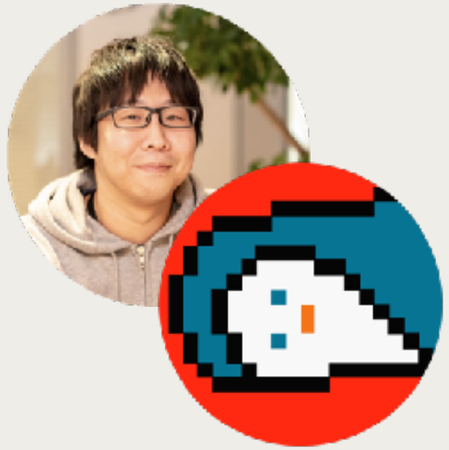
AWSのマネージドサービスを活かした Kubernetes 運用と Amazon EKS によるクラスタのシングルテナント戦略について

AWS Summit Tokyo 2019

freee 株式会社



freee株式会社
SRE

Kosuke Adachi
@foostan

- 2015年12月 ~ freeeに入社
- 2018年11月 ~ SRE

- 3年ぐらいフロントエンドとサーバサイドの開発
- SREに移ってからEKS移行やマルチクラスタデプロイツールの開発など
- 趣味
  - 自動化
  - キーボード制作/設計

質問 ✋

K8s本番で使っているよ ✋

EKS本番で使っているよ ✋

本日お話すること

インフラリソースのコード化と
Kubernetesのシングルテナント化でサービスの運用コストを分散させる

サービスの運用コストを分散させる？

サービス規模が拡大、サービス数が増加、開発者が増加

⬇

SREがボトルネックに

- 強い権限を持っているので何でも屋になりがち
- SREに問い合わせが集中
- 目先のタスクに追われる日々
- SREの人数はなかなか増えない

⬇

開発者チームにサービスの運用をおまかせする

おまかせする内容

・インフラ構築
・Kubernetesクラスタ構築
・アプリケーションデプロイ
・サービス監視
・アラート対応
など、基本的にサービス運用に必要なことすべて

開発チームだけでサービス運用の殆どをまかなえるような基盤づくりをSREが行う

Overview

01　freee について

02　インフラリソースのコード化

03　シングルテナントで権限を分離してクラスタの運用をおまかせする

04　EKSをマネージドサービスと組み合わせてクラスタの運用コストを抑える

05　マルチテナントからシングルテナントなEKSに移行した実例

Section

01

freee について

freeeについて

MISSION

生産年齢人口が劇的に減少し、慢性的な人手不足となる日本で労働生産性向上は緊急の課題となっています

freeeは「人工知能」と「統合基幹業務システム」をクラウド技術を活用し、業務効率化のサポートを続けることで、中堅中小企業のバックオフィス業務効率化を目指しています

スモールビジネスを、

世界の主役に。

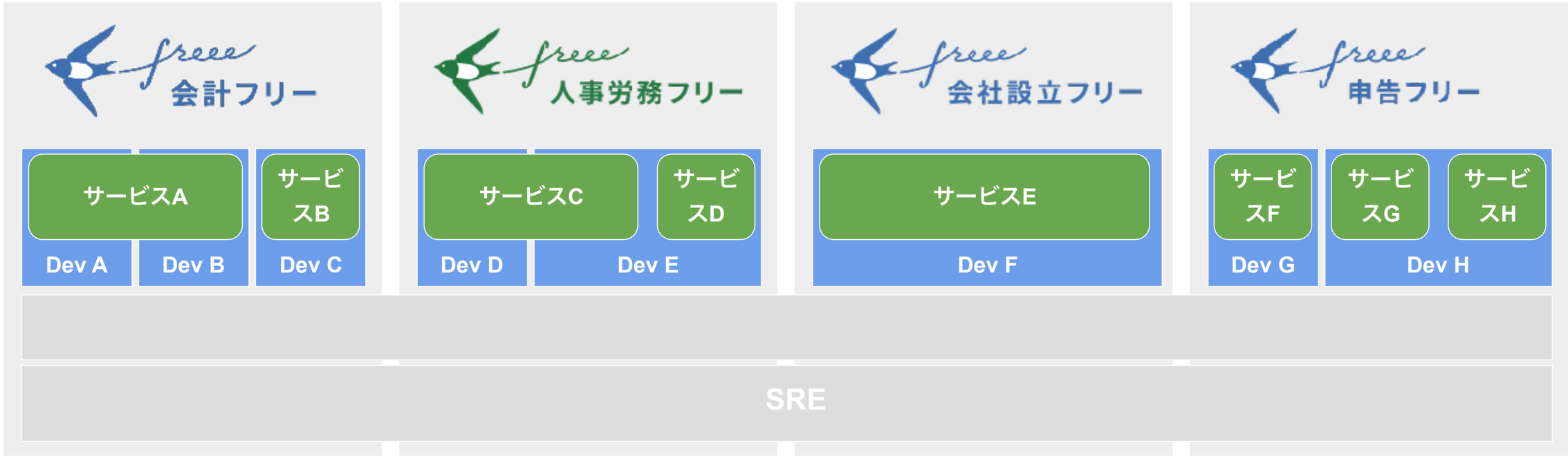
PRODUCTS

- 会計フリー
- 申告フリー
- 人事労務フリー
- マイナンバー管理フリー
- 会社設立フリー
- フリーカード
- 開業フリー

その他インターナルな
マイクロサービス多数

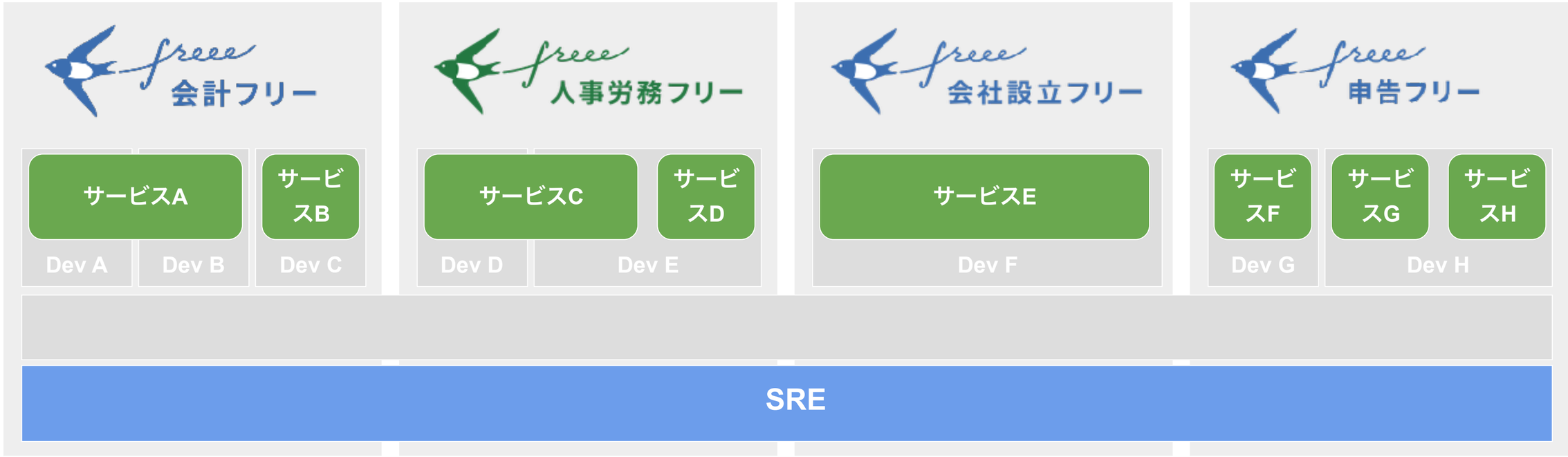
freee の 開発チーム

- 100人以上、1チーム: 2 ~ 10人程度
- 1チームで複数のサービスを兼任することが多い
- サービスの規模によっては複数のチームで開発することもある



freee の SREチーム

- 8人
- すべてのプロダクト/サービスのインフラを支える横断的なチーム
- サービスの価値をユーザーに届けるために、安定したインフラを提供し続けるのがミッション



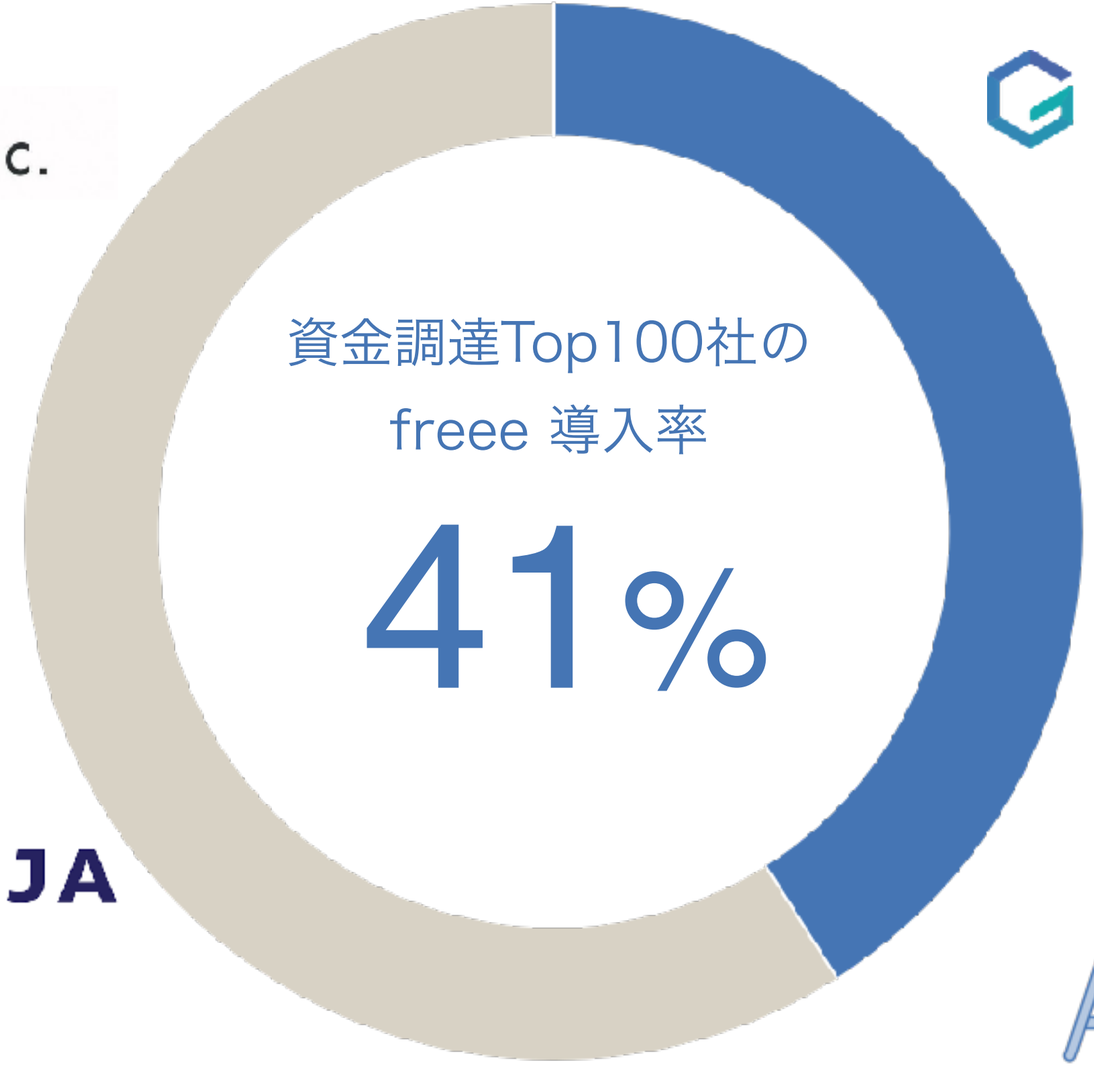
IPO準備・成長企業への導入が加速

PORT INC.

RakSul

SOLDOUT

ABEJA



GA TECHNOLOGIES

SmartNews

※ 出典：entrepedia ベンチャーリスト

※ 資金調達額TOP100社：直近1年で1億円以上の資金調達をした企業を対象に調査

受託業務に係る内部統制の保証報告書(SOC1 Type2報告書)を受領

SOC保証:
上場企業が自社の財務報告がきちんとしていることを保証するもの
freee(会計ソフト) を利用する場合、freeeも監査の対象になる
SOC1を取得していればfreeeがSOC保証が満たされていると認められる

相応のセキュリティ対策が必要

例えばDBに直接アクセスする場合
認証、認可、履歴管理が必要

EC2インスタンス等からのアクセスも
SecurityGroup などで明確に権限管
理できていることが望ましい

Section

02

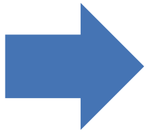
インフラリソースのコード化

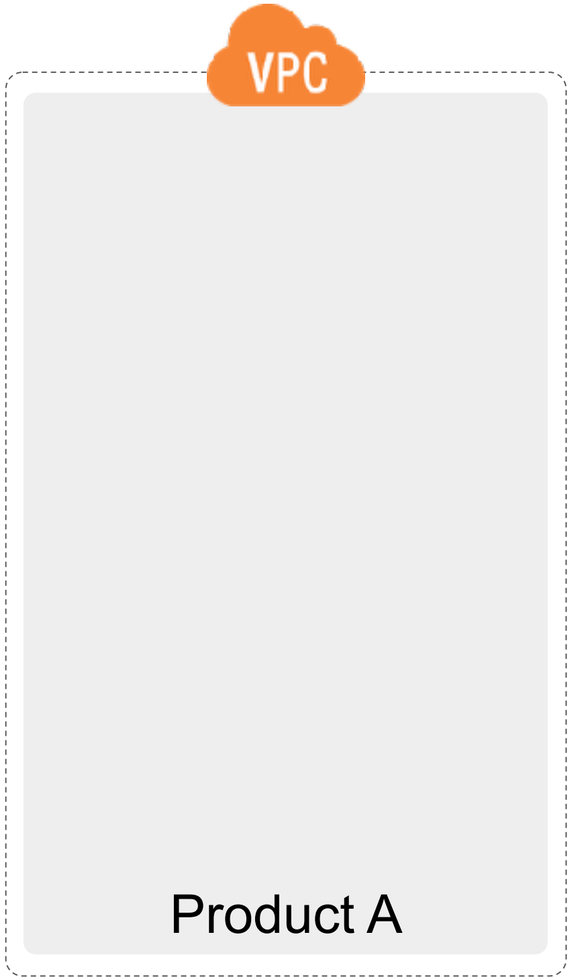
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

・ネットワーク整備



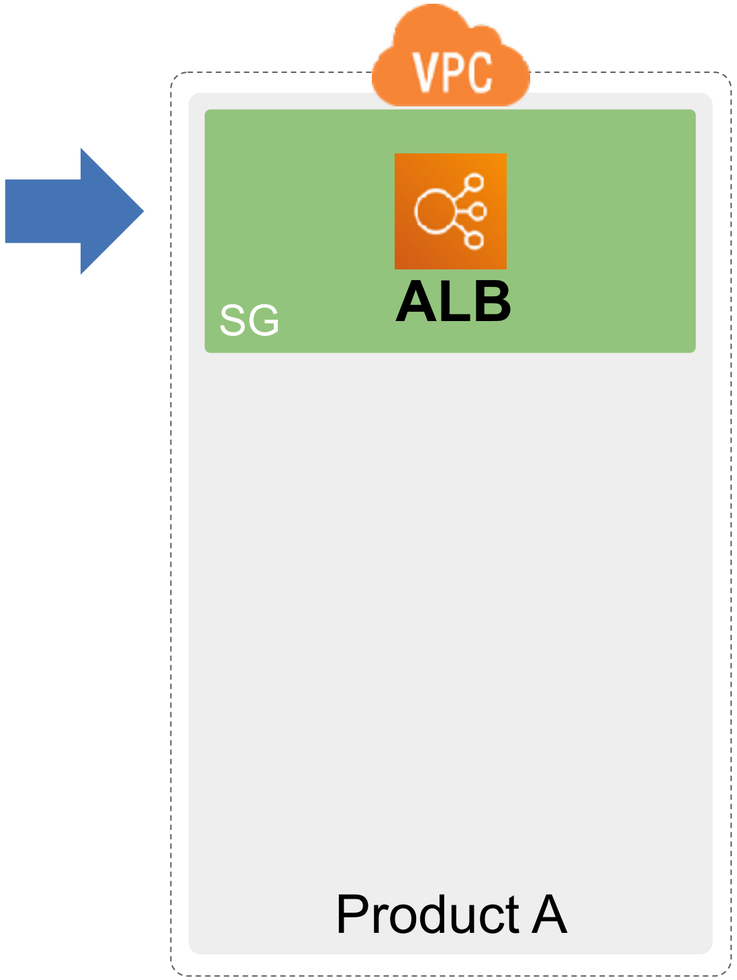


SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

・ネットワーク整備
・LB追加

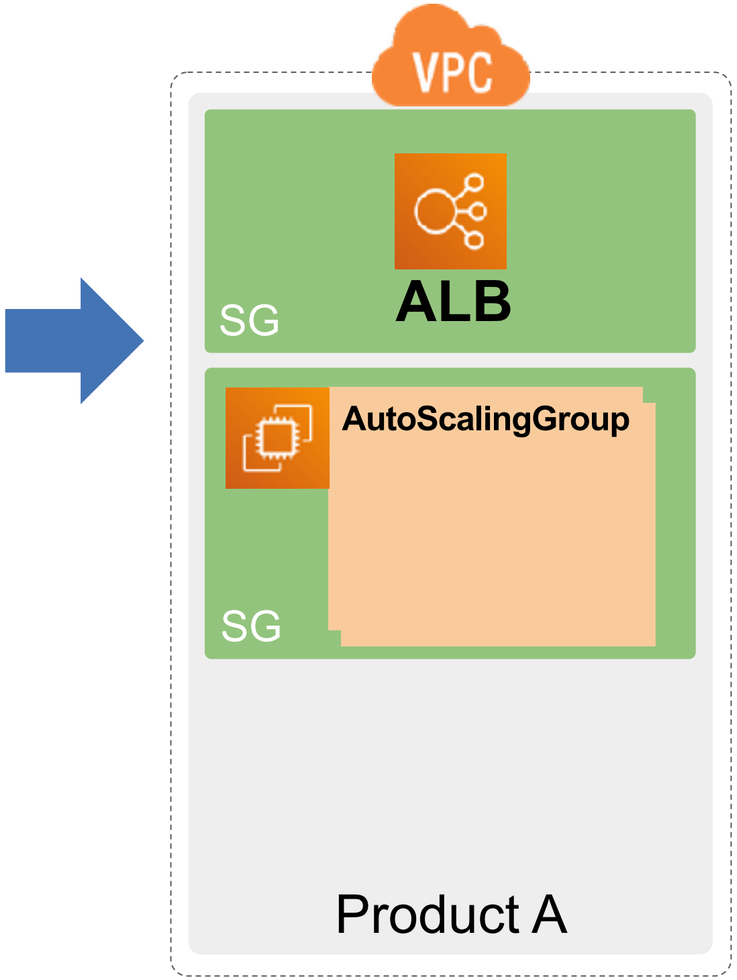


SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup 追加

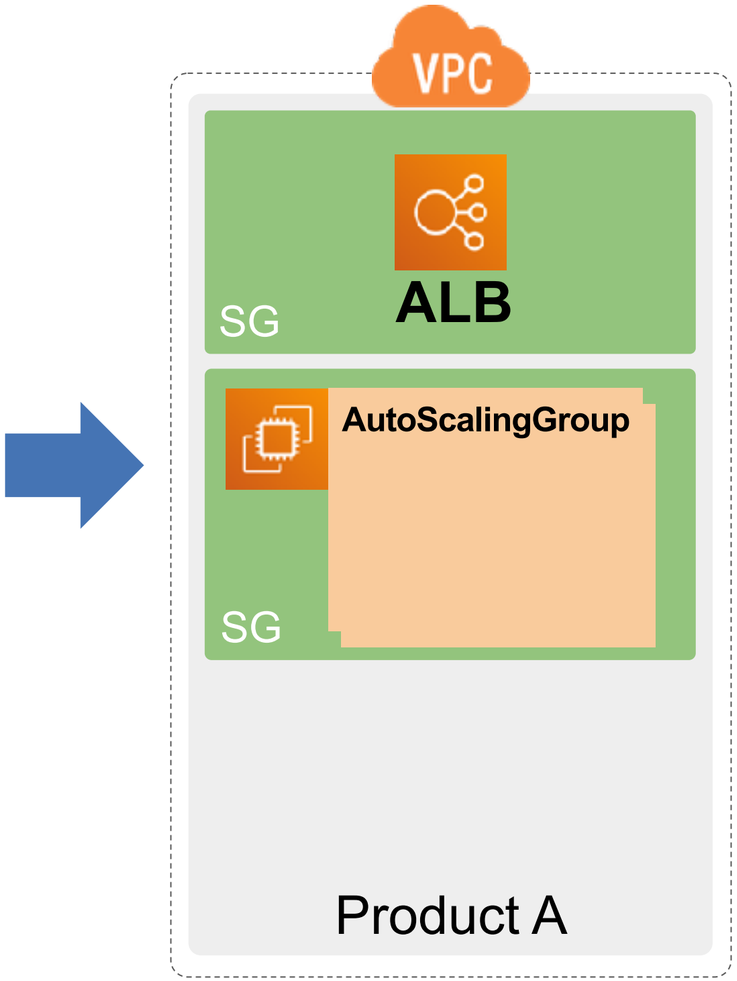


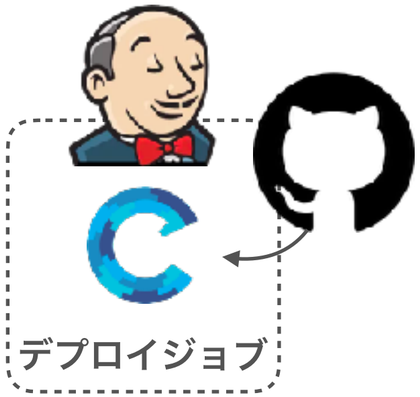
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup 追加
- デプロイ環境整備



