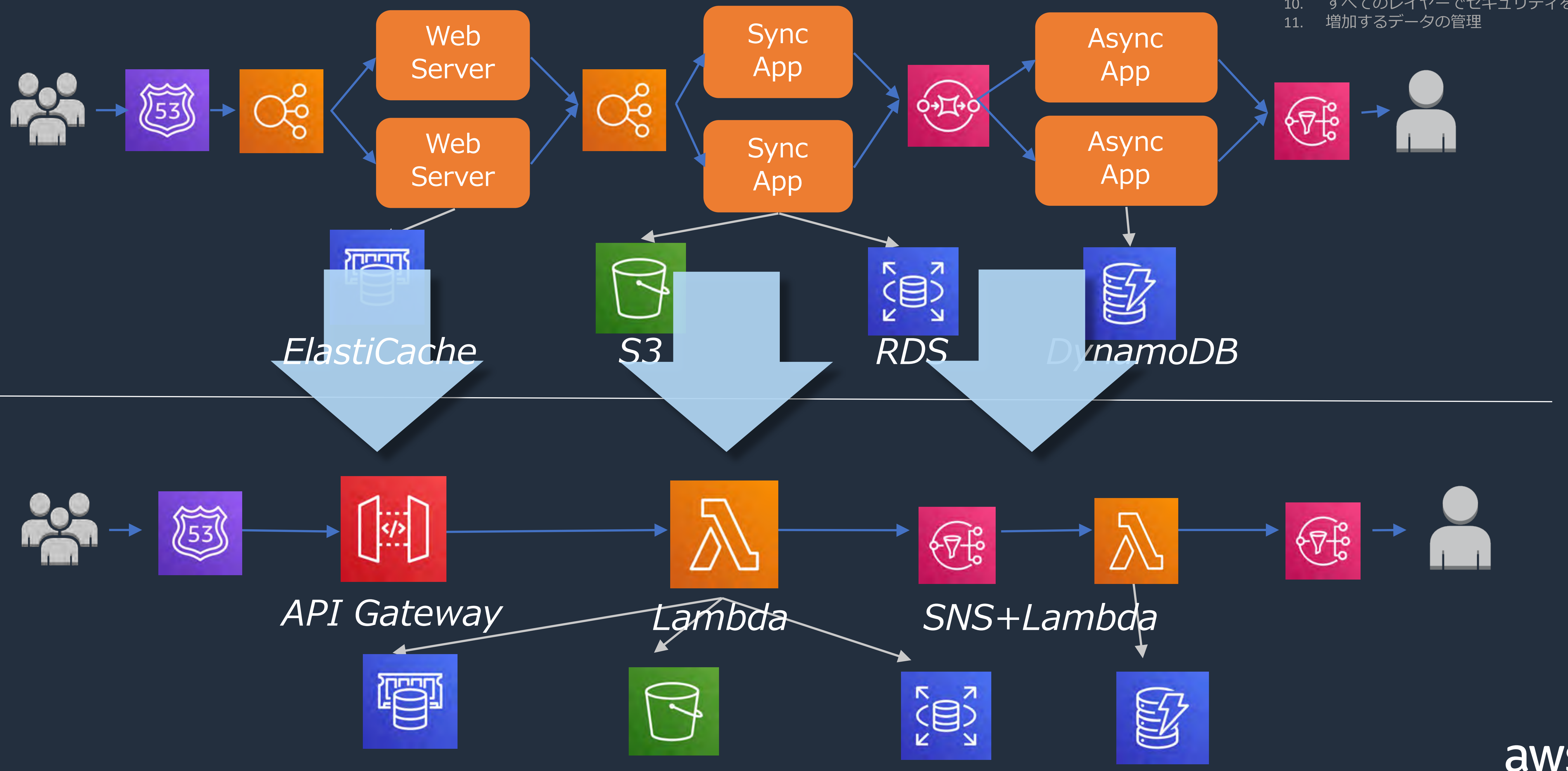
データストアのマネージドサービス化

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



API とコンピューティング リソースのマネージドサービス化

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

6.適切なデータベースソリューションを選択する

適切なデータベースソリューションを選択する

テクノロジーに基づいてワークロードを選択するのではなく、ワークロードに基づいてテクノロジーを選択する

検討事項

- 読み取りと書き込みのニーズ
- 必要なストレージの合計サイズ
- オブジェクトの標準的なサイズとオブジェクトへのアクセスの特性
- 耐久性の要件
- レイテンシーの要件
- サポートされる最大同時接続ユーザー数
- クエリの特性
- 必要とされる整合性統制の強度

