Gunosy

グノシー におけるEKS活用事例

株式会社 Gunosy
技術戦略室SRE部　城井大輝

2020年3月20日

自己紹介

Gunosy

■ 城井大輝 / Daiki Shiroi

■ 2019/04/01 ~
  – SRE Engineer at Gunosy Inc.

■ GitHub: @RossyWhite
■ Twitter: @rossywhite_

~ 今回のテーマ ~
# グノシーにおけるEKS活用事例

株式会社Gunosy

Gunosy

ギリシャ語で「知識」を意味する「Gnosis（グノーシス）」＋「u(“you”)」
「”Gnosis” for “you”」あなたのための知識
＝情報を届けるサービスを提供し続ける、という意味

- 2012年11月創業
- 2015年4月東証マザーズ上場
- 2017年12月東証第一部に市場変更
- 従業員数 215 名
  （2019年5月末現在 連結ベース）
- 事業内容
  - 情報キュレーションサービスその他メディアの開発及び運営
- 提供サービス
  グノシー、ニュースパス、LUCRA（ルクラ）

企業理念「情報を世界中の人に最適に届ける」

お話すること

Gunosy

■ コンテナ移行によって得られたメリット

■ プロダクション環境でEKSを運用するための工夫
  – 良かったことも苦労していることも。。

今日のゴール

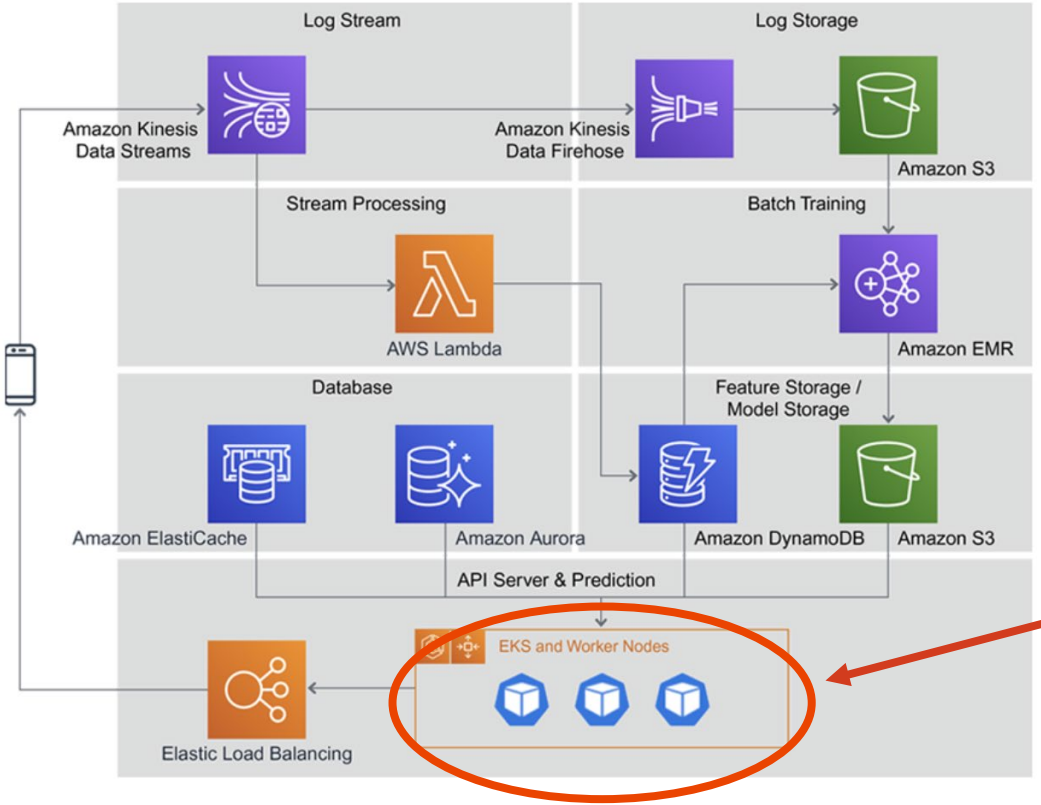
Gunosy

■ EKS導入のメリットや課題をイメージできるようになる

グノシーのシステムの概要と課題

グノシーのアーキテクチャ

Gunosy



今日の話の中心

https://aws.amazon.com/jp/solutions/case-studies/gunosy/ より

従来の主な使用技術

Gunosy

- 構成管理
  - **AWS OpsWorks**
- 開発言語
  - Golang
- データストア
  - RDS, DynamoDB, ElastiCache...
- モニタリング
  - CloudWatch, Datadog..
- その他
  - CircleCI