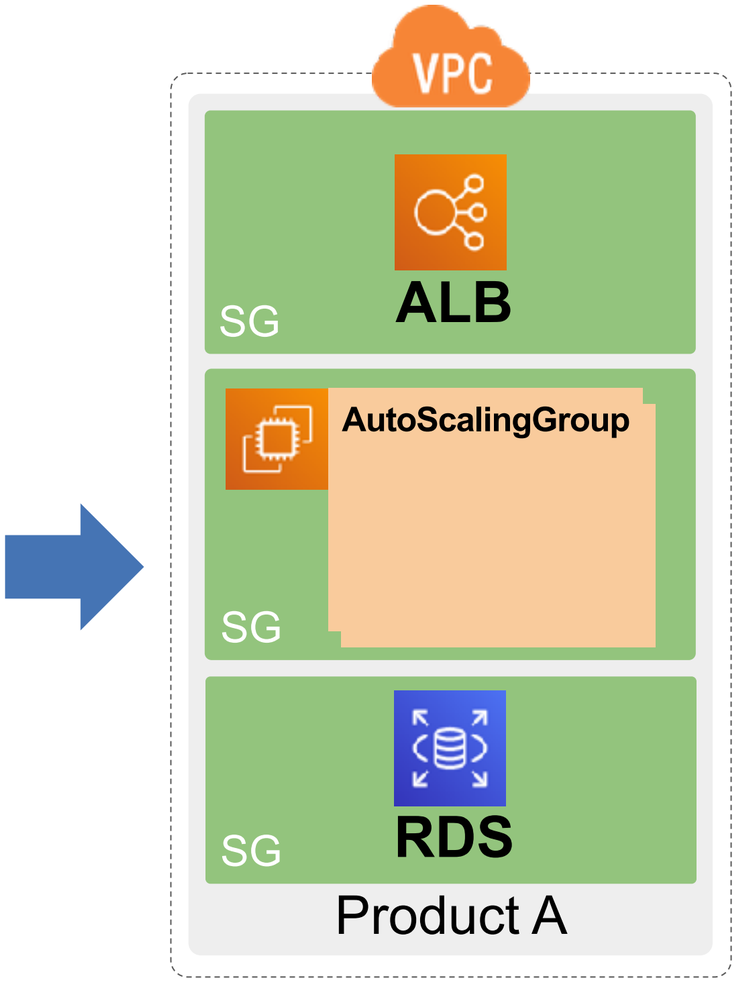


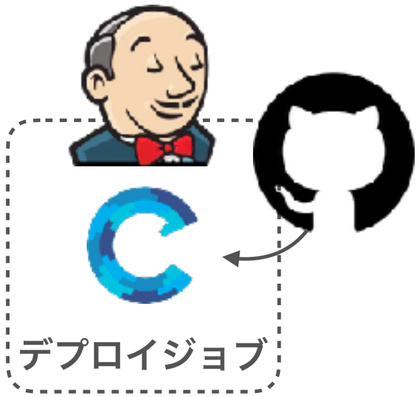
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup 追加
- デプロイ環境整備
- DB追加



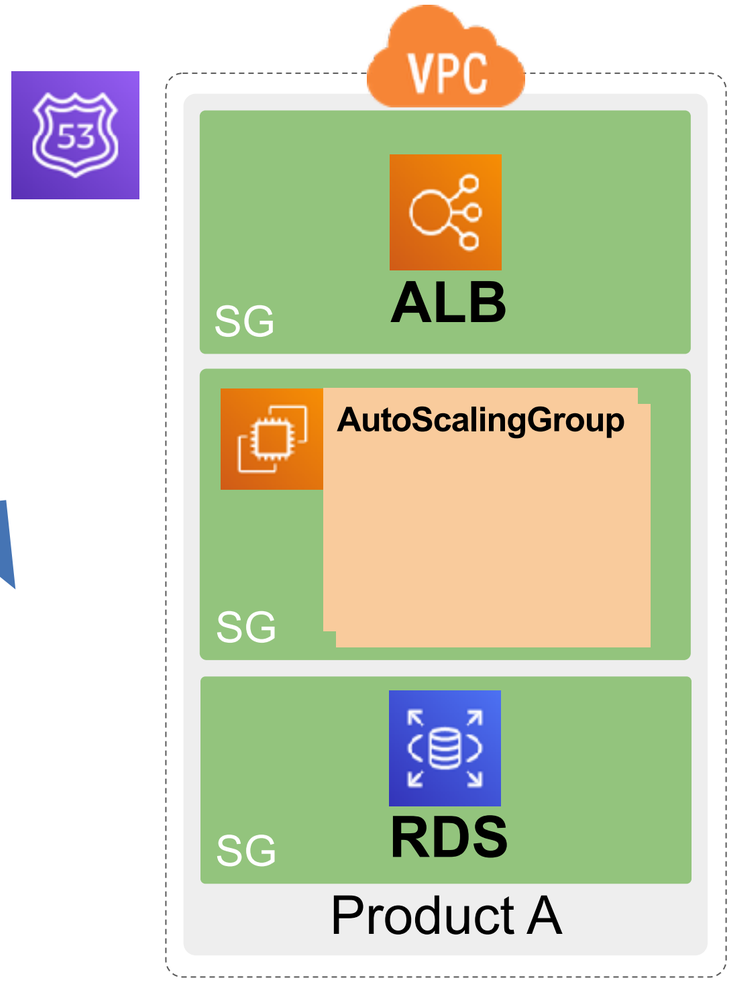


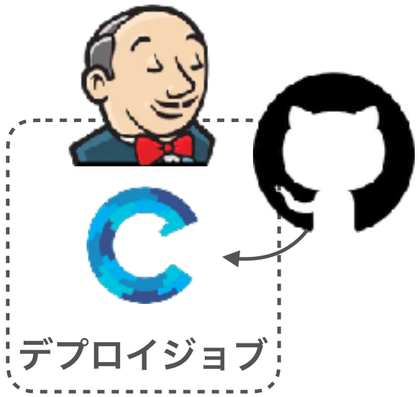
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup追加
- デプロイ環境整備
- DB追加
- Route53登録



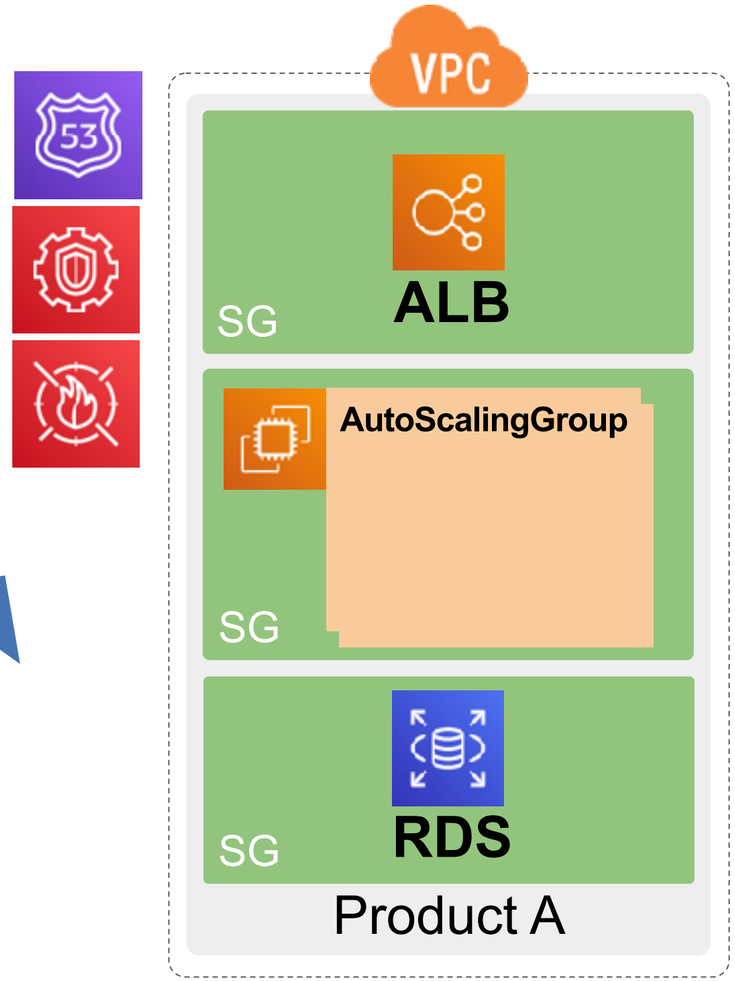


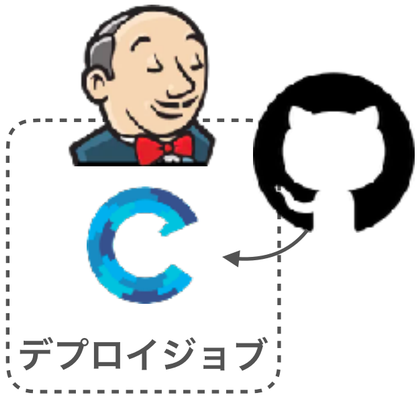
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup追加
- デプロイ環境整備
- DB追加
- Route53登録
- セキュリティ確保



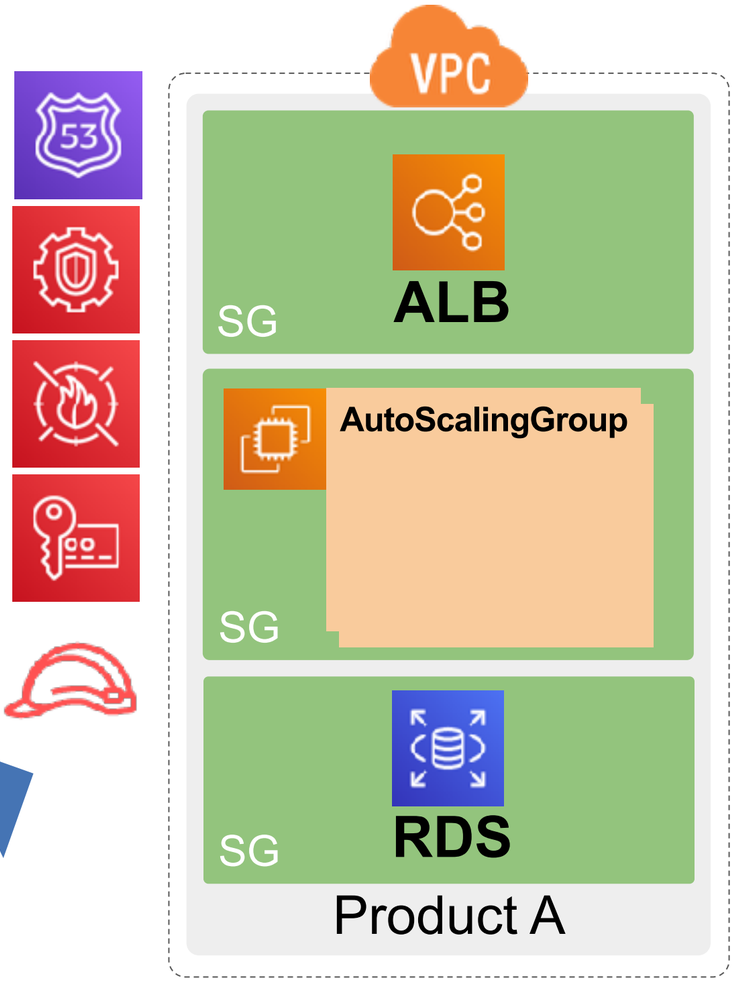


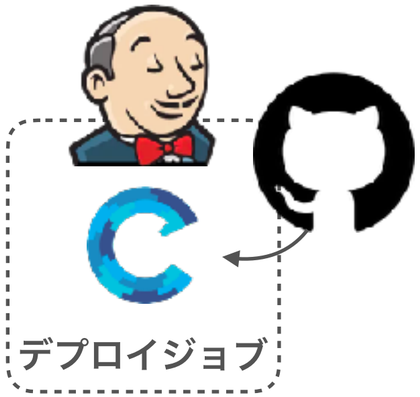
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup 追加
- デプロイ環境整備
- DB追加
- Route53登録
- セキュリティ確保
- IAMロール追加



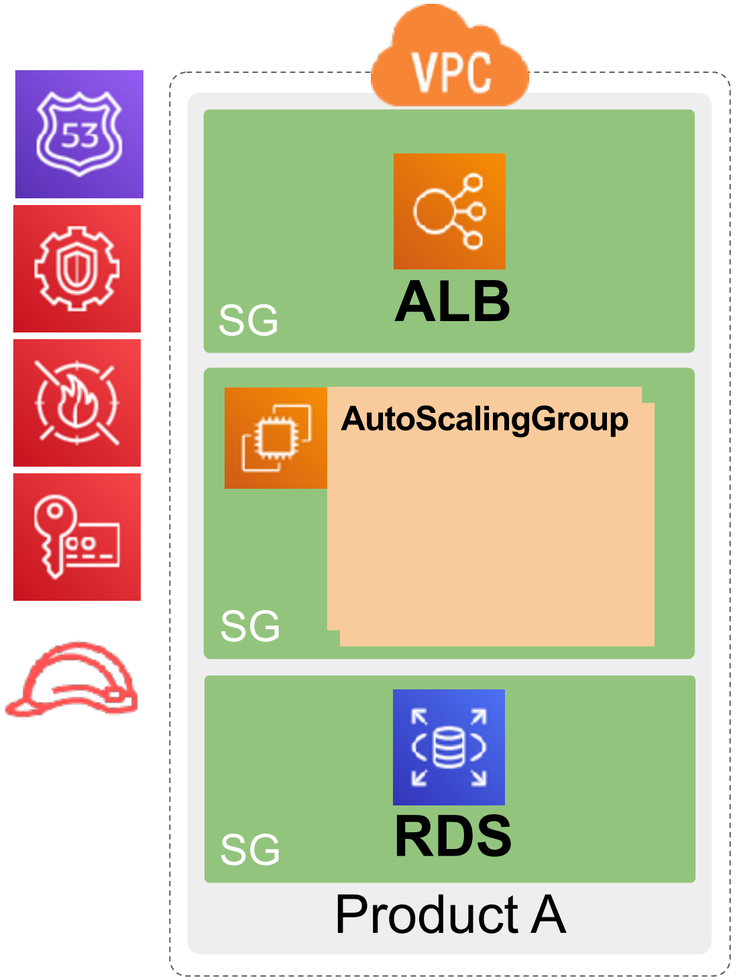


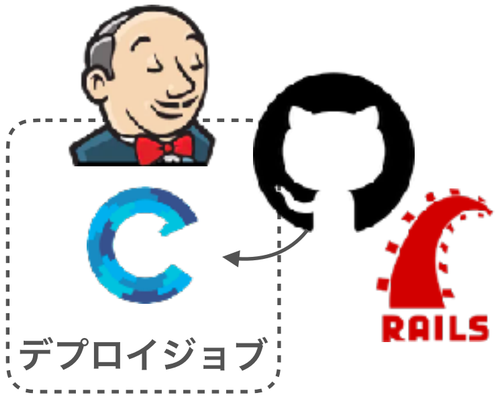
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup追加
- デプロイ環境整備
- DB追加
- Route53登録
- セキュリティ確保
- IAMロール追加





- アプリケーション開発

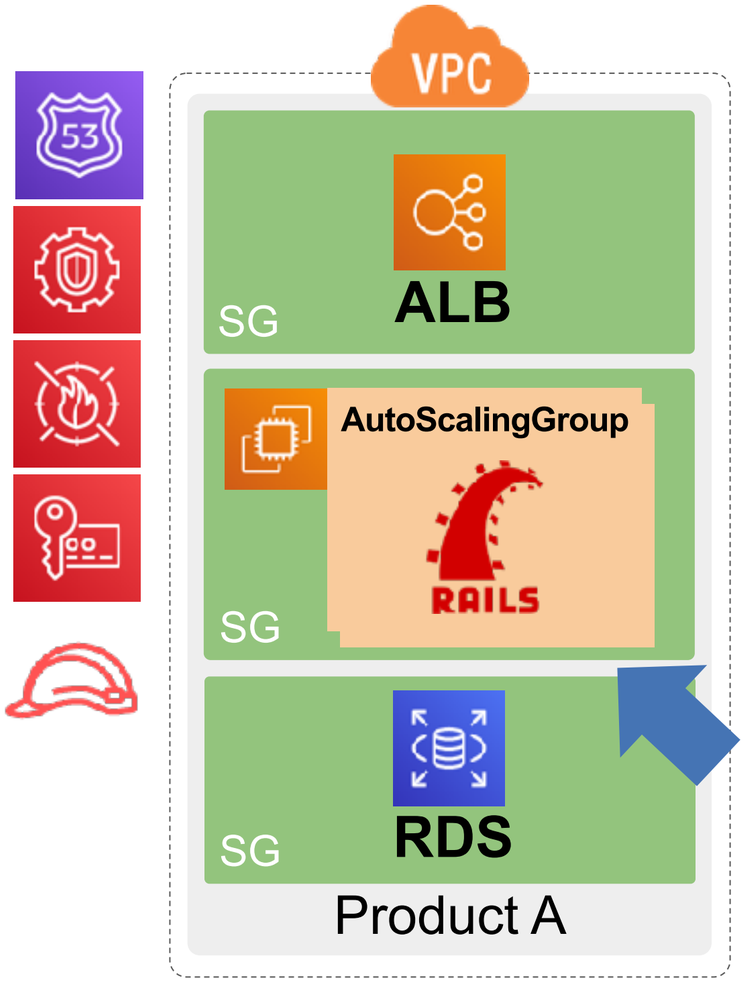
Developers

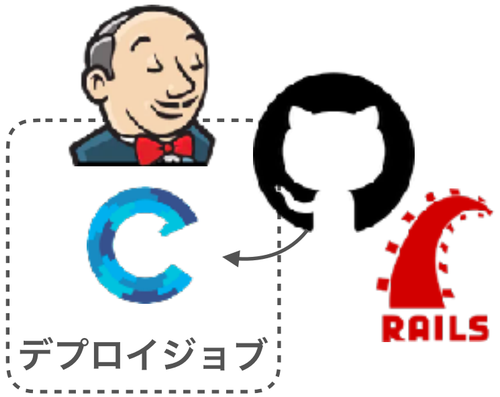
SREと開発チームの役割

とある新規プロダクトをリリースするとして

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup追加
- デプロイ環境整備
- DB追加
- Route53登録
- セキュリティ確保
- IAMロール追加