データベース移行戦略 – 例

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

- パフォーマンス
- スケーラビリティ
- 複雑性
- 機能

DB
タイプ

リレーショナル

- ストアドプロシージャ
- ライセンスの要件
- OS へのアクセス
- TLS エンドポイント

非リレーショナル

- キーバリューストア
- 列指向ストア
- ドキュメント指向ストア
- グラフデータベース

データベース
メンテナンス

Amazon RDS

EC2 インスタンスの DB

Amazon Redshift

Amazon DynamoDB

Amazon ElastiCache

7.単一障害点をなくす

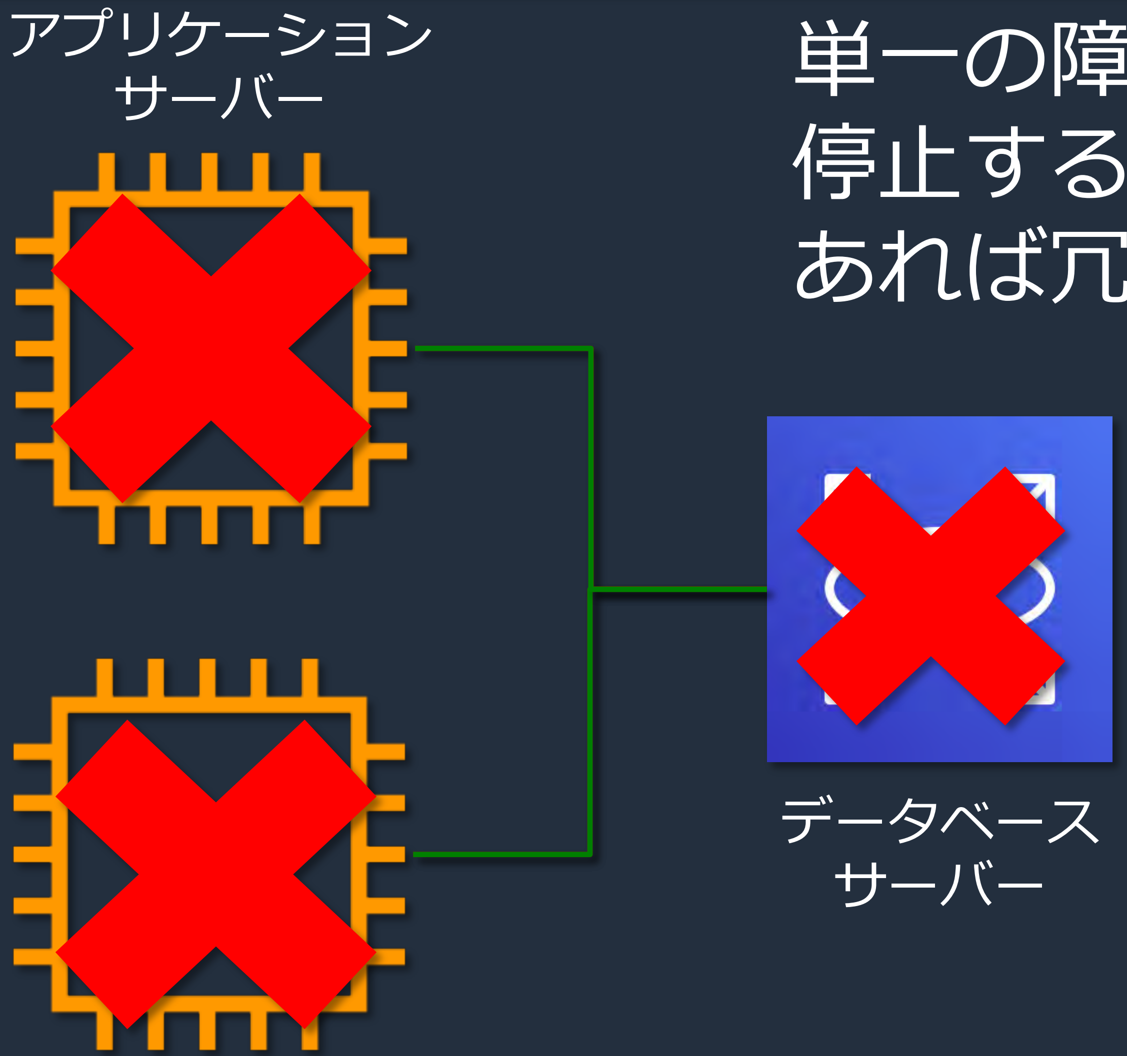
単一障害点とは

- 単一障害点とは・・・
  - ➢ その箇所が停止するとシステムの全体が停止するような箇所
  - ➢ SPOF (Single Point of Failure) とも呼ばれる