従来の課題

Gunosy

全てがOpsWorks文脈という訳ではないが。。。

- Chefのクックブックの管理コスト
- インスタンスの立ち上げ速度
  - Setupが遅い...
- スケーラビリティ
  - 柔軟にスケールできない
  - スパイキーなアクセスで落ちる
- コスト効率が悪い
  - サービスごとにVMが存在
  - スポットインスタンスが使いにくい
- 手作業でのデプロイ
- ローカル開発環境が貧弱

EKSへの移行

移行により解決したかったこと

Gunosy

■ コストダウン

■ ポータブルなインフラ

■ 開発環境の改善
- CI/CD, リリース頻度
- ローカル開発/テスト環境の整備

■ 柔軟なスケーリング

など

Kubernetes (EKS) 選択の背景

Gunosy

- Kubernetes自体はオープンソース
  - 情報量・最悪コード読める
- コントロールプレーンはマネージド
- 一貫性のある定義方法、拡張方法
- 利用側にとってはシンプル(`kubectl apply`するだけ)
- 新規プロダクトでの事例
- Kubernetesのエコシステム

移行プロジェクト概要

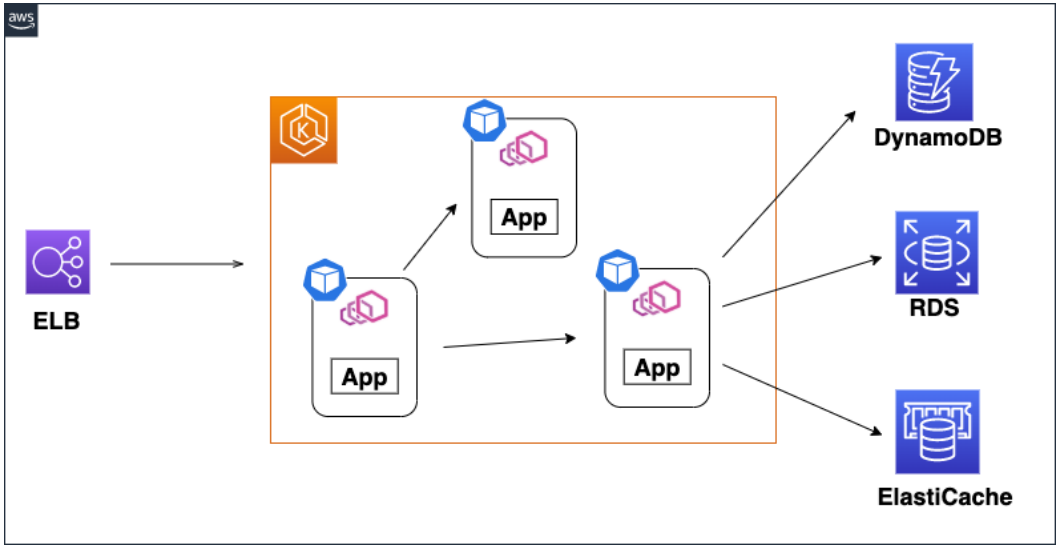
Gunosy

- 4人くらい
  - サーバーサイドエンジニア + SRE
- アプリのコンテナ化 + クラスタ構築
- 半年くらい?

クラスタ作成

Gunosy

- PrivateSubnet内にASG
- App + Envoy(サイドカー)
  - Istioとかは入れてない
- トラフィックはNodePort経由
- ログはfluentdで収集
  - Papertrailに集約



アプリのコンテナ化

Gunosy

■ Dockerで動作可能に
- カスタムjsonの内容の移植
  - ConfigMaps, Secrets
  - 環境変数

■ Podとして動作可能に
- 依存をサイドカーに寄せる
- プロキシ(Envoy)通じて通信できるか

移行による嬉しさ

Gunosy

■ 大幅にコストカット
- クラスタの6~7割くらいがスポットインスタンス
- インスタンス一台あたりの稼働率が上がった
- **1/3くらいコストが削減された**