- アプリケーション開発
- アプリケーションデプロイ

Developers

プロダクト(サービス)はリリースして終わりではない

本番はここから

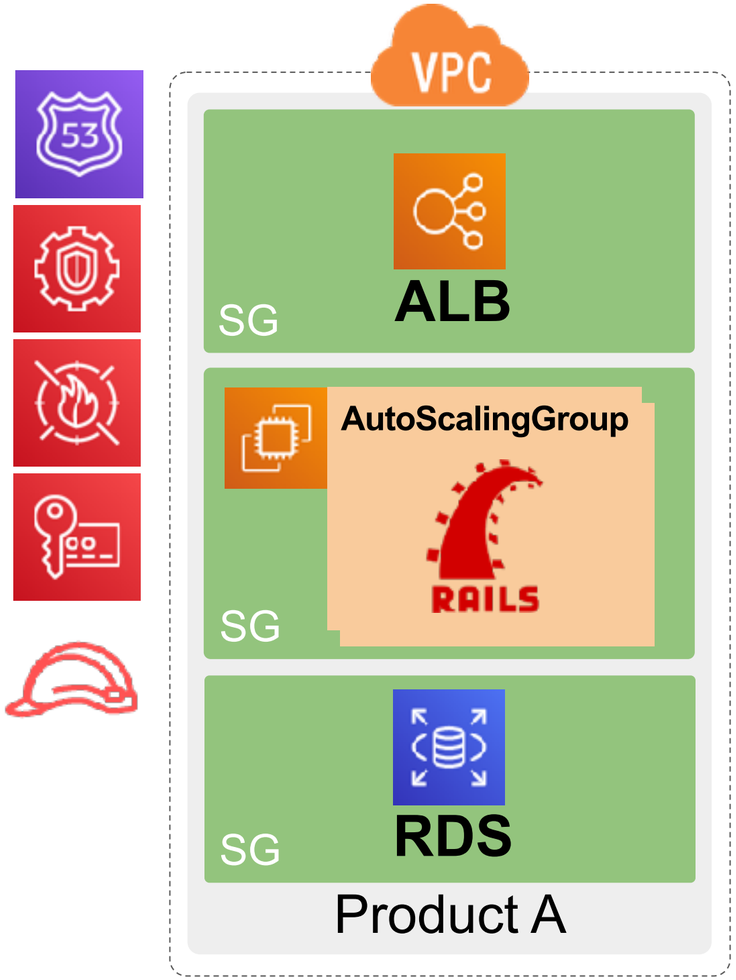
SREと開発チームの役割

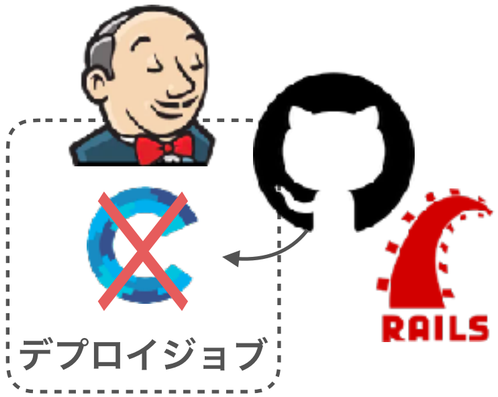
運用フェーズでは問い合わせはSREに集まりがち

SRE

デプロイ失敗しました

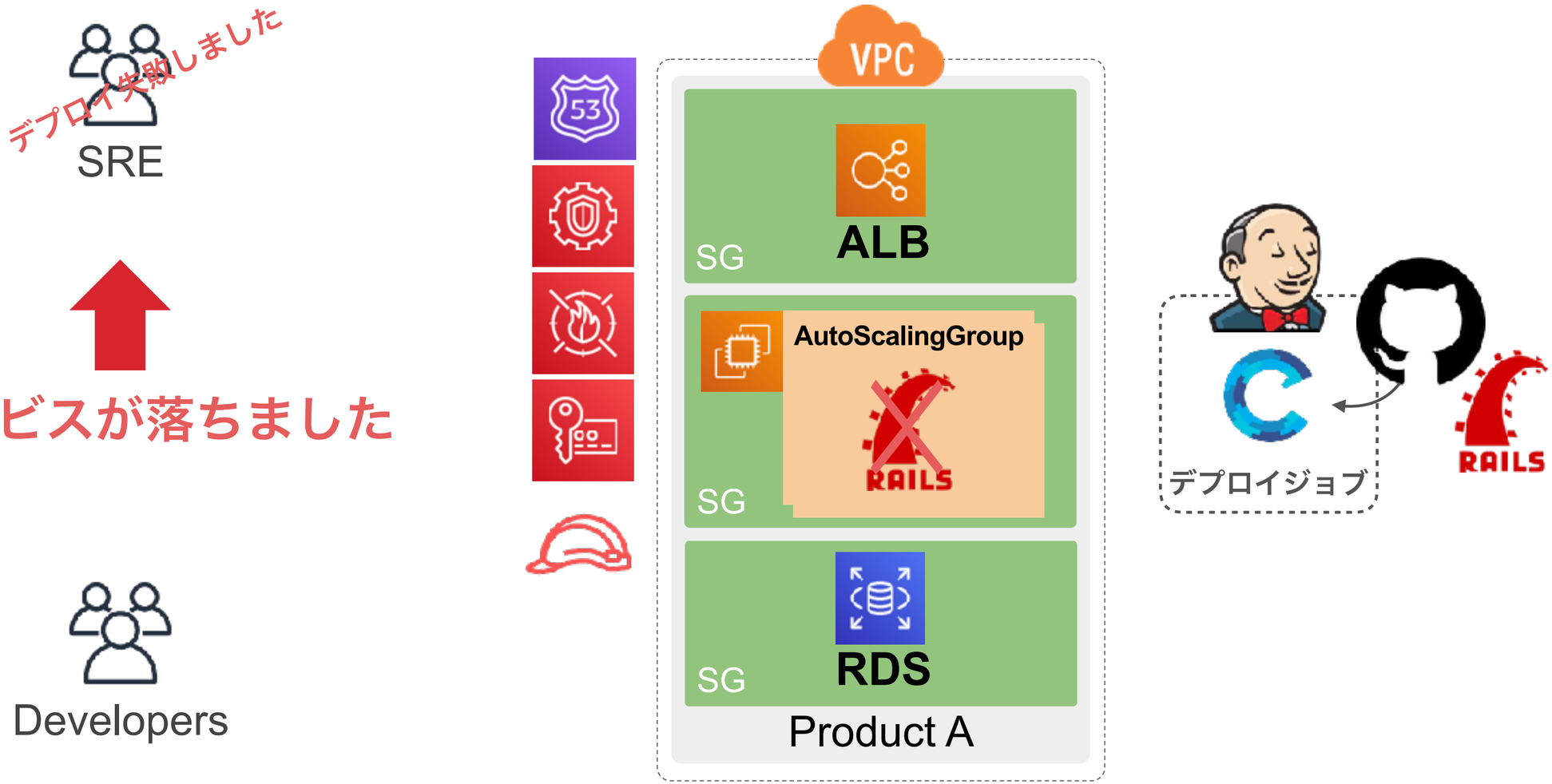
Developers





SREと開発チームの役割

運用フェーズでは問い合わせはSREに集まりがち

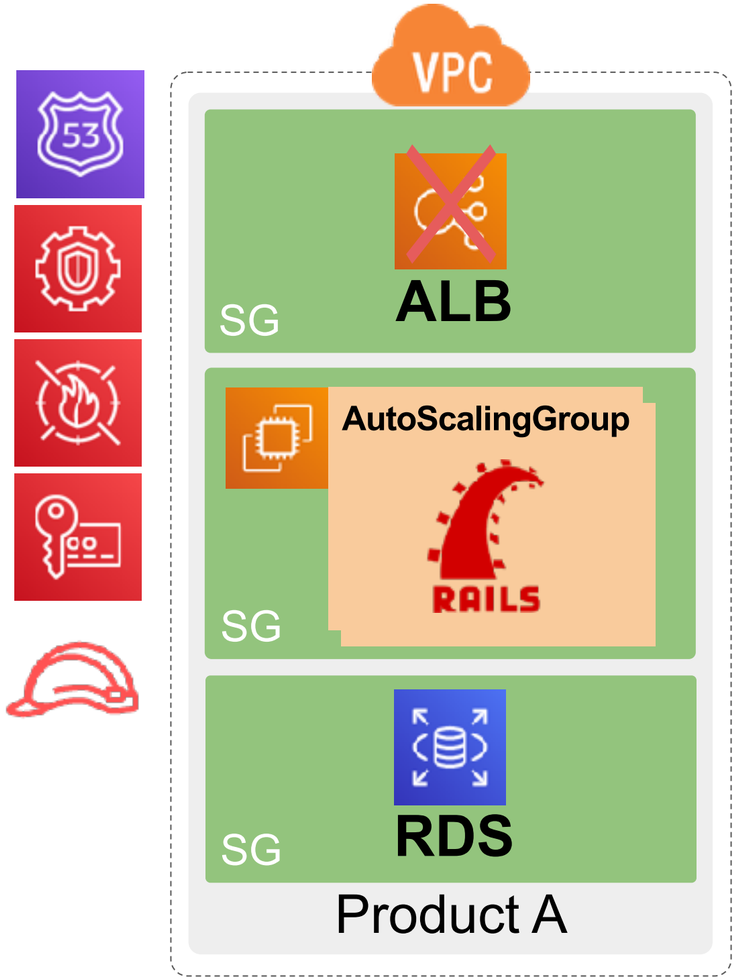


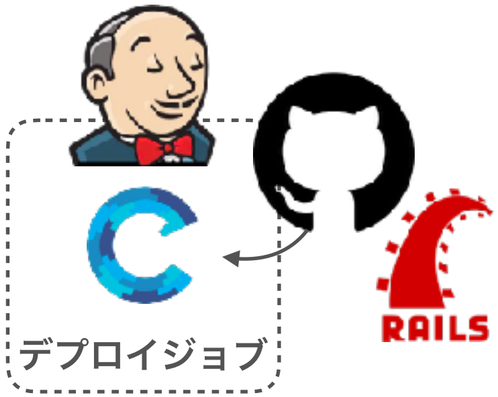
SREと開発チームの役割

運用フェーズでは問い合わせはSREに集まりがち

アクセス数増加して
さばききれません

Developers



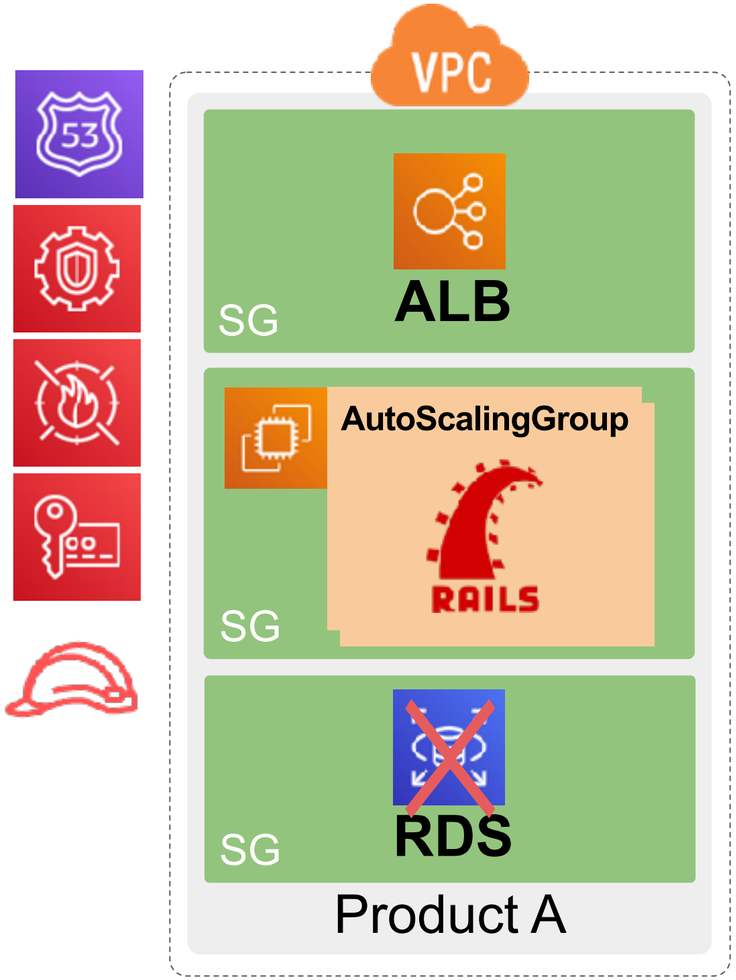


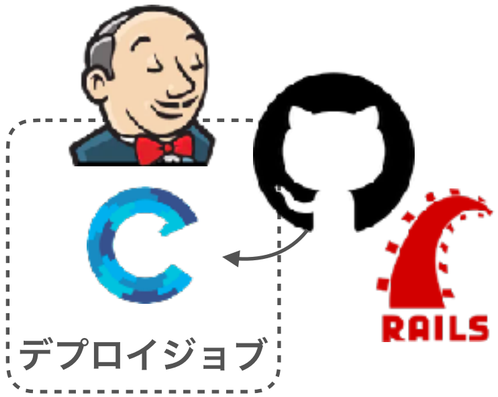
SREと開発チームの役割

運用フェーズでは問い合わせはSREに集まりがち

DBのIOPS高いです、
耐えられません

Developers



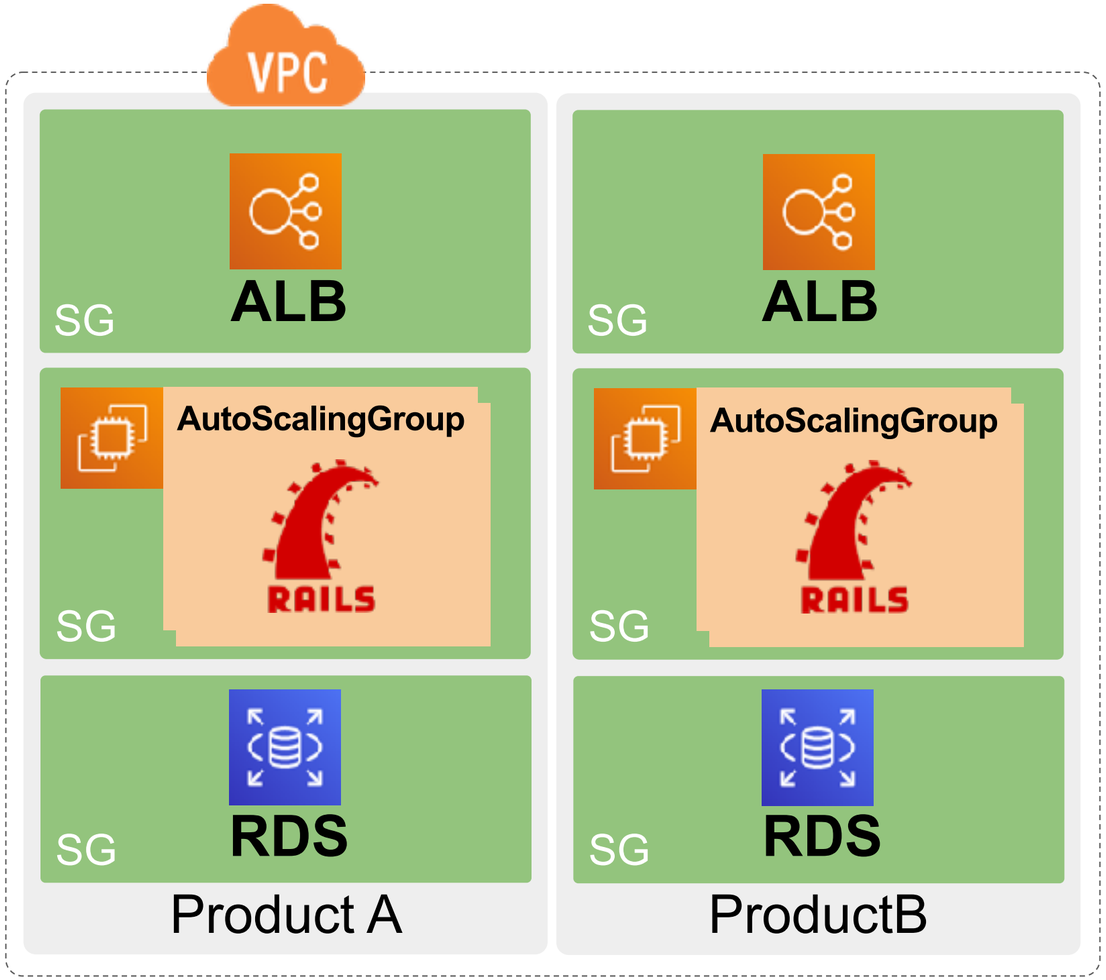


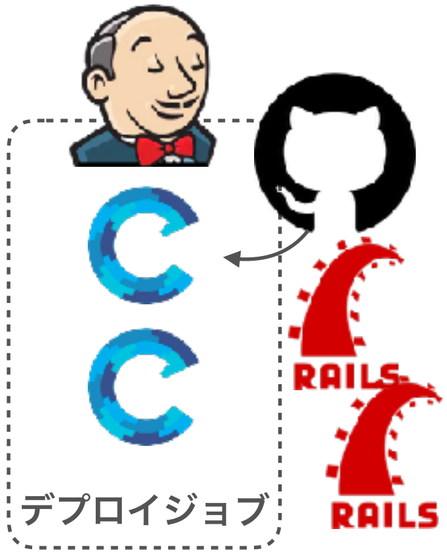
SREと開発チームの役割

運用フェーズでは問い合わせはSREに集まりがち

サービスが増えました

Developers





SREと開発チームの役割

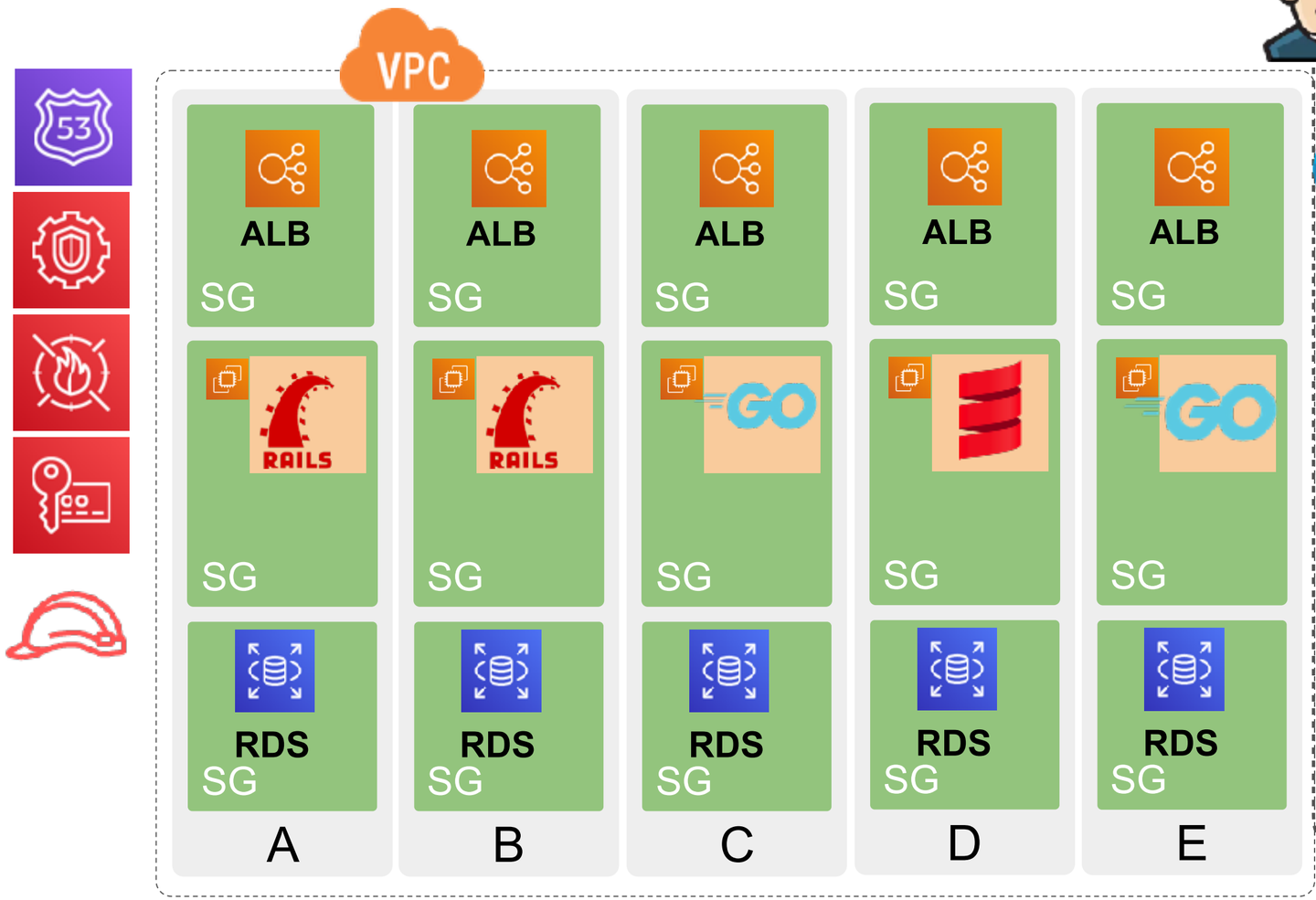
運用フェーズでは問い合わせはSREに集まりがち

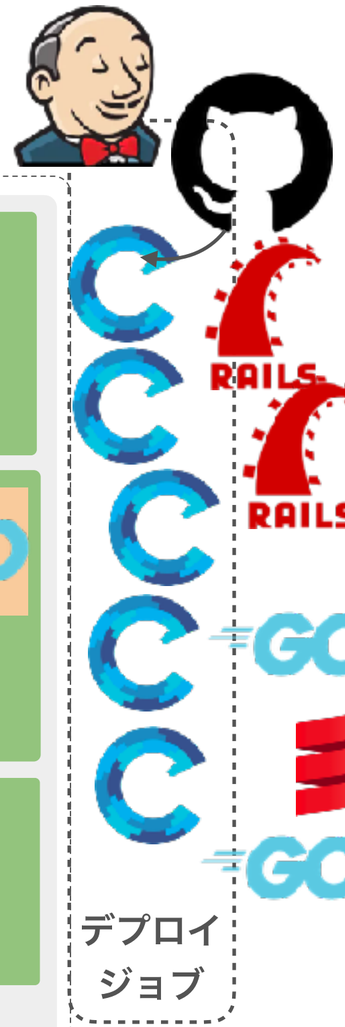
SREと開発チームの役割

運用フェーズでは問い合わせはSREに集まりがち

サービスが増えました

Developers





SREがボトルネックに

- サービスが増えるに従って SRE への依頼件数も増加
- 開発者の方が圧倒的に多いので、SRE がボトルネックに
- 何でも屋になりがちで、目先のタスクに追われる日々

マイクロサービス化の流れ

開発組織の拡大に伴い、これまでのEC2 + Auto Scaling だと辛くなってきた

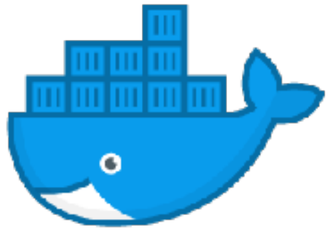
- 言語やフレームワークの多様化
- 複雑化するデプロイフロー
- 依存するサービスの増加
- SREに問い合わせがさらに集中