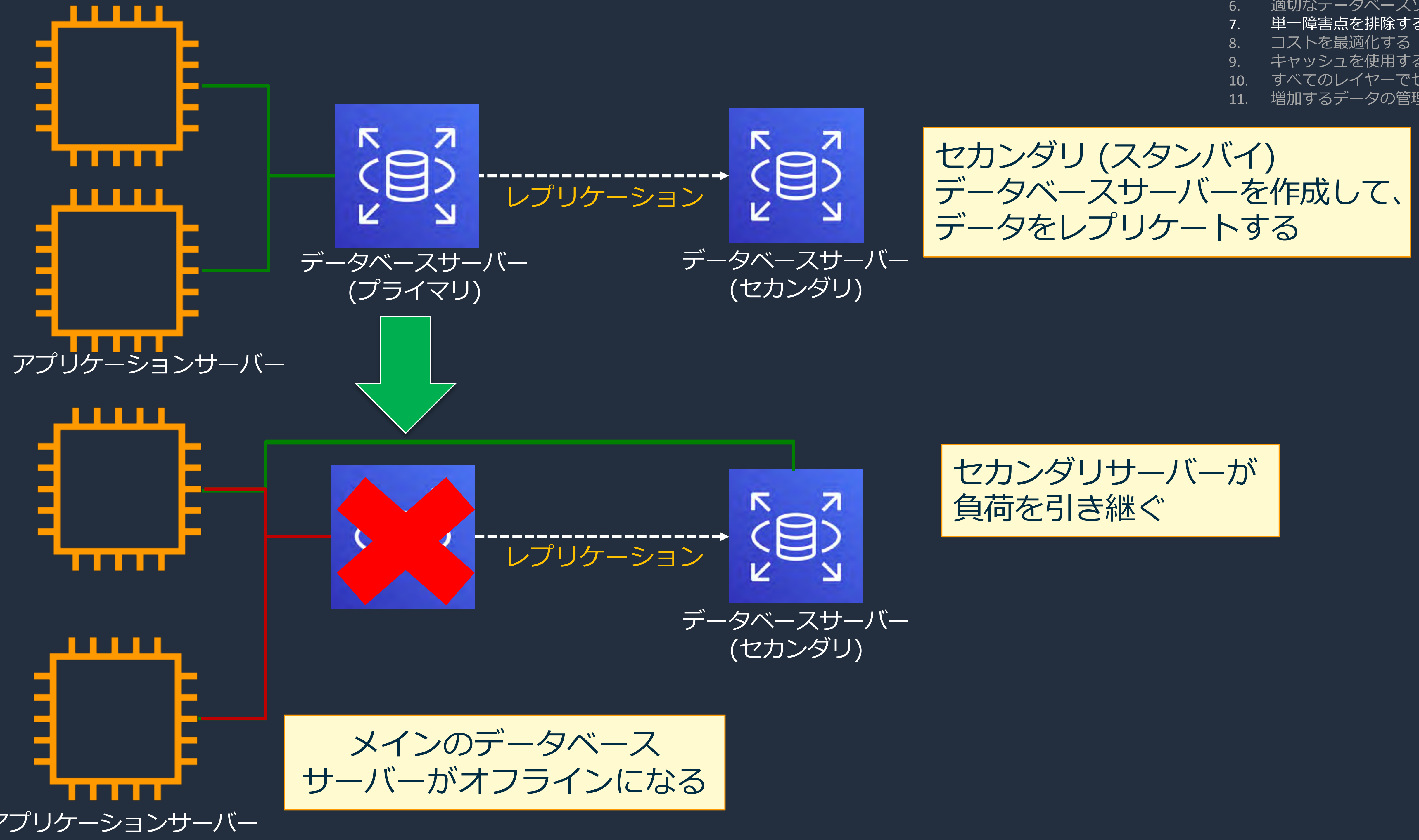
単一障害点をなくす：アンチパターン

すべてのものは故障するものと考え
逆算して設計する

単一の障害によってシステム全体が停止するのを避けるため、可能であれば冗長性を実装する



単一障害点をなくす：ベストプラクティス



8.コストを最適化する

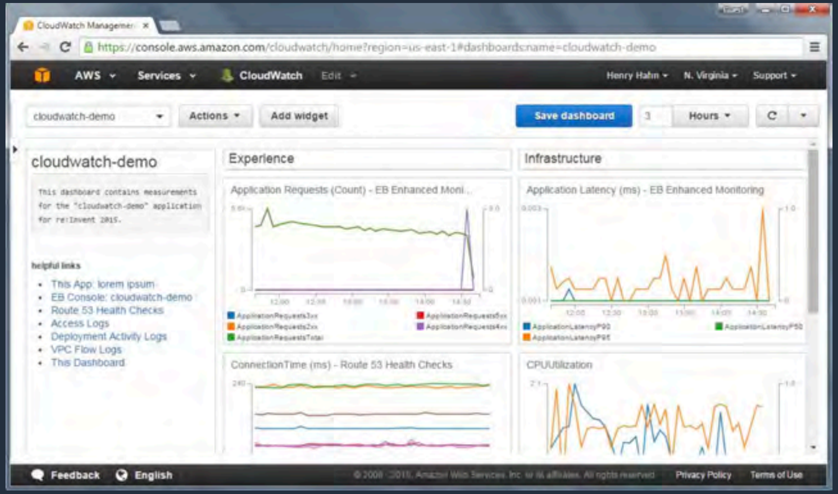
AWS のコスト特性

- 初期投資が不要
  - ➢ データセンターの運用と保守への投資が不要に
- 従量課金 (変動費)
  - ➢ リソースを使用したときに、使用した分だけ支払う
- TCO (Total Cost of Ownership) で考える
  - ➢ TCO とは・・・総所有コストのことで、ある設備などの資産に関する、購入から廃棄までに必要な時間と支出の総計
  - ➢ システムや資産の維持管理のために、必要なランニングコスト (人件費、運用保守費用、セキュリティ機能等) も考慮する

モニタリングを検討する

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

- リソースモニタリング
  - ➢ 割り当てたリソースのサイズは適切か
  - ➢ どの程度使用されているか



- コストモニタリング
  - ➢ コストエクスプローラー等を活用し、日々の利用料金を把握する

※コストエクスプローラとは・・・
毎月または毎日のコストを、
7 日間から最大 1 年間表示可能なコスト可視化ツール

コストを最適化する

AWS の柔軟なプラットフォームを活用して、コスト効率を上げる

検討事項

- リソースのサイズとタイプがジョブに適しているか
- どのメトリックスをモニタリングする必要があるか
- 使用されていないリソースがオフになっていることをどのような方法で確認するか
- リソースの使用頻度はどの程度か
- サーバーのいずれかをマネージドサービスに置き換えられるか

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

9.キャッシュを使用する

キャッシュとは

- キャッシュとは・・・
  - ➢ 一度アクセスしたデータを
一時的に保管し、次回アクセス時よりも高速にアクセスする仕組み
  - ➢ Web アプリケーションにおけるキャッシュ例
    - CDN (Contents Delivery Network)
      - クライアント (アクセス元) に近いネットワークに、コンテンツのコピーを配布するエッジサーバーを用意し、クライアントがコンテンツにアクセスするときには、最も近いエッジサーバーへ誘導する
  - ➢ システム内部におけるキャッシュ例
    - インメモリデータベース (Memcached, Redis 等)
      - データを Disk や SSD ではなく、メインメモリに保存するデータベース。メモリ上にすべてのデータが保存されているため、マイクロ秒レベルのアクセスを実現できる