■ 柔軟なスケーリング
- ASGのオートスケーリング
- インスタンスの初期化が軽くなった

■ 開発速度の向上

プロダクション運用を支える仕組み

プロダクション運用を支える仕組み

Gunosy

- スケーリング戦略
- Infrastructure as Code
- 開発環境

スケーリング戦略

スケーリング戦略

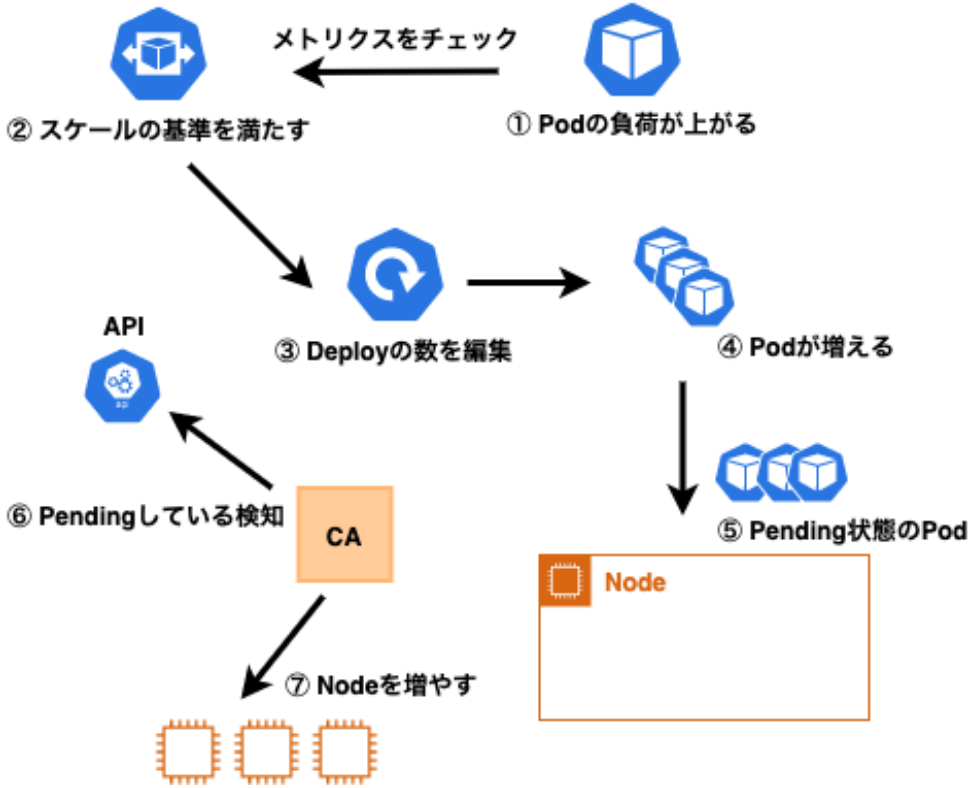
Gunosy

Horizontal Pod Autoscaler(HPA) + **Cluster Autoscaler**(CA)

- HPA
  - Podの負荷に応じてPod数を増減させる
    - CPU, Memory, CustomMetrics
- CA
  - 起動できないPodがある場合にノードを増やす

スケーリングの流れ

Gunosy



- deploymentレベルでHPAを設定しておけば、ノードのCPU使用率を考えなくても良いのでシンプル

HPA + CAだけではうまくいかなかった

Gunosy

- Podがpendingしないとスケールアウトしない
  => 負荷に先回りすることはできない

- スケールまでのラグの要素
  - HPAのインターバル
  - CAのインターバル
  - ノードの初期化
  - Podの初期化

3~3.5minくらいはかかる
(もちろん環境次第)

Push通知時の瞬間的なリクエスト数に対応できない

Push通知時のスケーリング

Gunosy

■ Push送信時には、通常の10~30程度のrpsが瞬間的に発生
　　=> HPA+CAの方式だと間に合わない

■ 事前にスケールさせておく必要がある

■ Push送信イベントをlambda受け取ってHPAの**minReplicas**を更新

その他の機構

Gunosy

■ **VPA(Vertical Pod Autoscaler)**
- PodのResource Request を調整してくれる
- 一部のPodは数を固定して、VPAで調整してる