freeeを支えるインフラ系ツール

すべてのアプリケーションをコンテナ化

本番環境のコンテナのランタイムとして採用

インフラリソースのコード化

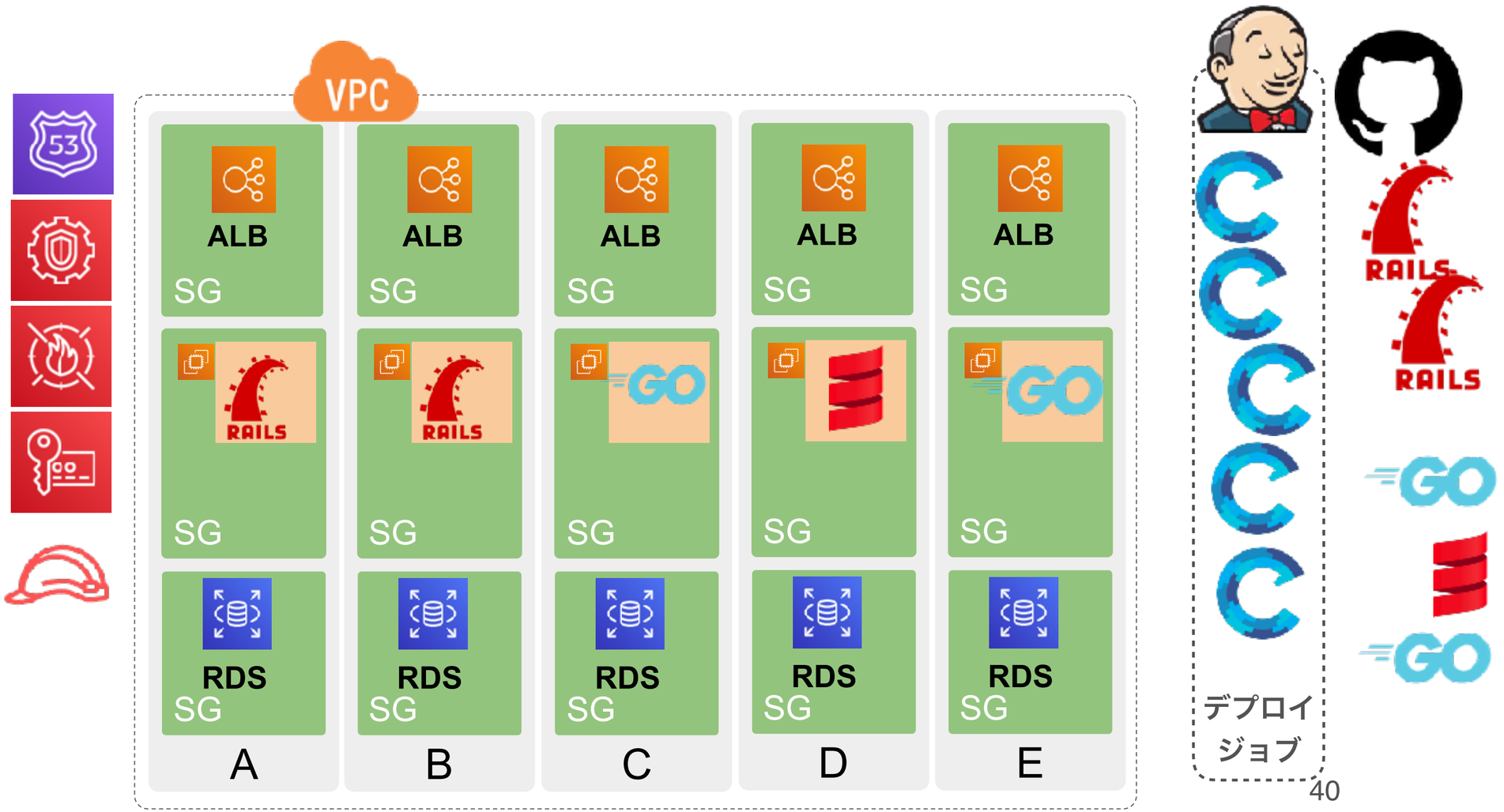


docker

すべてのアプリケーションをコンテナ化

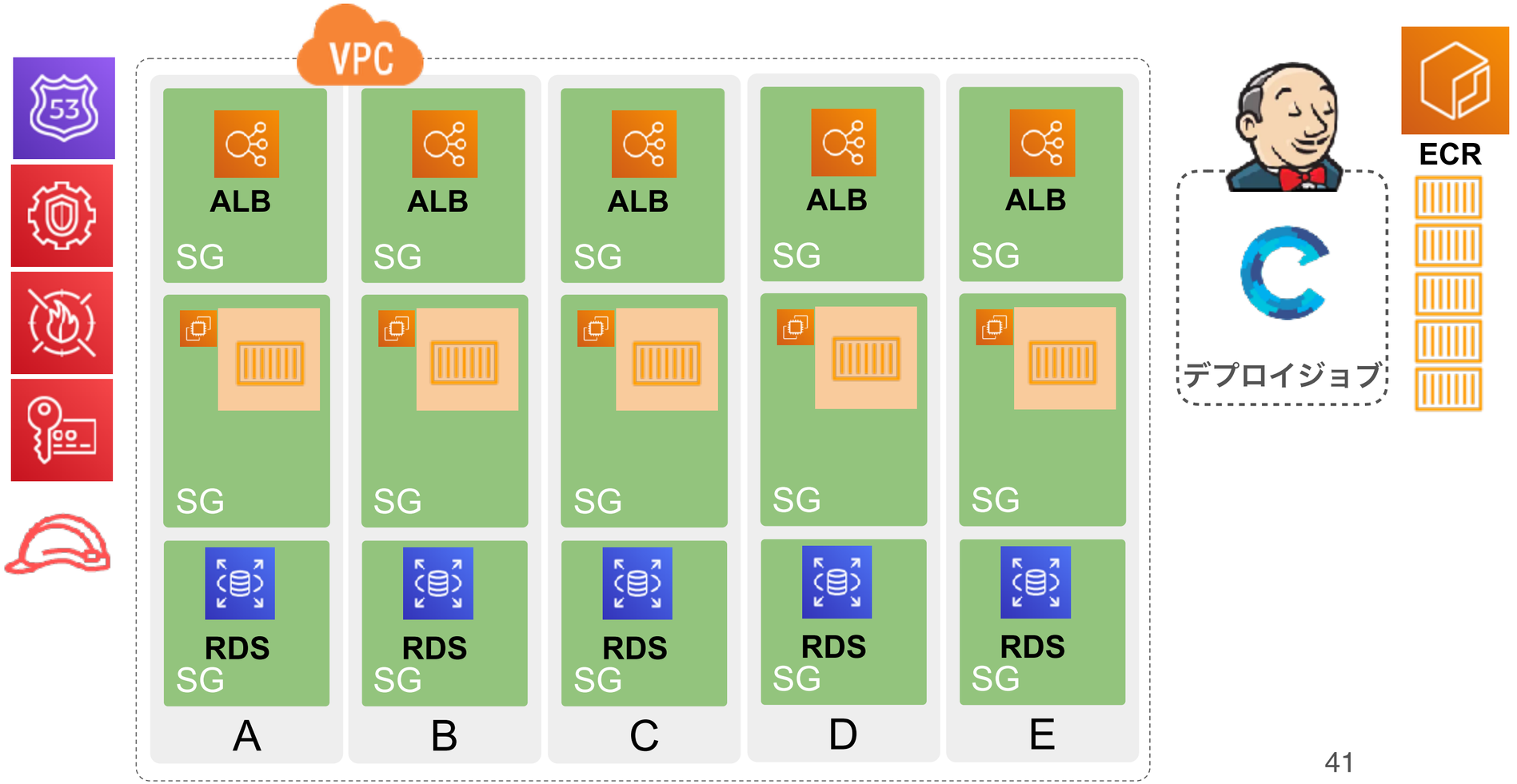
多様化する言語やフレームワークを吸収

すべてのアプリケーションをコンテナ化



すべてのアプリケーションをコンテナ化

コンテナに吸収されて考え方がシンプルに



kubernetes

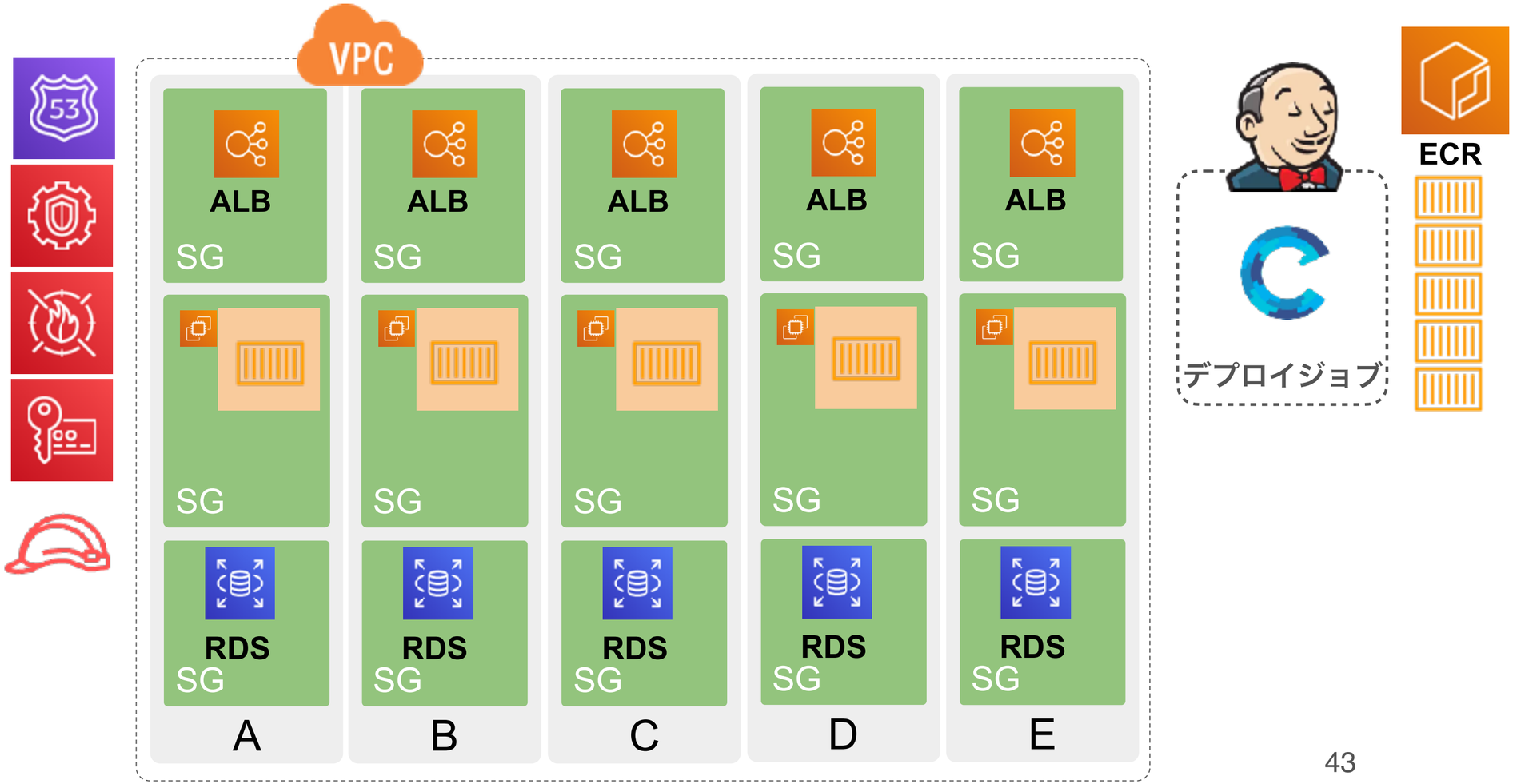
本番環境のコンテナのランタイムとして採用

アプリケーションの動作環境をマニフェストとしてコード化

宣言的にデプロイ、オートスケーリング、セルフヒーリングを実現

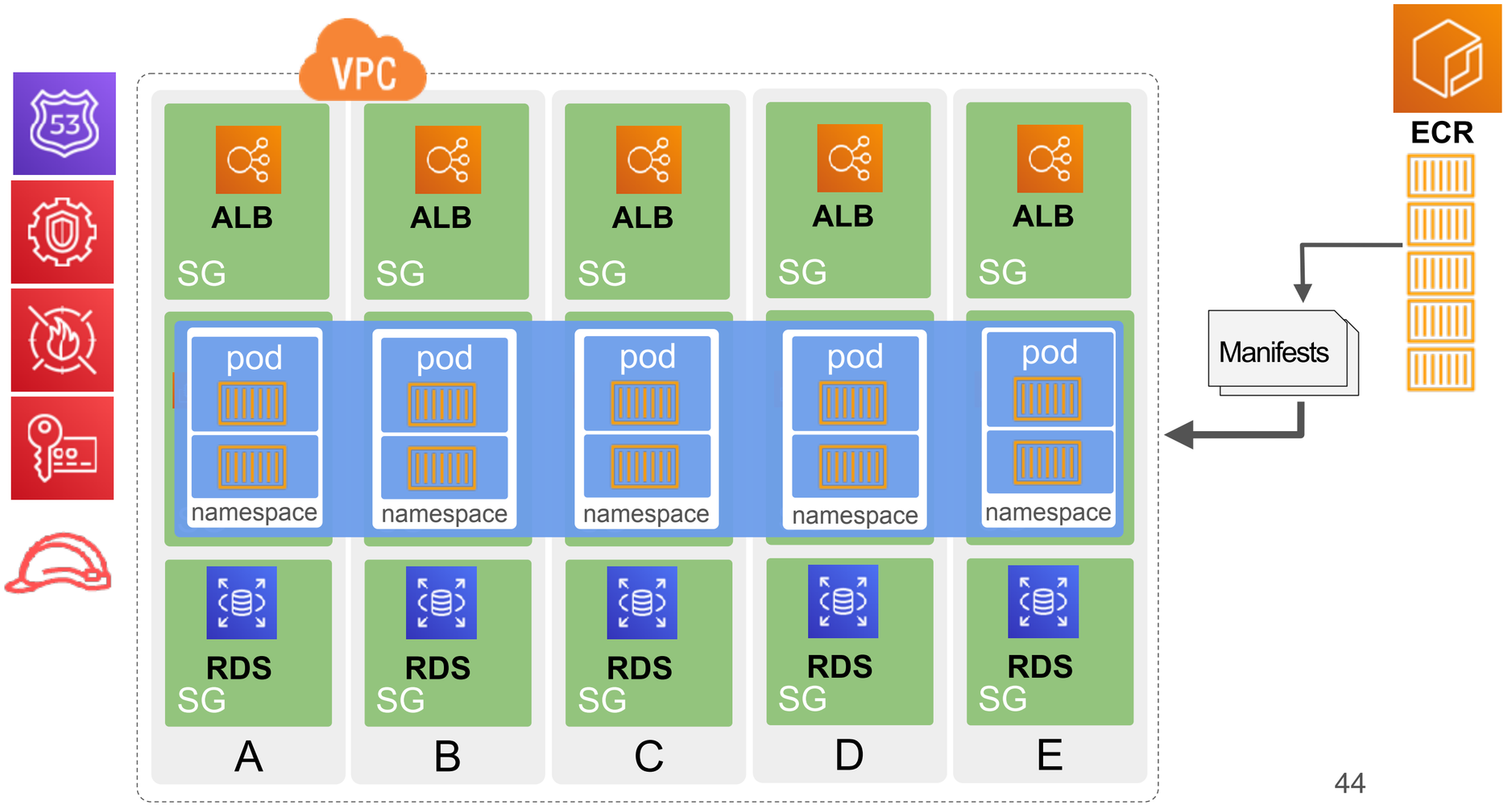
コンテナをKubernetesで動かす

アプリケーションの構成がコード化される



コンテナをKubernetesで動かす

アプリケーションの構成がコード化される



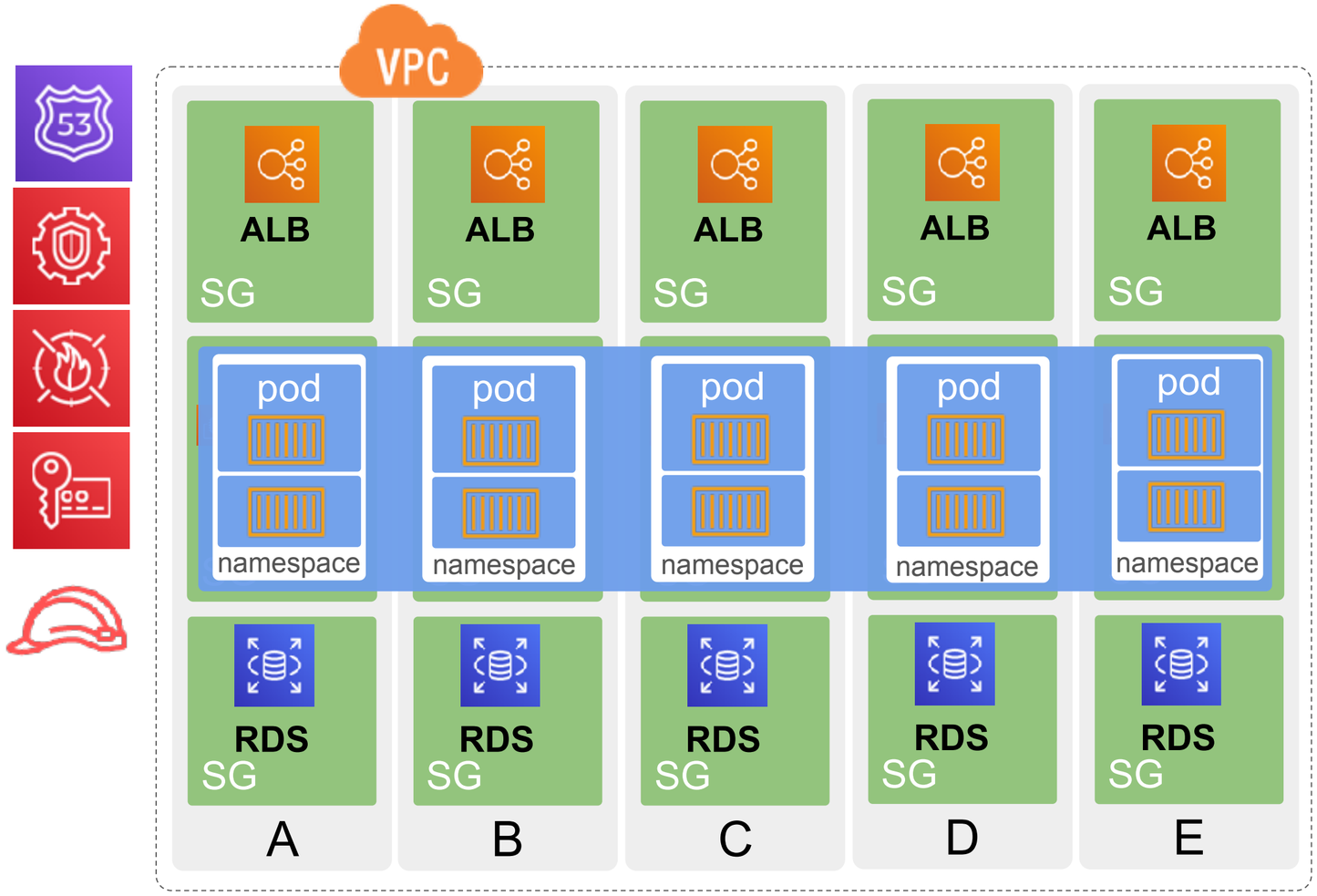
AWSリソースのコード化

宣言的にAWSのリソースを確保

宣言的にAWSリソースを確保

SRE

- ネットワーク整備
- LB追加
- AutoScalingGroup 追加
- デプロイ環境整備
- DB追加
- Route53登録
- セキュリティ確保
- IAMロール追加

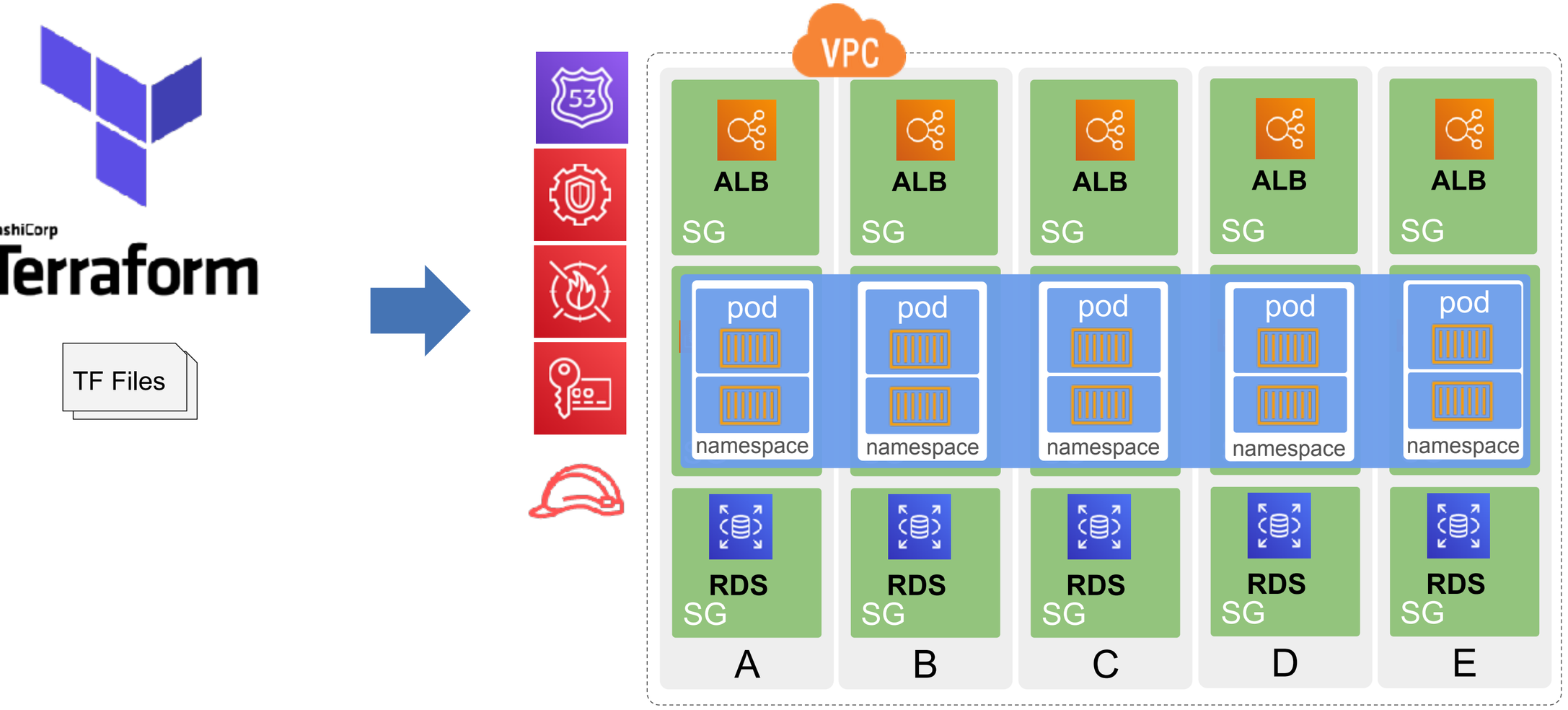


宣言的にAWSリソースを確保

AWSリソースがコード化される

HashiCorp Terraform

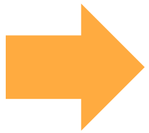
TF Files



インフラリソースがコード化されると開発チームとSREとのコミュニケーション方法が変わる

インフラリソースがコード化されていない世界

Developers



サービスが増えました

SRE

この言葉には以下の内容が含まれているのではないか

インフラの構築には強い権限が必要

現状のインフラ構成を理解していない

インフラを触るのはなんとなく怖い

SREにお願いするしかない

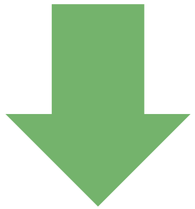