キャッシュを使用する：CDN

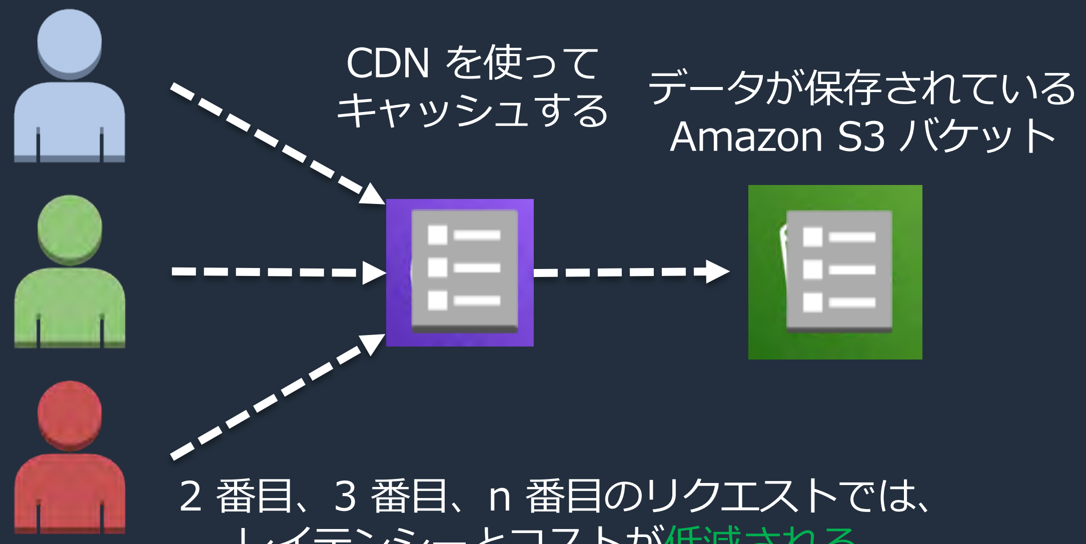
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