■ **descheduler**
- ノード間での偏りの調整
- 高負荷なノードからPodをevict
- CronJobとして、2回/day くらい実行している

infrastructure as code

使っている(いた)ツール

Gunosy

- Codenize.tool
- Chef (OpsWorks)
- Terraform
- Kustomize

前提

Gunosy

- インフラのコード管理自体は以前より行っていた

- Codenize.toolsを使用 (OpsWorks自体は管理対象外)
  => 1年ほど前にTerraformに移行開始

- Terraformへの移行とコンテナへの移行が同時に発生

Kubernetesのマニフェストをどこにおくか

Gunosy

Kubernetesのマニフェストも管理しなければなくなった。。。

[初期の判断]

単一のリポジトリでの管理を試みる

- 今までの慣例から自然な流れ
- 一元管理した方が一貫性が保てる

- 開発スピードの低下・SREのレビュー負担

何が問題だったか

Gunosy

- ライフサイクルの異なるリソースが同一リポジトリに存在
  - 開発者が複数リポジトリを触る必要性
  - デプロイのスピードが低下
- SREのレビュー負担
  - コンテキストもあまり分からない
  - 人数的にも厳しい
- リポジトリ自体の肥大化
  - applyが遅くなる

境界線の再定義する必要性

リソースの2つのレイヤー

Gunosy

■ クラスタレベル(サービスが共有して使用)
- Daemonsets, VPC, Subnet...

■ サービスレベル(サービス固有)
- Deployments, Services, ConfigMaps, ALB, IAM, SG….

現状の管理方法

Gunosy

- クラスタ関連のリソース
  - DaemonSets
  - kube-system関連のDeployment
- terraformで管理されてたAWSリソース
- codenize.toolsの残党... 

中央リポジトリ

- 各Microserviceのリソース
  - k8sのマニフェスト
    - Deployments
    - Service
    - ConfigMaps

各マイクロサービス
のリポジトリで管理

Kustomize

Gunosy

主にk8sのマニフェストの管理、Applyに使用

```
~/someApp
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── development
    │   ├── cpu_count.yaml
    │   ├── kustomization.yaml
    │   └── replica_count.yaml
    └── production
        ├── cpu_count.yaml
        ├── kustomization.yaml
        └── replica_count.yaml
```

公式からの抜粋
(実際も似たような感じ)

- 軽い
- 分かりやすい
- 環境ごとに簡単に値を変更可能

(若干Helmに移行検討中)