インフラリソースがコード化された世界

コード書きました！
レビューお願いします！

開発チームがインフラのコードを書いてSREがレビューする

インフラ構築の権限が与えられている
現状のインフラ構成は既存のコードから読み取れる
インフラを触るのはまだ怖いけどSREにレビューしてもらえる

インフラリソースがコード化されると開発チームとSREとのコミュニケーション方法が変わる



開発者チームにサービスの運用をおまかせできるかもしれない

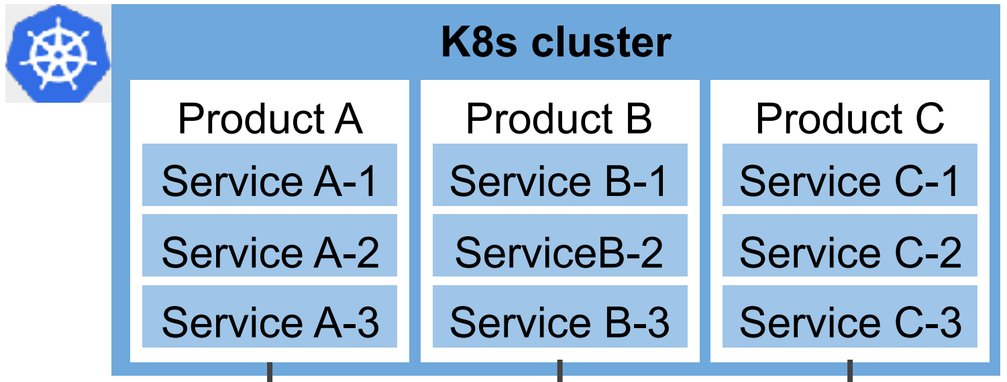
Section

03

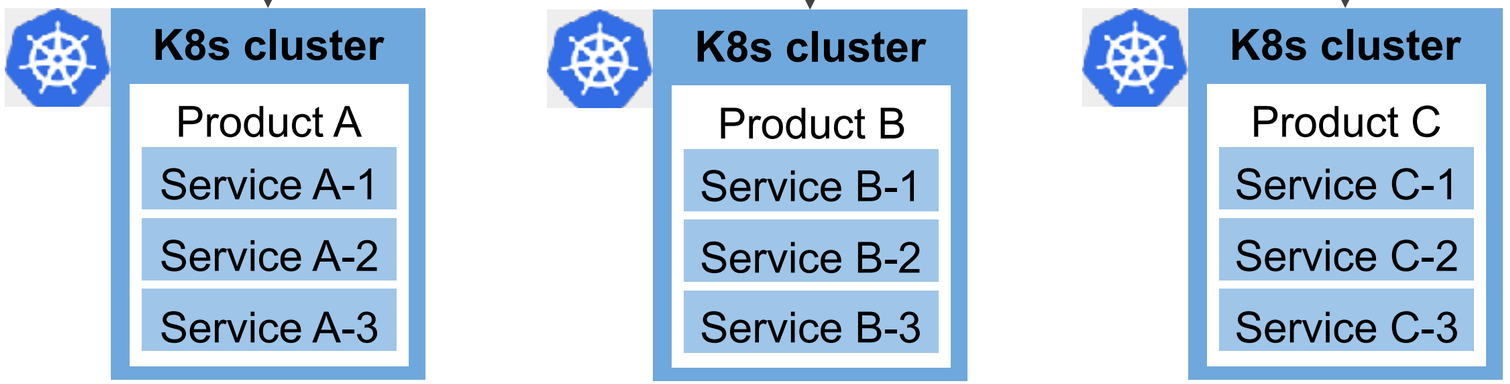
シングルテナントで権限を分離してクラスタの運用をおまかせする

マルチテナント か シングルテナント か

すべてのプロダクトが動いている
マルチテナントクラスタ



プロダクト(分離したい権限)単位で分割したシングルテナントクラスタ



マルチテナント か シングルテナント か

シングルテナントのメリット

- Blast radius(障害の影響範囲)が小さい
- セキュリティの境界線の明確化
- クラスタ全体に関係するアップデート作業がしやすい

シングルテナントのデメリット

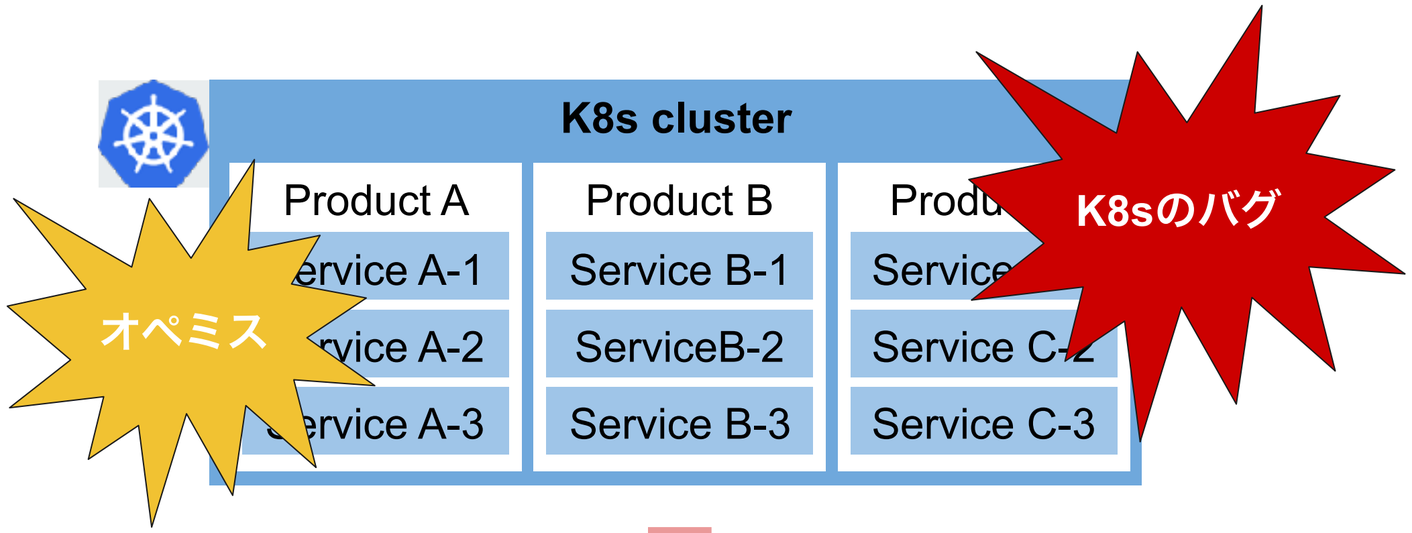
- 利用料金が増える
- 運用コストが増える

権限移譲により運用コストの分散は可能

Blast radius(障害の影響範囲)が小さい

リスクを分散し、心理的安全性を高める

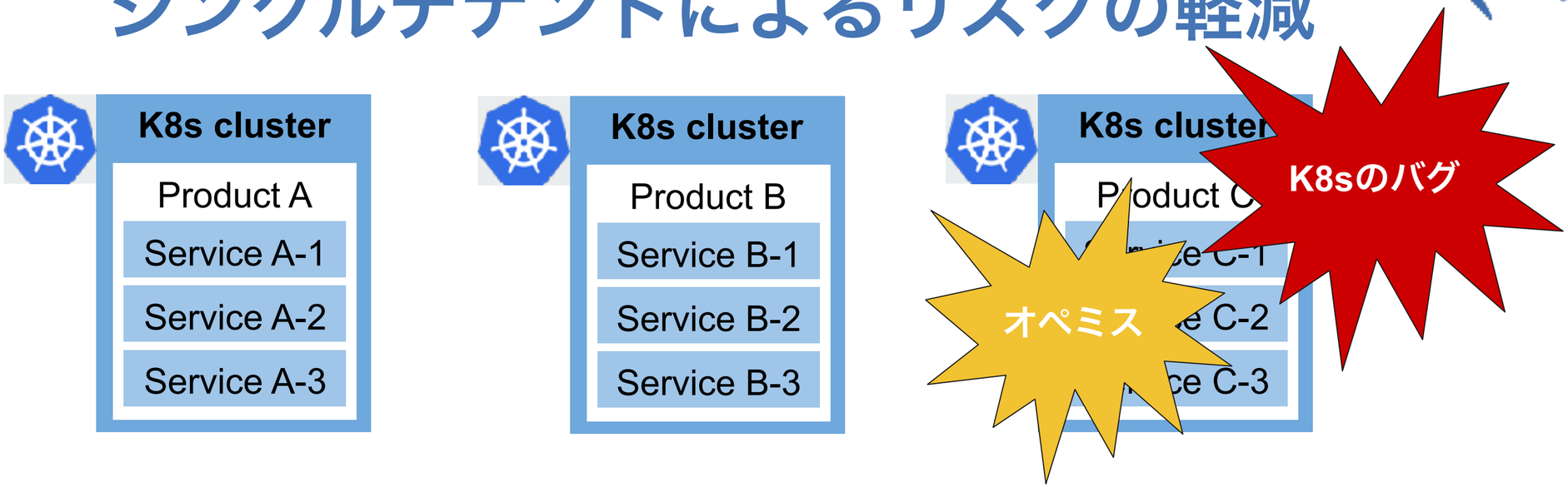
マルチテナントのリスク



全サービスダウンの危険

- Blast radius(障害の影響範囲)が大きい
- 運用の難易度が高い
- チャレンジしづらい空気

シングルテナントによるリスクの軽減



一部のみサービスダウン

- Blast radius(障害の影響範囲)が小さい
- 運用の難易度は下がる
- チャレンジしやすい空気
- 心理的安全性が高い

セキュリティの境界線の明確化

プロダクト間の不正なアクセスをどう防ぐか

受託業務に係る内部統制の保証報告書(SOC1 Type2報告書)を受領

SOC保証:
上場企業が自社の財務報告がきちんとしていることを保証するもの
freee(会計ソフト) を利用する場合、freeeも監査の対象になる
SOC1を取得していればfreeeがSOC保証が満たされていると認められる