キャッシュを使用して、重複する
データ取り出しオペレーションを最小限にする

アンチパターン

3 件のリクエストでレイテンシー
とコストが同じである

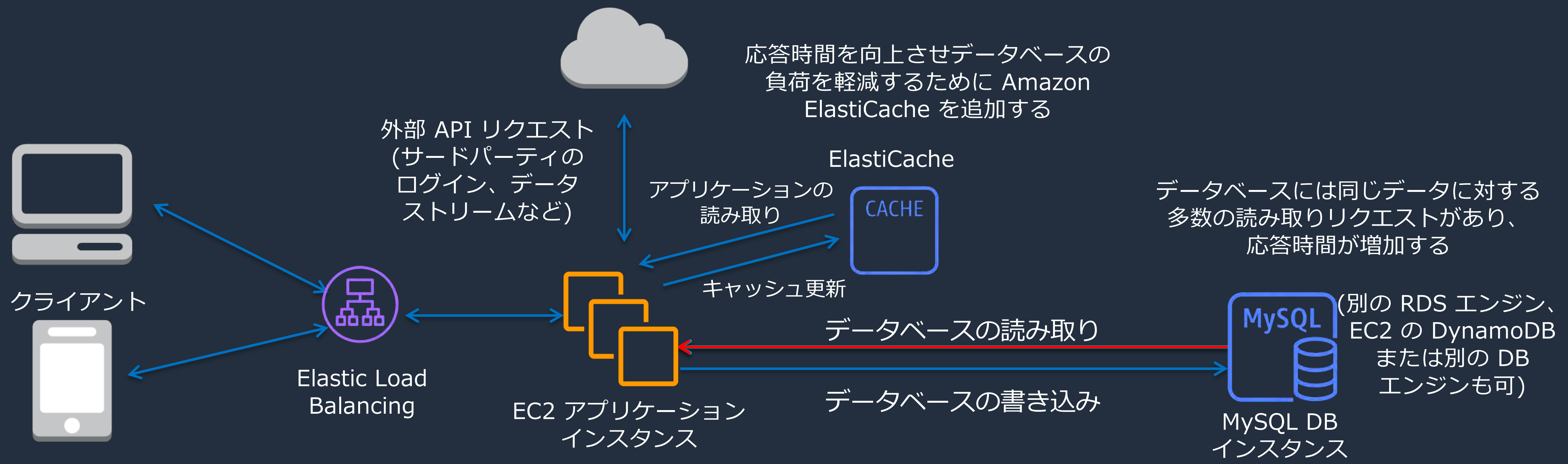
ベストプラクティス



2 番目、3 番目、n 番目のリクエストでは、
レイテンシーとコストが低減される

キャッシュを使用する：インメモリデータベース

キャッシュを使用して、重複する
データ取り出しオペレーションを最小限にする



1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

10. すべてのレイヤーでセキュリティを確保する

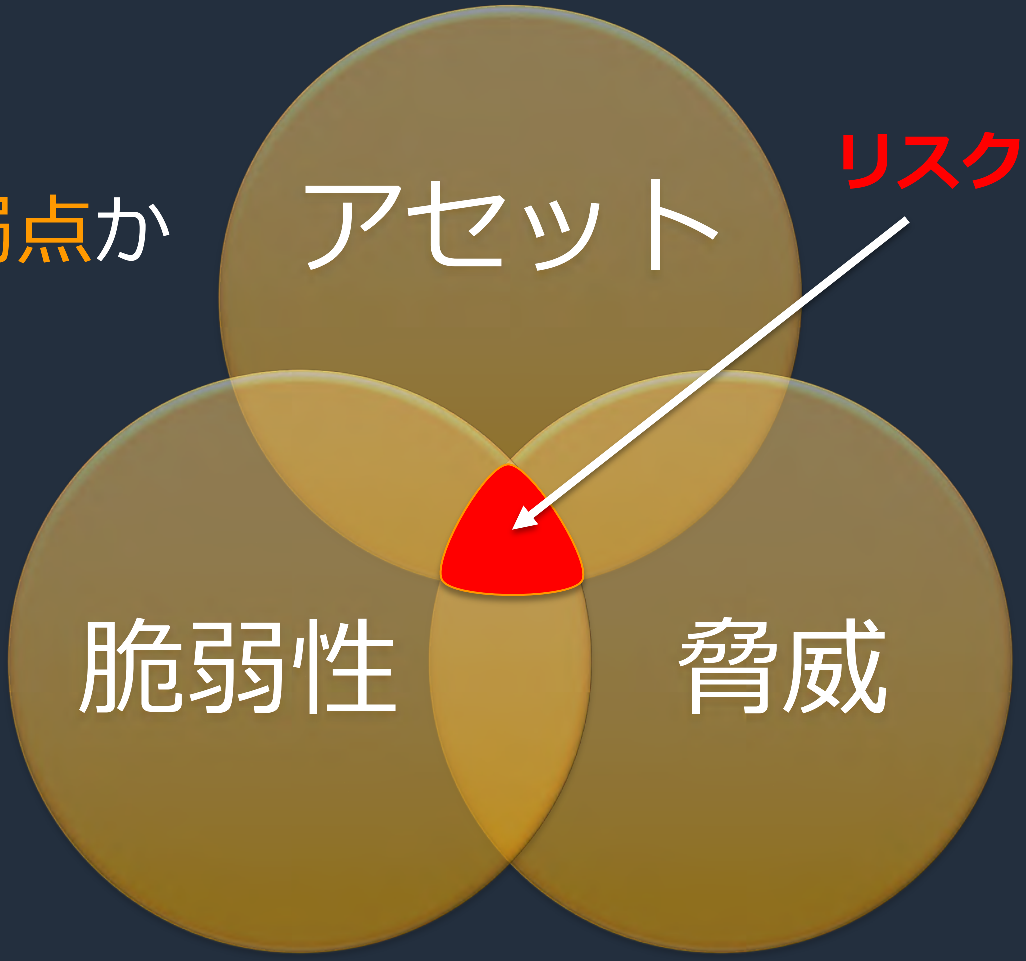
セキュリティの目的
ビジネスリスクの回避・継続性の担保

- セキュリティリスクの事前軽減
- セキュリティ被害時の回復力向上
- 経営・株主・顧客に対する説明責任
- 運用方法の統一化・運用コスト低減

リスク評価とトレードオフ

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

1. 何を保護しようとしているか
2. 何に対して保護する必要があるのか
   - それはサービスまたはシステムのどちらの弱点か
3. 適切に保護するために
   どのくらいの時間、労力、資金が必要か



責任共有モデルを理解する

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

お客様

クラウド内の
セキュリティに対する責任
SECURITY 'IN'THE CLOUD

お客様のデータ			
プラットフォーム、アプリケーション、IDとアクセス管理			
オペレーティングシステム、ネットワークとファイアウォール構成			
クライアント側データ暗号化 データ整合性認証	サーバー側暗号化 (ファイルシステムやデータ)	ネットワークトラフィック保護 (暗号化、整合性、アイデンティティ)	

AWS

クラウドの
セキュリティに対する責任
SECURITY 'OF'THE CLOUD

ソフトウェア			
コンピュート	ストレージ	データベース	ネットワーキング
ハードウェア/AWSグローバルインフラストラクチャー			
リージョン	アベイラビリティゾーン	エッジロケーション	

https://aws.amazon.com/jp/compliance/shared-responsibility-model/

すべてのレイヤーでセキュリティを確保する

**インフラストラクチャのすべての
レイヤーにセキュリティを構築する**

検討事項

- インフラストラクチャの各部分を隔離する
- 転送中および保管中のデータを暗号化する
- 最小権限の原則を使用して、きめ細かなアクセスコントロールを実施する
- 多要素認証を使用する
- マネージドサービスを活用する
- リソースへのアクセスを記録する
- デプロイを自動化して、セキュリティの整合性を維持する