できている事、できていない事

Gunosy

■ できていること
- KubernetesのyamlIに関してはきれいに分離されている

■ できていないこと
- 以前からterraformで管理されてたIAM・SGなどがサービス側に移譲できてない

開発者に自由を与えつつ、守るべきは守る仕組みをどう作るか?
- 規模感、組織の文化にもよる

開発環境

従来の開発環境

Gunosy

■ localではdocker-composeで動かすが、デプロイはOpsWorksで行う
=> **localとremoteでの環境差**が大きい

■ リモートの環境は供用として使用

お気持ち

Gunosy

■ 本番反映される環境と開発環境の差分は小さくしたい

■ ローカルでテストデータ生成は嫌

■ でも手軽さは欲しい

Skaffold

Gunosy

■ GoogleがOSSで開発しているCI/CDツール

■ コードの変更を検知して、コンテナのビルド~デプロイまで行う

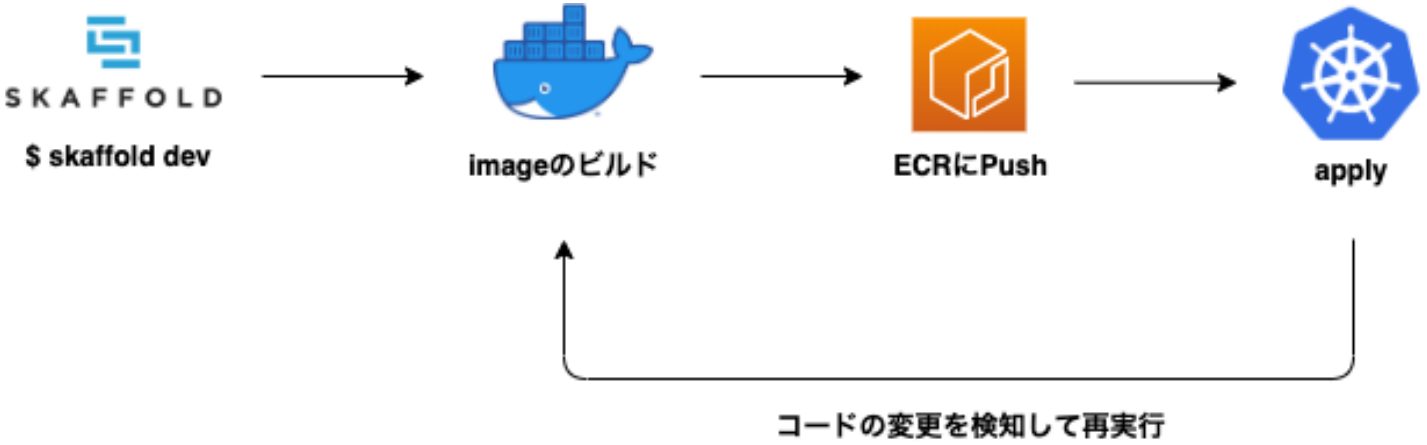
■ リモート環境でローカル開発っぽいことができる

SKAFFOLD

Skaffold

Gunosy

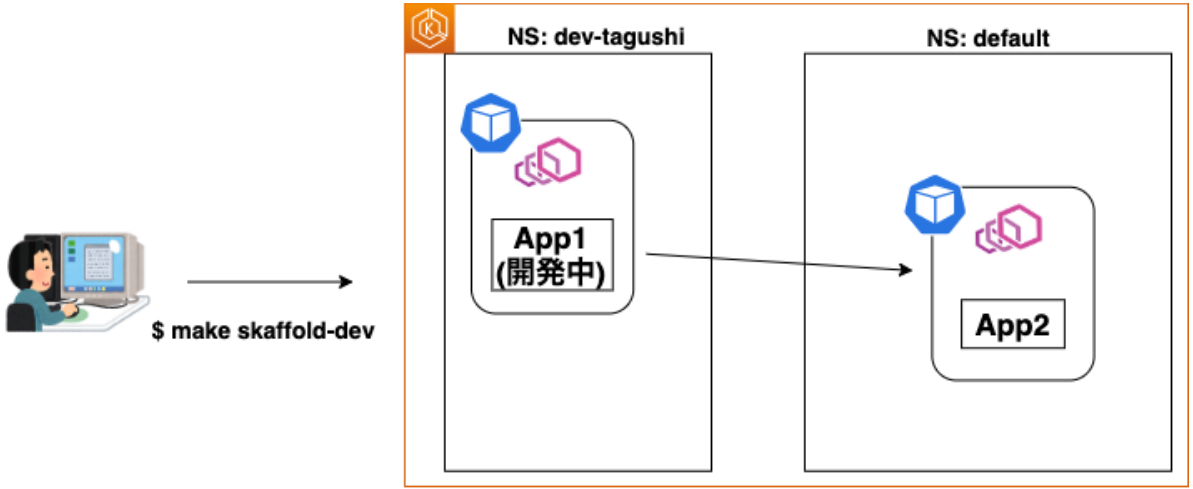
■ コードの変更を検知 => 自動で再デプロイ



現在の開発の流れ

Gunosy

■ ローカルではなく、Stagingクラスタを使用
- skaffoldの起動時に開発者ごとの一時的なNamespaceを作成
- ECRにpushされるimageはタグで識別
- 依存しているアプリはdefault namespaceにあるPodにアクセス



この方法の嬉しさ

Gunosy

■ localで動かすのと違って、本番同等の環境で開発可能

■ 依存先のマイクロサービスは常に最新状態
  – localで動かすなら毎回pullして来ないといけない

■ 開発者ごとに隔離された環境を提供可能

今の課題

開発速度と信頼性の担保

Gunosy

■ Envoyのyamlがつらい
  – そもそもk8sのyamlもそこまで優しくはない

■ キャッチアップの努力は必要だが、負荷は減らしたい

■ 最低限必須な設定は担保したい
  – 信頼性に影響が出るようなもの
  – 命名規則だったり、社内で使っているタグ
  – 監視の設定し忘れ防止

■ Canary Releaseもしたい

<u>今はmicroservice用のboilerplateの提供に留まっている</u>

まとめ

まとめ

Gunosy

- 移行を頑張るだけの価値はある
  - コスト削減
  - 開発速度向上
- Kubernetesは難しくない
  - 勉強は必要だがそれは他も同じ
  - プラクティスは出揃いつつある
  - 周辺ツールを使えばなんとかなる(EKSもある)
- k8sそのものより、エンジニアの責務の分担が難しい

Gunosy

情報を世界中の人に最適に届ける