相応のセキュリティ対策が必要

例えばDBに直接アクセスする場合
認証、認可、履歴管理が必要

EC2インスタンス等からのアクセスも
SecurityGroup などで明確に権限管理できていることが望ましい

マルチテナントで境界線の明確化は難しい

Security Group による分割は不可

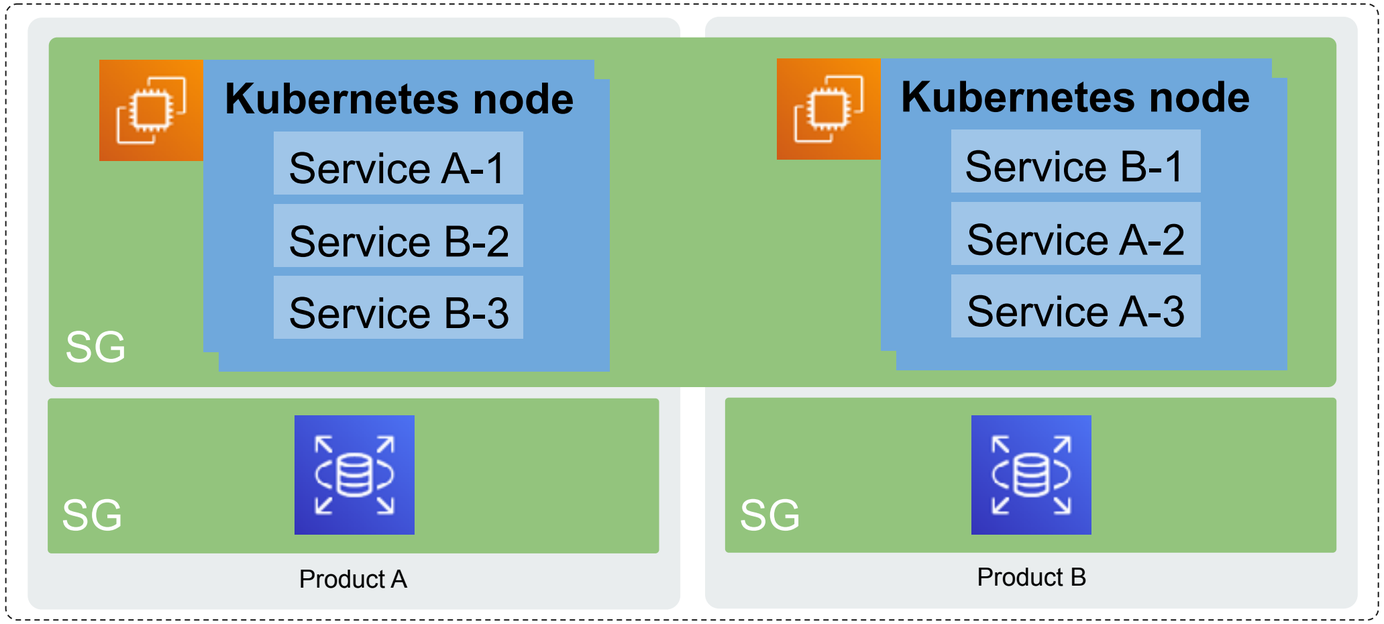
IAMとKiamでAWSリソースへの制御は可能

RBACでNamespace間のアクセス制御は可能

ただしプロダクト間でVMは共通

↓

プロダクト単位でNodeGroupを分割すれば対応可能だが、そのための仕組みづくりが必要



シングルテナントなら境界の明確化は容易

テナントは分離したい権限単位になっている

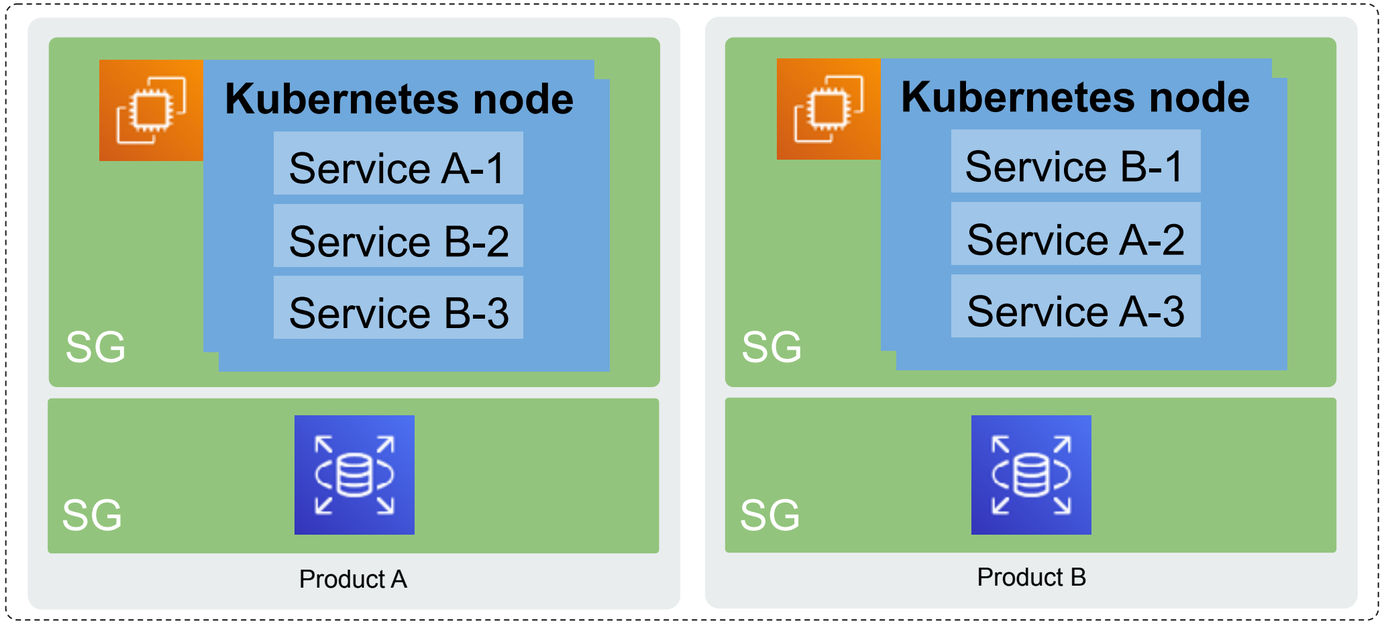
Security Group が利用可能

RBACを併用

VMレベルで分割されている

↓

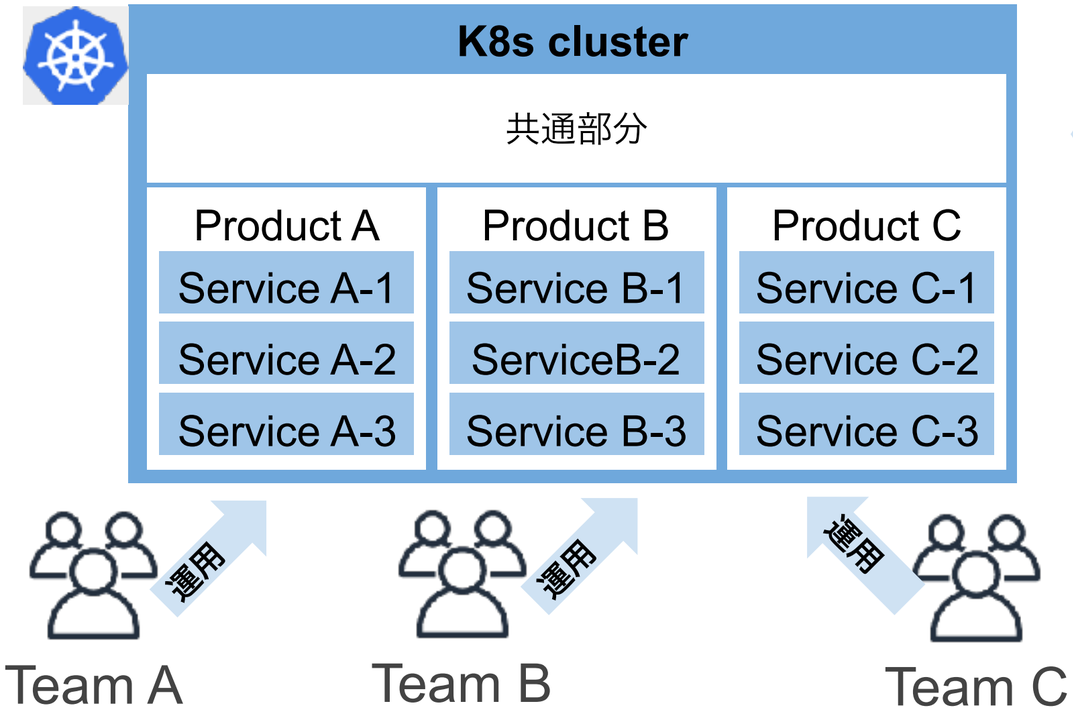
今まで運用してきた(枯れた)構成と
一緒なので扱いが簡単



クラスタ全体に関係するアップデート作業がしやすい

- Kubernetesクラスタは頻繁にアップグレードが必要
- 共通部分で利用しているツールのアップデートも必要

マルチクラスタはクラスタ全体に関係する
アップデート作業がしづらい

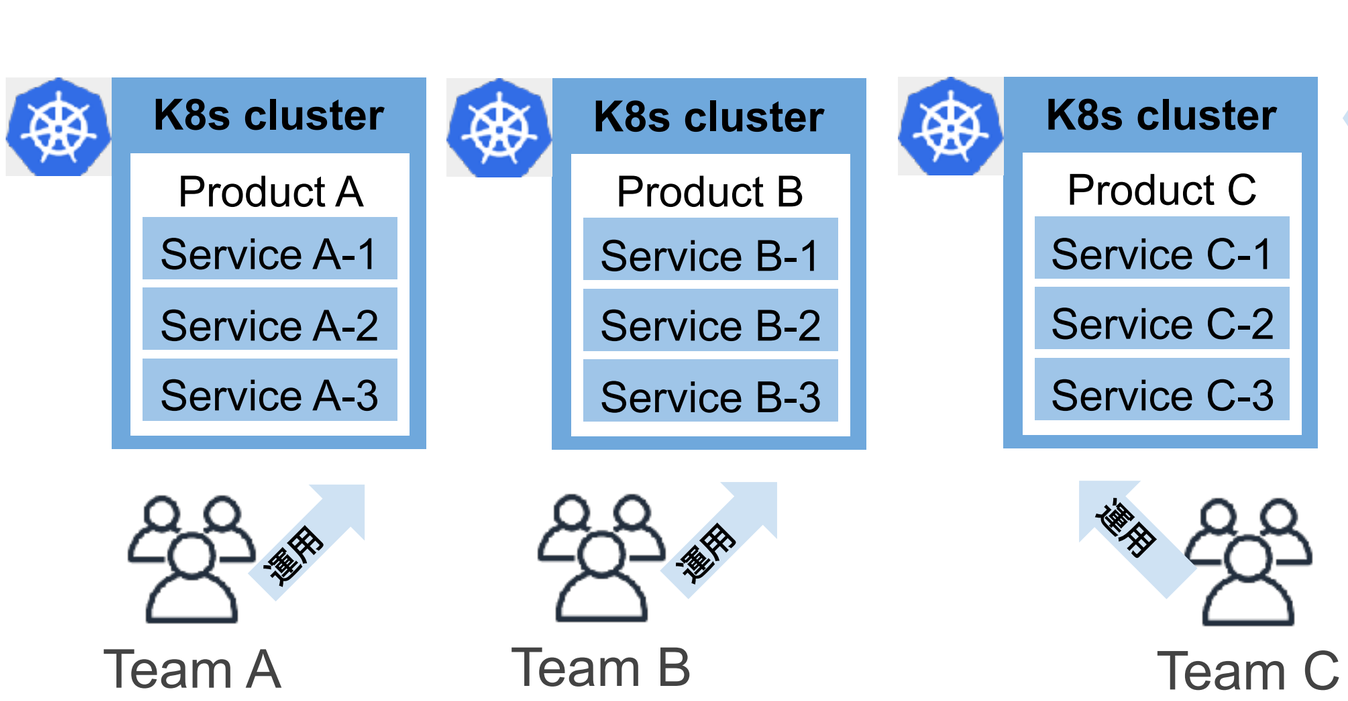


クラスタのアップグレードなど

SRE

- サービスをすべて停止させる必要がある
- アップグレードに失敗する可能性がある
- 失敗したときのロールバックのコストが高い

シングルクラスタはクラスタ全体に関係する
アップデート作業がしやすい



- サービスの停止は最小限
- アップグレードに失敗しても最小限
- 失敗したときのロールバックのコストも最小限

クラスタの運用をおまかせするならシングルテナントがマッチする

シングルテナントのメリット

- Blast radius(障害の影響範囲)が小さい
- セキュリティの境界線の明確化
- クラスタ全体に関係するアップデート作業がしやすい

開発チームがクラスタを
運用するのは簡単ではない

各クラスタ/サービスを横断的に
面倒を見るチームを設置

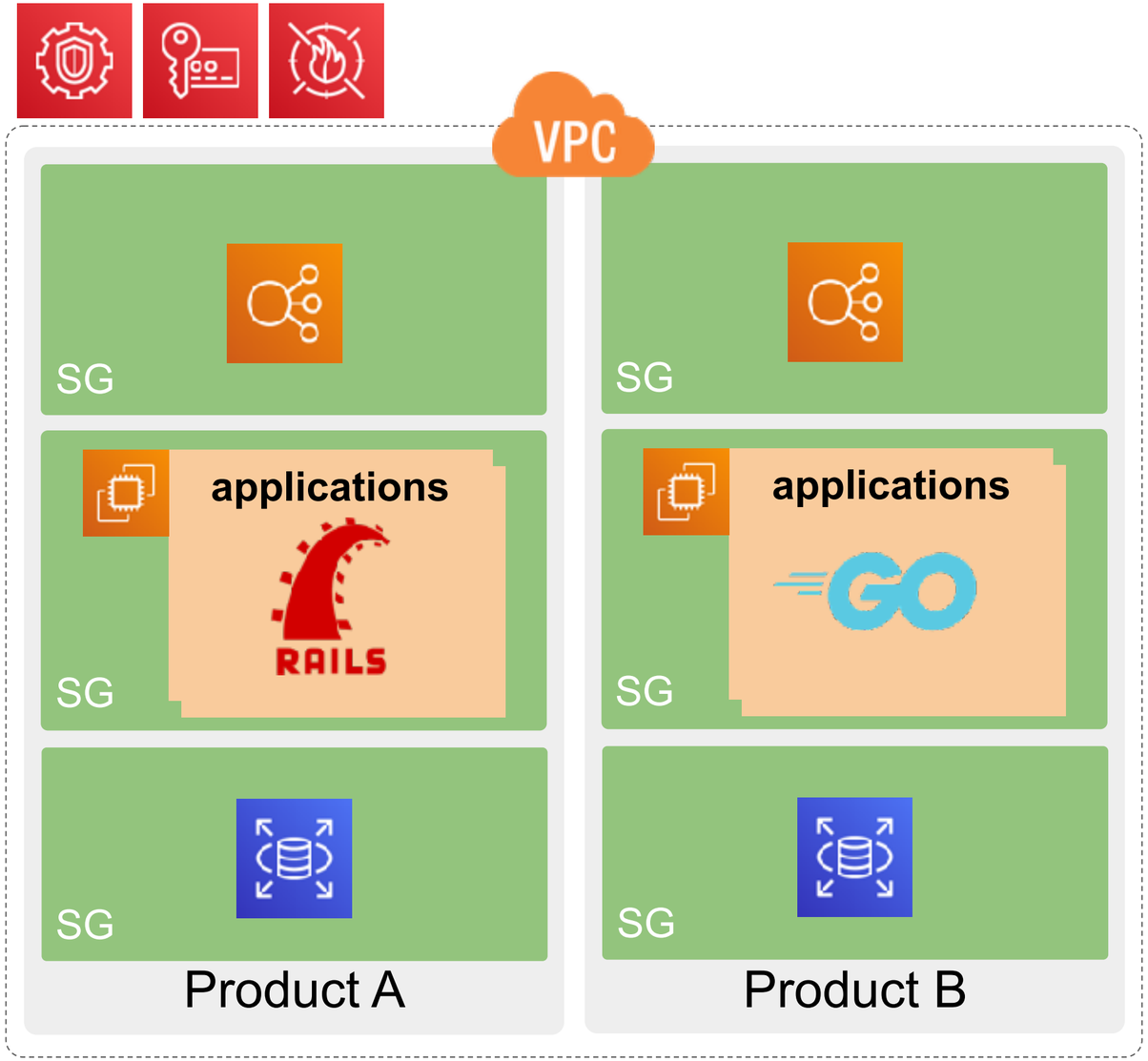
- **SRE**
  - 各種アップデート補助、インシデント対応補助、クラスタ作成補助、ツールの検証/作成、OSSへのコミット
- **サービス基盤**
  - 共通で使うライブラリを整備
- **マイクロサービス委員会**(SREとサービス基盤を含む各サービス担当者で構成)
  - 共通の方針や仕様の決定、情報共有、横展開