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

11.増加するデータの管理

クラウドで変わるデータ活用：
データは加工せず全期間を残す

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

• これまで：

1. ディスクが高価で上限がある
2. データはサマリーだけ、もしくは期間限定で保存
3. 処理できる内容は固定的

インフラ管理者の仕事：
ストレージを溢れさせず、時間内に処理が終るようにサイズや処理内容を調整する

On AWSクラウド：

1. 安価・上限無しのストレージ
2. オリジナルデータを全て残す
3. 処理対象・処理内容はビジネスに合わせて変わる

インフラ管理者の仕事：
データを活用して新しい課題に素早く対応できるインフラを用意する。個別リクエストへの対応

増加するデータの管理

- AWS 上でデータ活用する際のベストプラクティス：**データレイク**

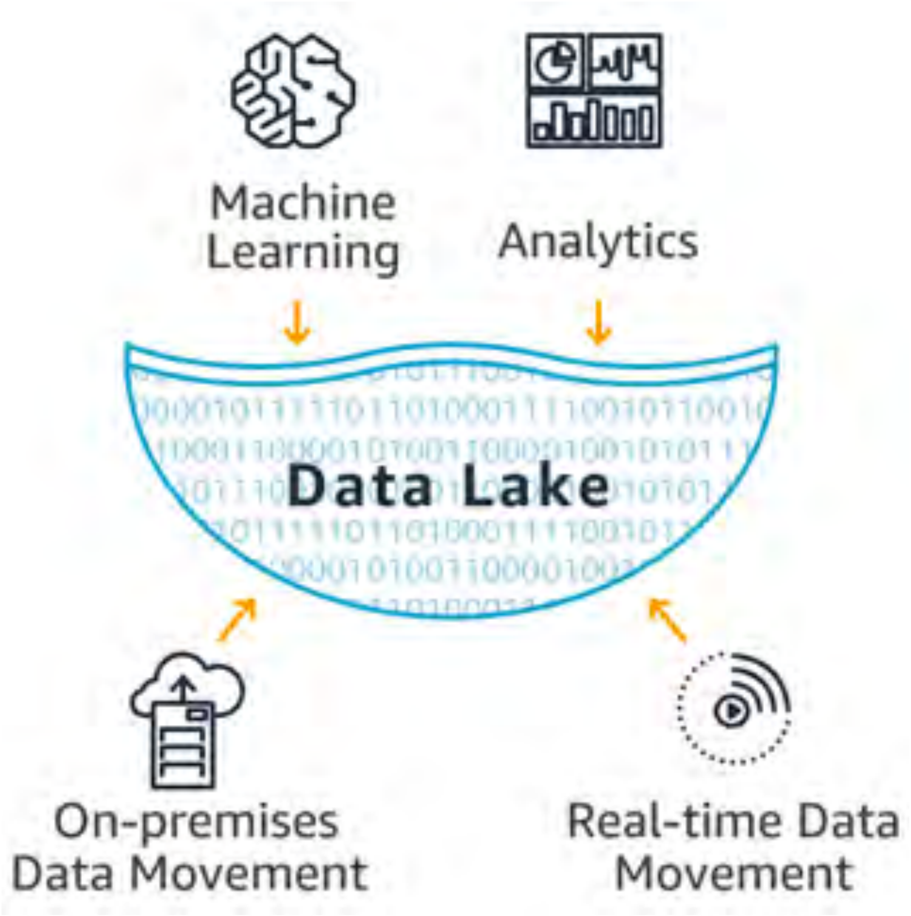
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

データレイクとは

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

AWSの公式ドキュメント
https://aws.amazon.com/jp/big-data/datalakes-and-analytics/what-is-a-data-lake/

データレイクは、規模にかかわらず、すべての構造化データと非構造化データを保存できる一元化されたリポジトリです。データをそのままの形で保存できるため、データを構造化しておく必要がありません。また、ダッシュボードや可視化、ビッグデータ処理、リアルタイム分析、機械学習など、さまざまなタイプの分析を実行し、的確な意思決定に役立てることができます。

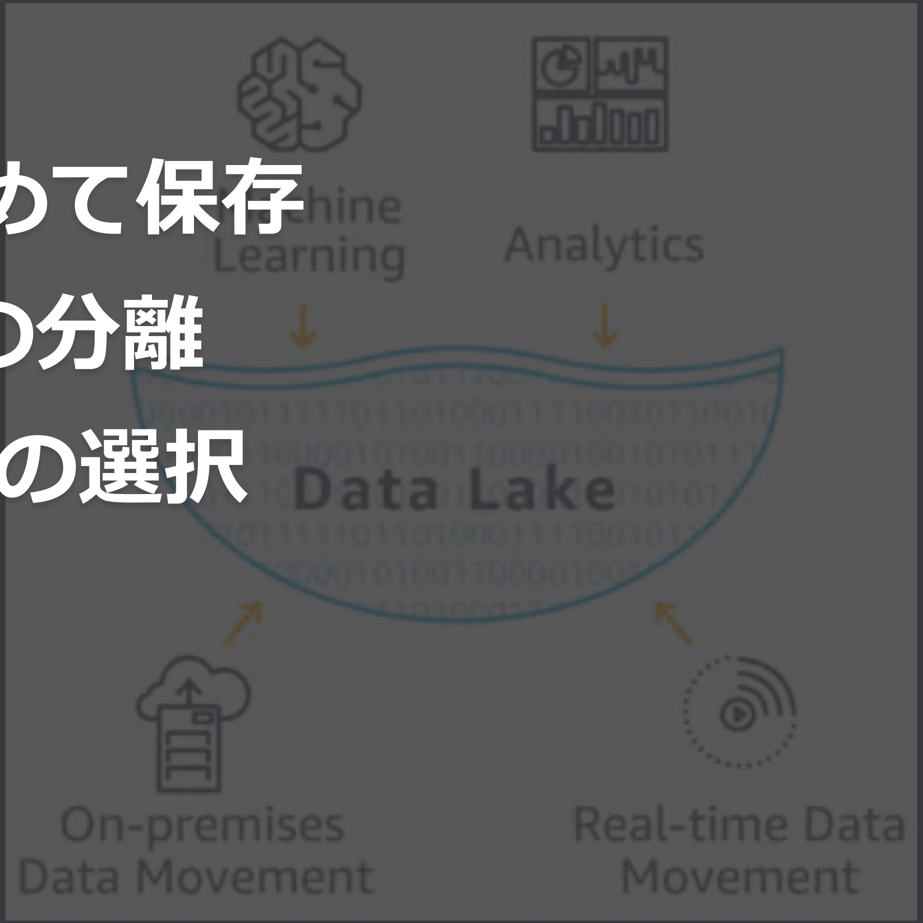


データレイクとは

AWSの公式ドキュメント
https://aws.amazon.com/jp/big-data/datalakes-and-analytics/what-is-a-data-lake/