Section

04

EKSをマネージドサービスと組み合わせてクラスタの運用コストを抑える

Kubernetesにどこまで任せる？



kubernetes

- Application
- Database
- LoadBalancer
- Security
- Auth

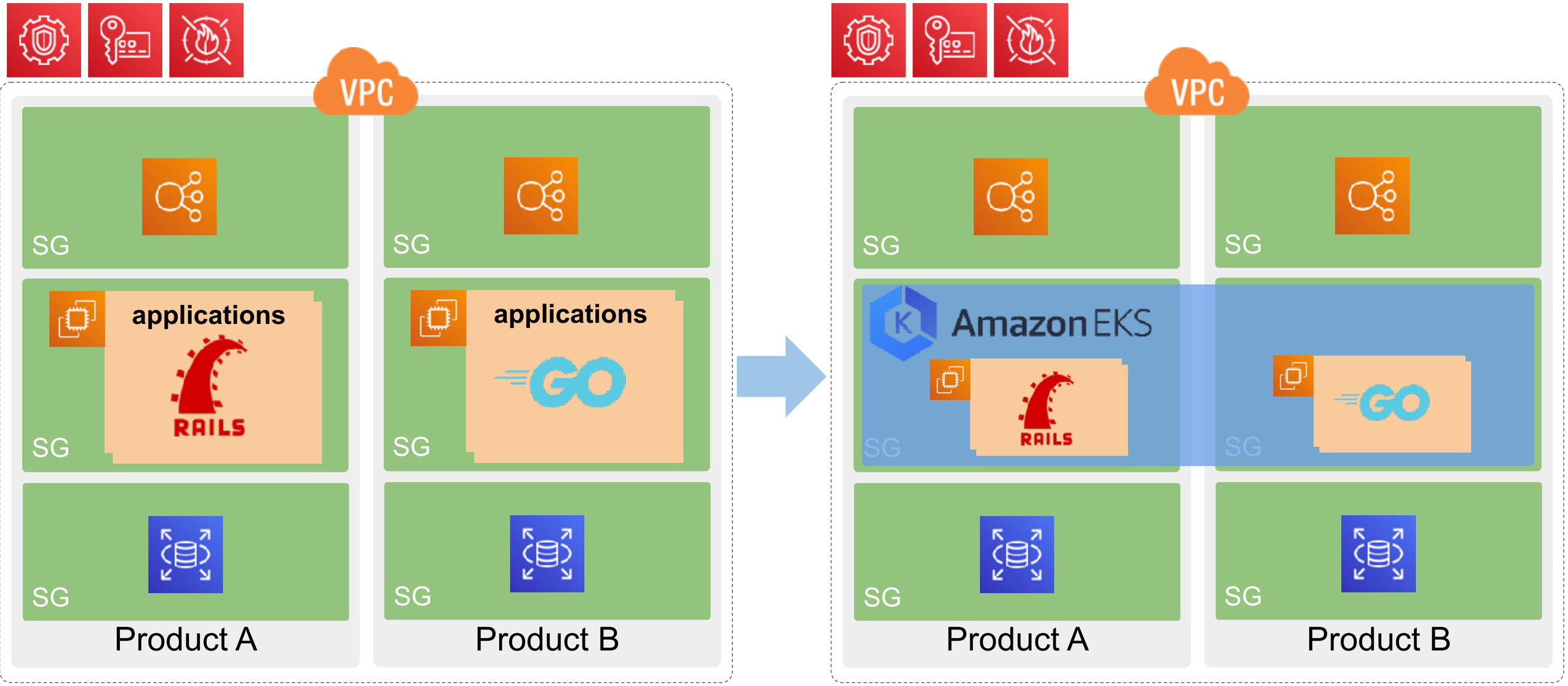
Kubernetesでなんでもやろうとしない

マネージドサービスは積極的に利用し、
システムに Kubernetes(EKS) を組み込む

枯れた運用ノウハウは最大限に活かす

EKSをマネージドサービスと組み合わせてクラスタの運用コストを抑える

Kubernetes はアプリケーションを動かすことだけに利用する

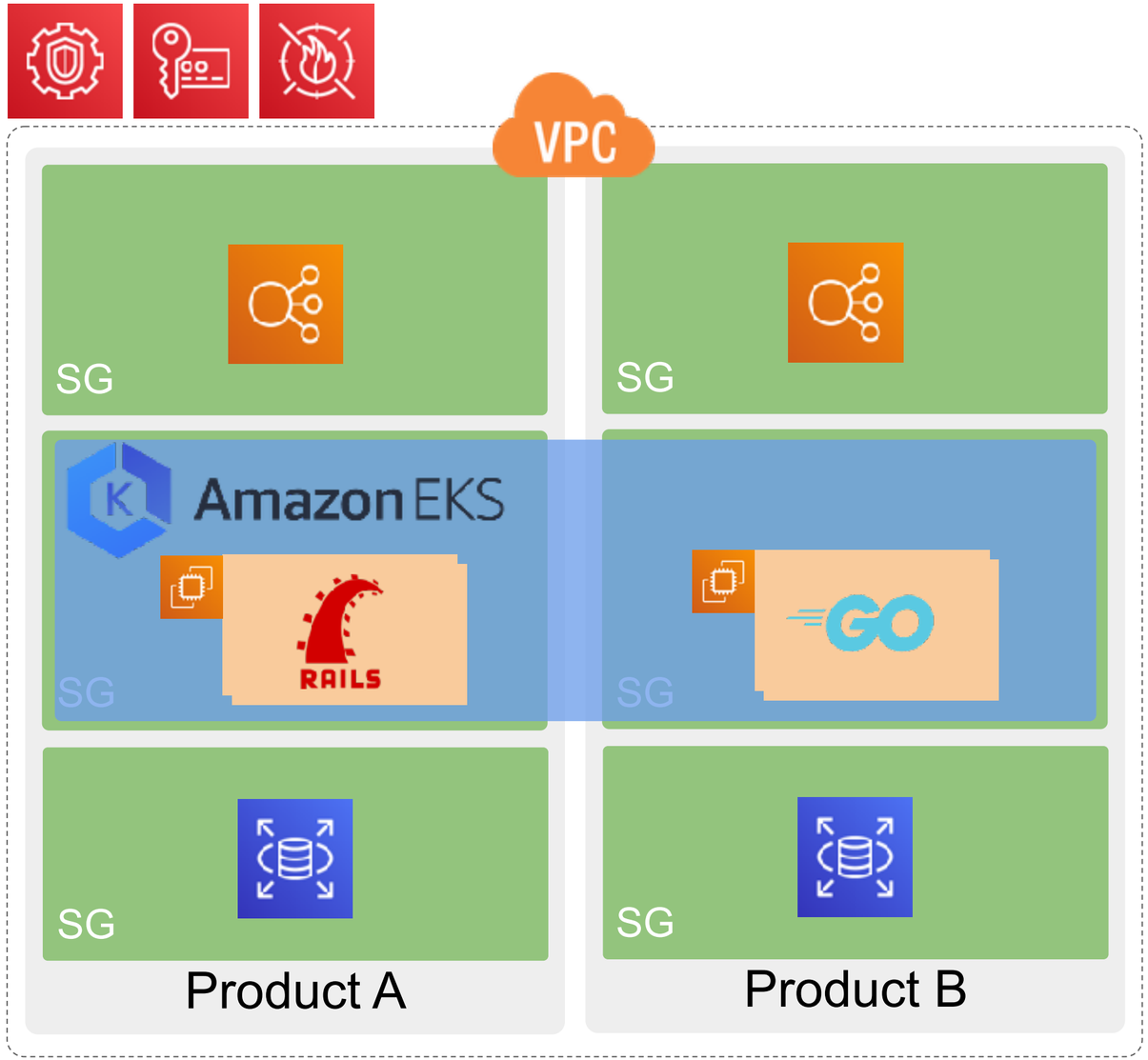


マネージドサービスと Kubernetes の得意分野が活きる

Security
GuardDuty/IAM/
WAF

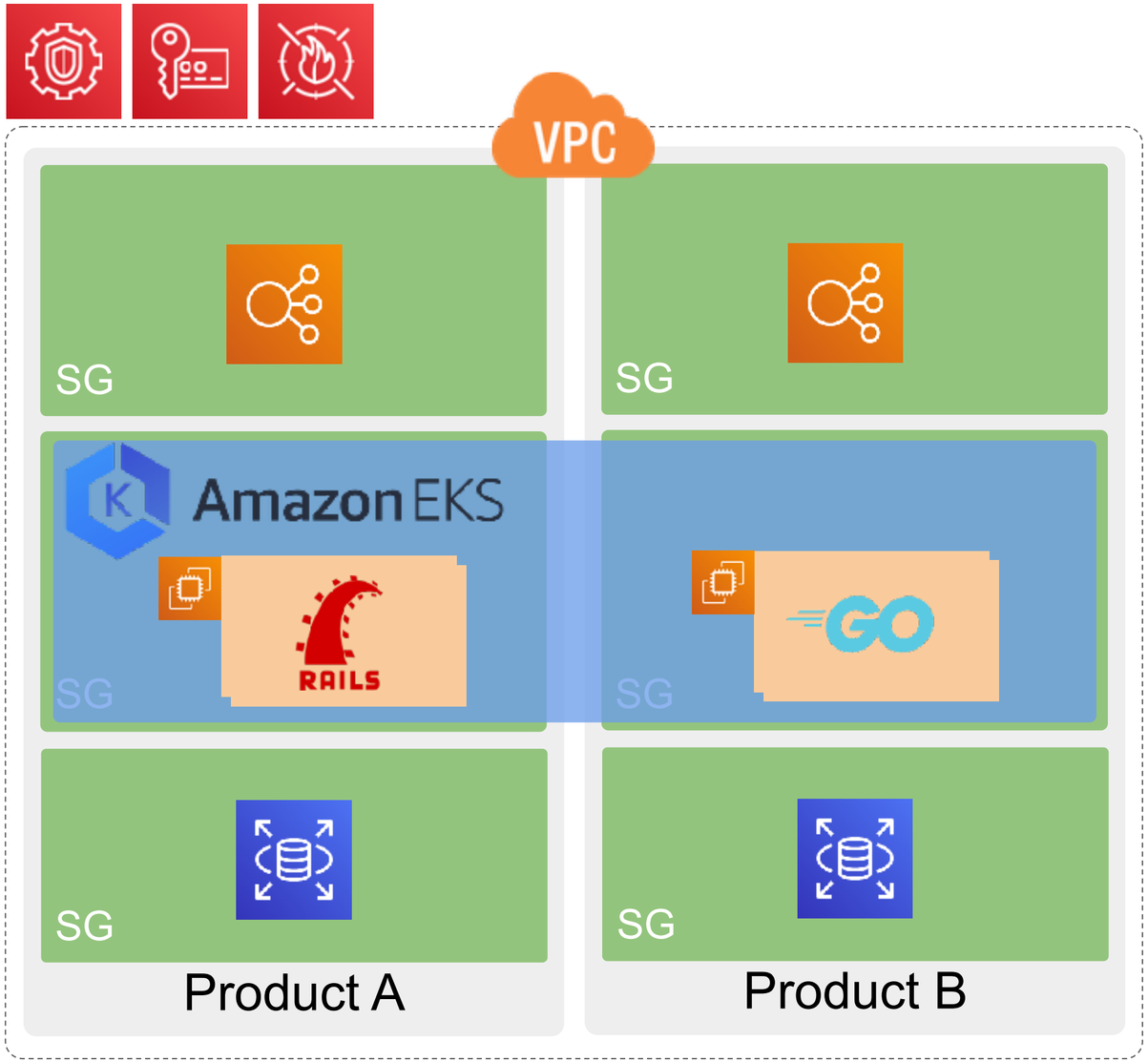
Load Balancer
Application/Classic
Load Balancer

Databases
MySQL/Redis/
ElasticSearch



- 宣言的デプロイ
- 自動配置
- セルフヒーリング
- オートスケーリング

部品の交換をやりやすい状態に保つ



より良いものが出てきたときにそれを取り込みやすい状態にしておく

AWS App Mesh

Istio

Knative

EKS on Fargate

ECS on Fargate

Next generation LB

Next generation DB

Section

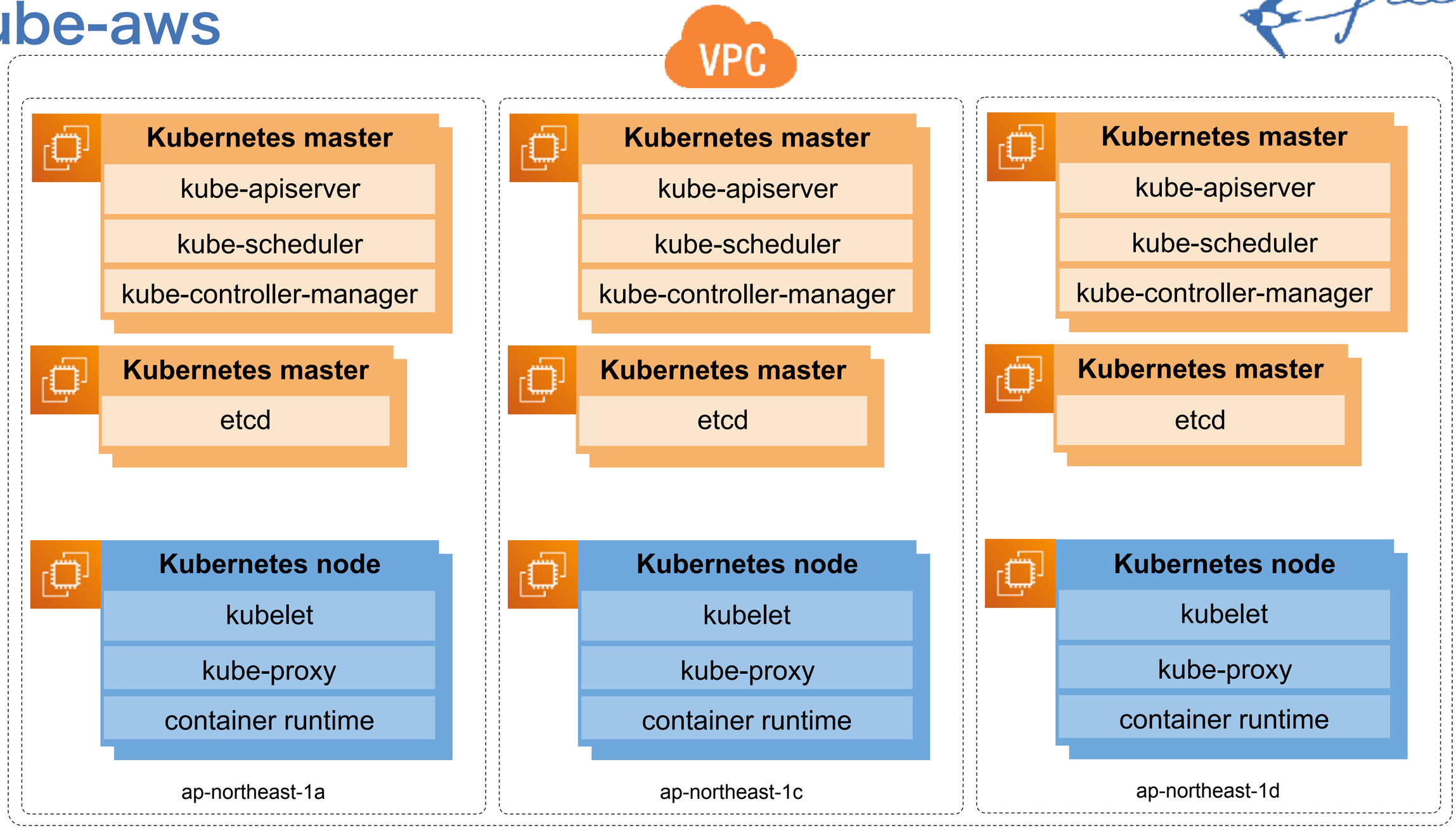
05

マルチテナントからシングルテナントなEKSに移行した実例

freee で Kubernetes が本番導入されたのは約1年前

- 新しいマイクロサービスが出てきたことがきっかけ
- TerraformによるAWSリソースのコード化は一部で導入済み
- Kube-aws を採用
- マルチテナント

kube-aws


VPC
Kubernetes master
kube-apiserver
kube-scheduler
kube-controller-manager
Kubernetes master
etcd
Kubernetes node
kubelet
kube-proxy
container runtime
ap-northeast-1a
ap-northeast-1c
ap-northeast-1d

EKS 2018/12/20 Tokyo Region

- kube-aws のクラスタの証明書が3月13日に切れる
- 証明書の入れ替えはちょっと面倒でサービスメンテが必要
- もうEKSに移行してしまおう！

EKS 移行プロジェクト

- 1月下旬頃からスタート
- 3月13日までに全プロダクトを移行する
- **シングルテナントに変更**する
- 必要な**AWSリソースは開発チーム主導で用意**してもらう
- **Kubernetesクラスタも開発チーム主導で構築**してもらう

SREから開発チームへ権限委譲を果たし、**開発チームにサービスの運用をおまかせする**ことが最大のミッション

プロジェクトの規模感

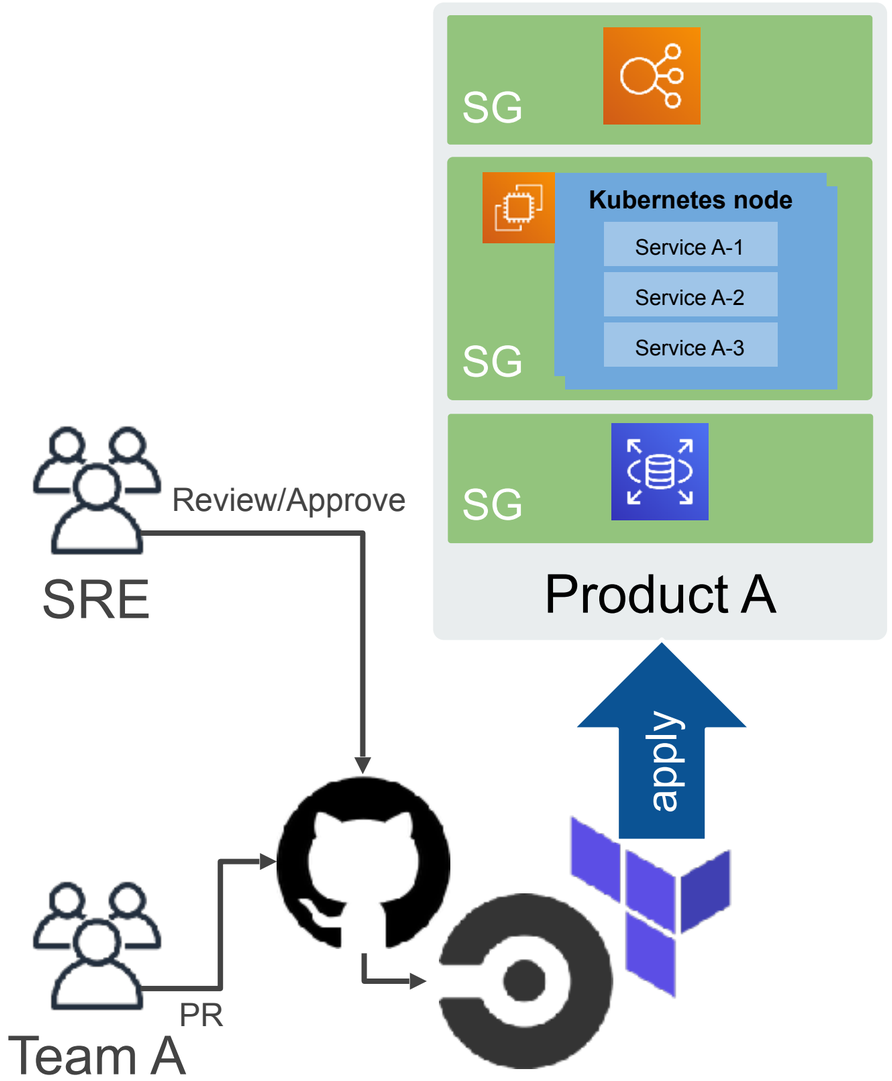
- kube-awsでもともと動いていたプロダクト数: 7
- EKSに移行したプロダクト数(移行中に4つ増えた): 11
- クラスタ総数: 26 (staging環境を含む)
- 関わった人数: 約33人