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理

データレイクは、規模にかかわらず、すべての構造化データと非構造化データを保存できる一元化されたリポジトリです。データをそのままの形で保存できるため、データを構造化しておく必要がありません。また、[illegible]可視化、ビッグデータ処理、リアルタイム分析、機械学習など、さまざまなタイプの分析を実行し、的確な意思決定に役立てることができます。



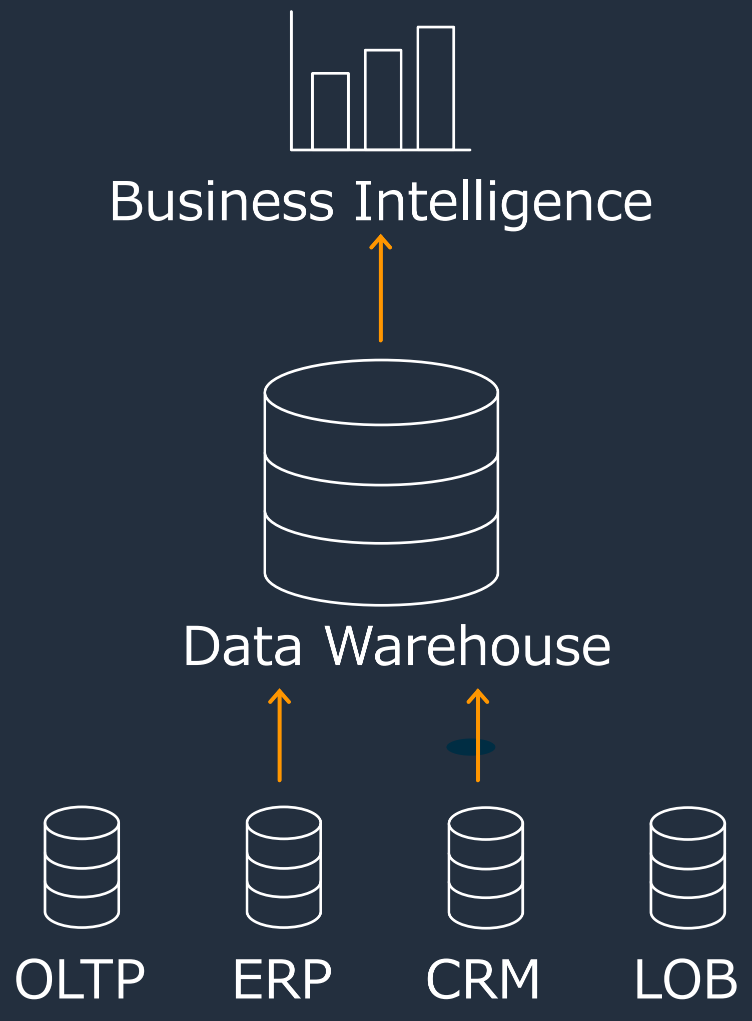
すべてのデータを 1 ヶ所に集めて保存

データストアとデータ処理の分離

用途に応じた適切な処理方法の選択

従来までの分析アプローチによくある姿

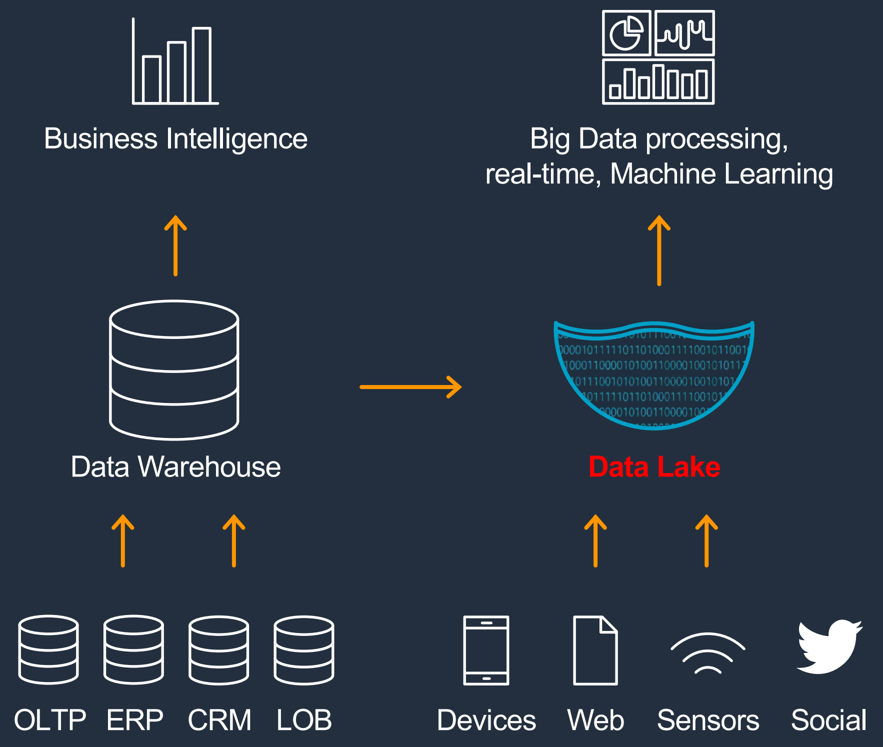
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



- リレーショナルデータ
- TBs-PBs スケール
- データロード前のスキーマ定義
- 運用レポート作成とアドホック操作
- 大きな初期設備投資額に加え、TBあたり年間およそ10万-50万ドルのランニングコスト

従来のアプローチを拡張するデータレイク

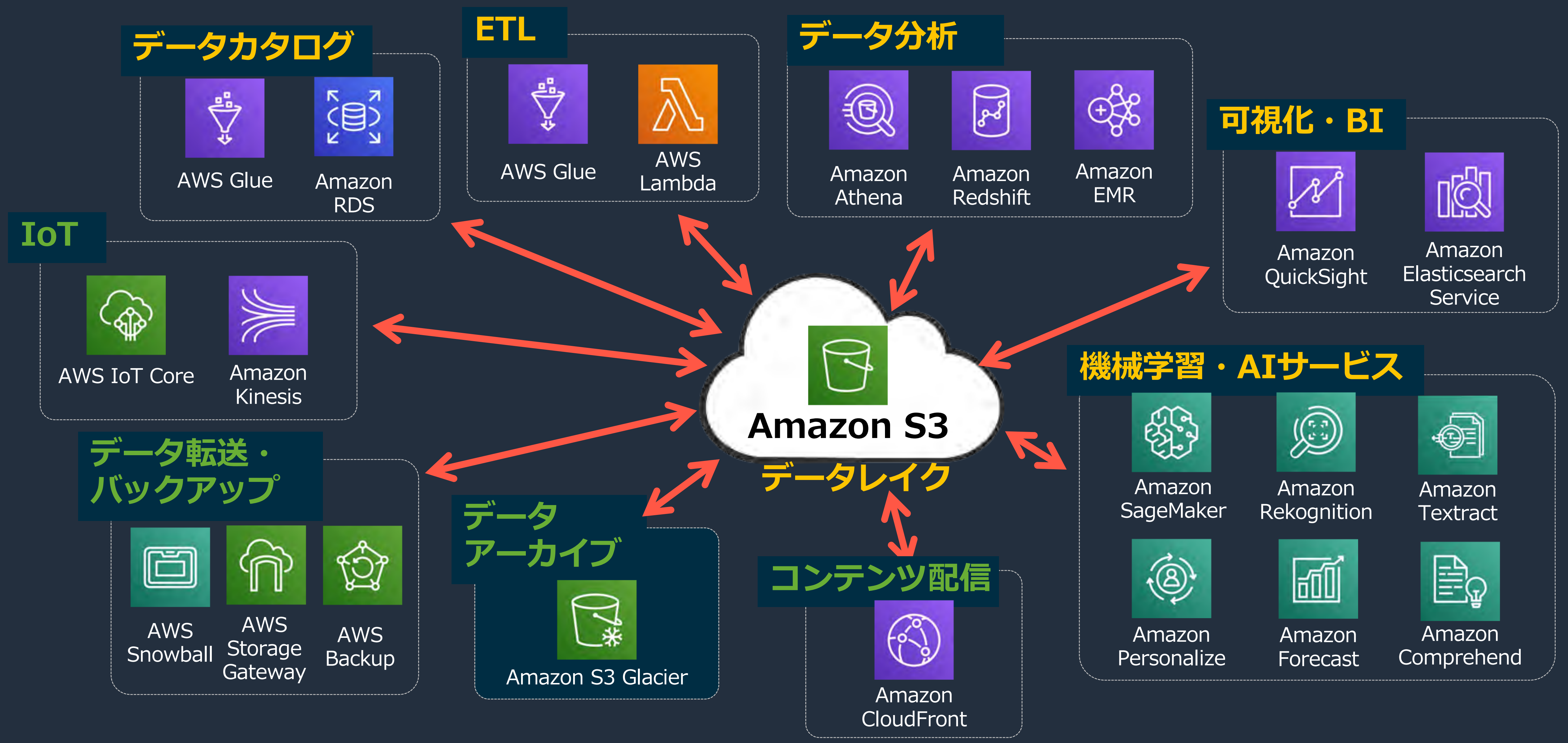
1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



- リレーショナルデータに加えて、非リレーショナルデータ
- TBs-EBs スケール
- 多様な分析エンジン
- 低コストのストレージと分析

AWS におけるデータレイク = Amazon S3

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切なデータベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべてのレイヤーでセキュリティを確保する
11. 増加するデータの管理



まとめ

まとめ : AWS 設計のベストプラクティスで 最低限知っておくべき 10 (+1) のこと

1. スケーラビリティを確保する
2. 環境を自動化する
3. 使い捨て可能なリソースを使用する
4. コンポーネントを疎結合にする
5. サーバーではなくサービスで設計する
6. 適切な
   データベースソリューションを選択する
7. 単一障害点を排除する
8. コストを最適化する
9. キャッシュを使用する
10. すべての
    レイヤーでセキュリティを確保する
11. 増加するデータの管理

Thank You !