EKS 移行プロジェクトで活躍したツール

- Terraform
- kubectl
- eksctl
- helm/helmfile
- eksclst

Terraform

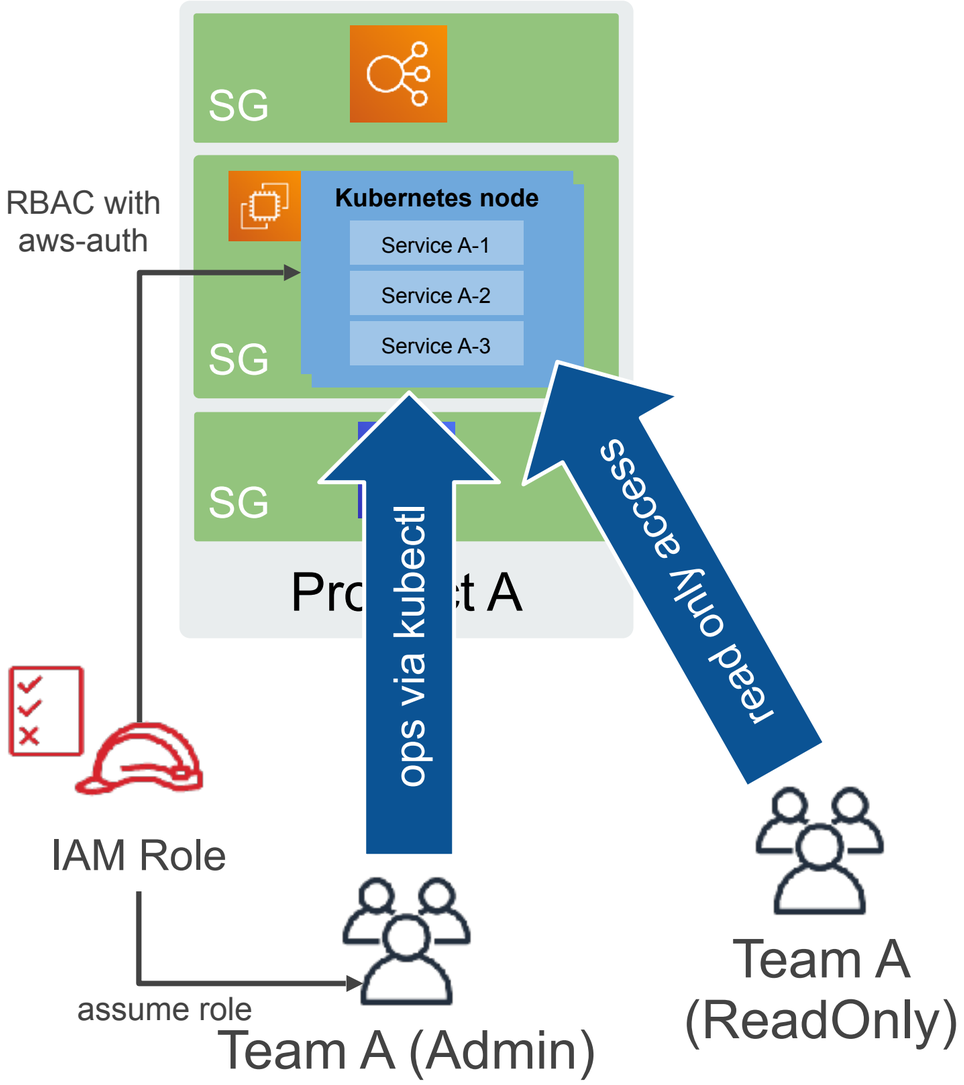
必要なAWSリソースはすべてTerraformで用意/SREのレビューを経てApply




```
resource "aws_lb" "product-a-internal" {
  name = "product-a-internal"
  internal = true
  load_balancer_type = "application"
  security_groups = ["${var.lb_security_groups}"]
  subnets = ${var.subnets}
  ip_address_type = "ipv4"
  enable_deletion_protection = true
}

resource "aws_route53_record" "product-a-internal" {
  zone_id = "${var.route53_hosted_zone_id}"
  name = "${var.route53_dns_name}"
  type = "A"
  alias {
    name = "${aws_lb.product-a-internal.dns_name}"
    zone_id = "${aws_lb.product-a-internal.zone_id}"
    evaluate_target_health = true
  }
}
```

kubectl

aws-auth を利用して IAM Role と紐付けて権限を絞って利用



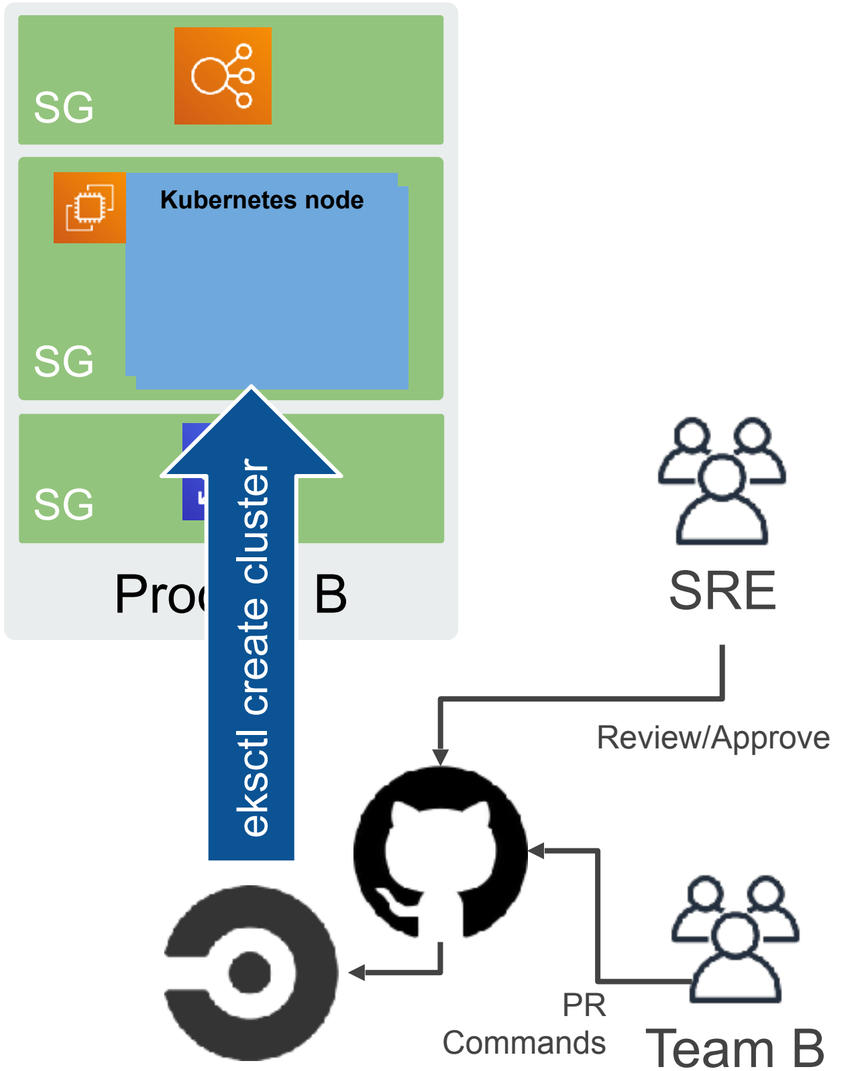

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapRoles: |
    - rolearn: {{ .Values.rolearn }}
      username: system:node:{{`{{EC2PrivateDNSName}}`}}
      groups:
        - system:bootstrappers
        - system:nodes
    - rolearn: arn:aws:iam::<ID>:role/team-a-admin
      username: team-a-admin:{{`{{SessionName}}`}}
      groups:
        - system:masters
    - rolearn: arn:aws:iam::<ID>:role/team-a-readonly
      username: team-a-readonly:{{`{{SessionName}}`}}
      groups:
        - system:authenticated
```

eksctl

cluster.yaml でクラスタを定義、eksctl create cluster で作成

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cluster-name
  region: ap-northeast-1
  version: "1.13"

vpc:
  id: “*****”
  cidr: "10.0.0.0/16"
  subnets:
    private:
      ap-northeast-1a:
        id: “*****”
      ap-northeast-1c:
        id: “*****”
```

```yaml
nodeGroups:
  - name: nodegroup1
    instanceType: r5.large
    desiredCapacity: 2
    availabilityZones:
      - ap-northeast-1a
      - ap-northeast-1c
    privateNetworking: true
    securityGroups:
      attachIDs:
        - ******
    iam:
      withAddonPolicies:
        imageBuilder: true
        autoScaler: true
      attachPolicyARNs:
        - arn:aws:iam::aws:policy/*****
```

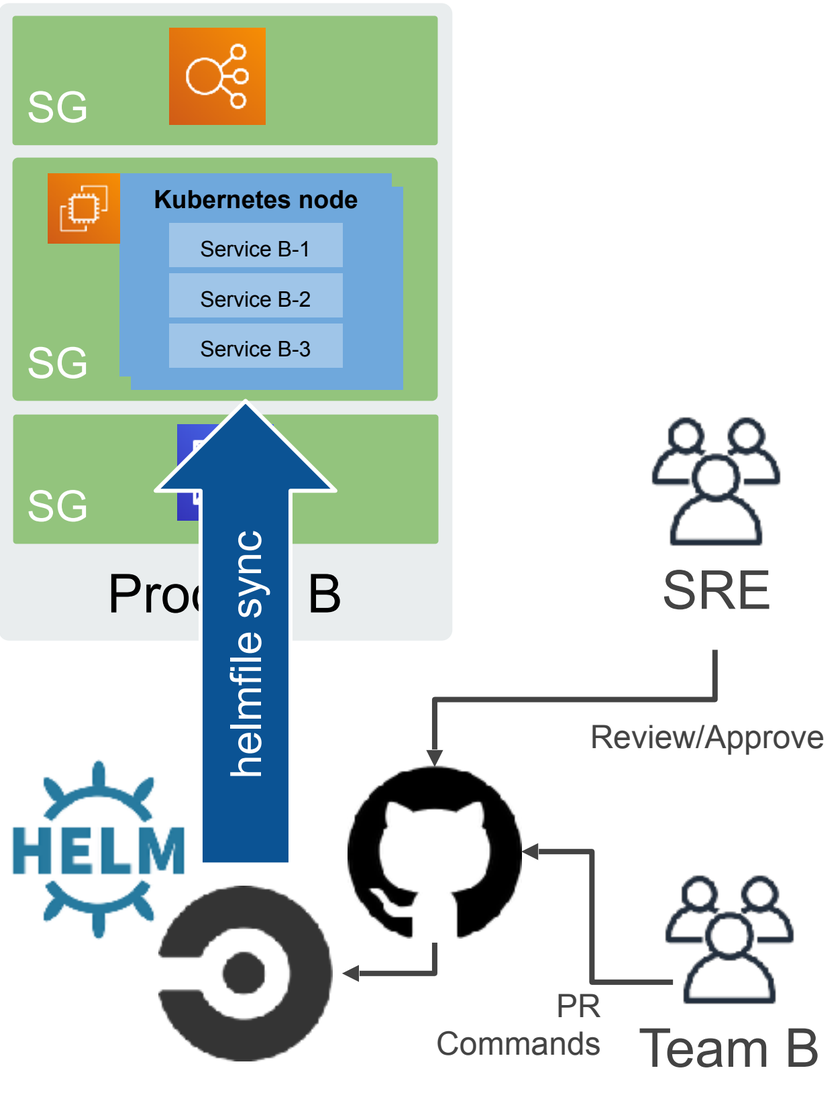


Helm/Helmfile によるアプリケーションデプロイ

GitOps で Kubernetesのマニフェストを安全にデプロイ

- Helm(Helm Chart)
  - The Kubernetes Package Manger
  - マニフェストをパッケージ化
  - よくあるツールのテンプレ
- Helmfile
  - Helm Chartの依存関係をファイルでファイルで定義
  - helmfile sync
  - helmfile diff
  - helmfile delete

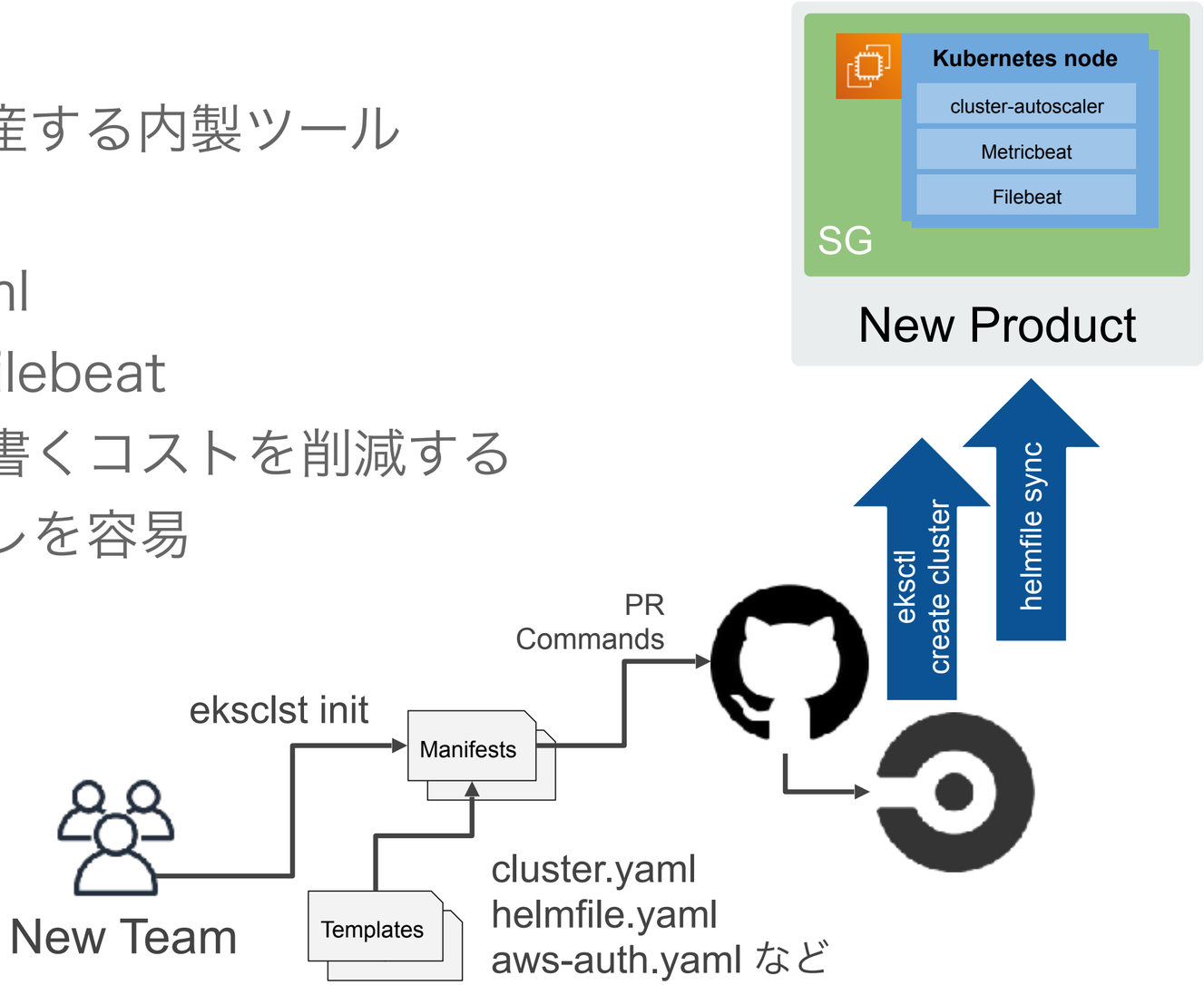
```
environments:
  production:
    values:
      - production.yaml

releases:
  - name: kube-state-metrics
    namespace: kube-system
    chart: stable/kube-state-metrics
    version: 0.13.0
  - name: metricbeat
    namespace: kube-system
    chart: stable/metricbeat
    version: 1.2.1
    values:
      - values.yaml.gotmpl
```



eksclstによるクラスタのテンプレ化

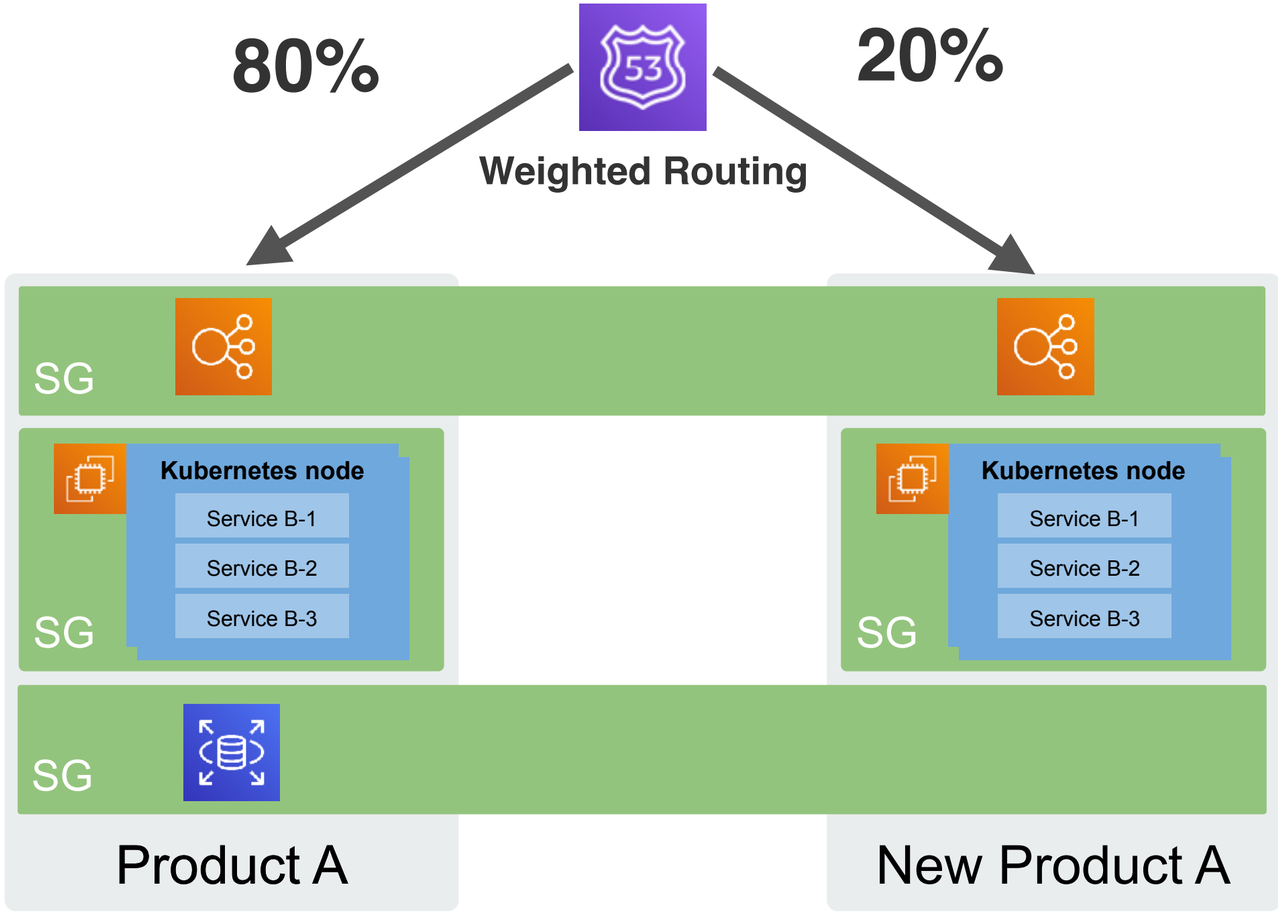
よくある構成のクラスタテンプレ化し、クラスタの作成/複製を容易にする

- クラスタを量産する内製ツール
- cluster.yaml
- aws-auth.yaml
- metricbeat/filebeat
- など、一から書くコストを削減する
  ためにテンプレを容易



移行作業

- 同じ構成のクラスタを用意
- AWSリソースは共有できるものは共有する(DBは必須)
- Route53のWeighted Routingを利用して徐々にリクエストを流し込む
- サービスによってはRoute53ではなくLBを共通化して、Kubernetes node を差し替える方法を使用
- **ノーメンテで切り替え**



Kube-aws上のプロダクト
(実際はマルチテナント)

Amazon EKS
EKS上のプロダクト

プロジェクトの成功の要因

- 関わった開発チームのKubernetesへの意欲が強い
- ドキュメントを皆で編集しながら(情報交換を密にしながら)進めた
- 完璧ではないドキュメントもメンバーが意図を汲み取って理解してくれた
- 最後までモチベーションが下がらなかった

まとめ

インフラリソースのコード化とKubernetesのシングルテナント化でサービスの運用コストを分散させる

- インフラリソースのコード化は必須、これを足がかりに権限委譲が進む
- クラスタ運用を開発チームにお願いするにはシングルテナントがおすすめ
- クラスタ自体の運用コストを抑えるにはマネージドサービスをうまく使う
- 開発チームにKubernetesに対する意欲があることが重要、理解されない場合は辛みが増えるだけ

スモールビジネスを、世界の主役に。

アイデアやパッションやスキルがあればだれでも、
ビジネスを強くスマートに育てられるプラットフォーム