コンテナうまみつらみ

Kubernetes初心者がEKSと格闘した1年を振り返る

（株）いい生活　多田 吉克（@ta_dadadada / Tada Yoshikatsu）

自己紹介

多田 吉克（@ta_dadadada / Tada Yoshikatsu）

- （株）いい生活 サービスプラットフォーム開発部エンジニア
- 物理学科の修士課程修了
- いい生活に新卒で入社、もうすぐ丸3年
- 2年目までは平凡なアプリケーションエンジニアだった
    - API の機能改修や品質改善（クエリの高速化とか）やっていた（主に Python）

会社概要

商号	株式会社いい生活
主力事業	不動産業界向けクラウドサービスの提供
設立	2000年1月21日
上場市場	東証二部 [3796]
資本金	628,411,540円 (2019年3月末現在)
従業員数	155名 (2019年3月末現在)
拠点	東京本社　〒106-0047　東京都港区南麻布5-2-32 興和広尾ビル
	大阪支店　〒530-0011　大阪府大阪市北区大深町4-20 グランフロント大阪 タワーA
	福岡支店　〒810-0001　福岡県福岡市博多区博多駅前3-25-21 博多駅前ビジネスセンター
	名古屋支店 〒450-6419　愛知県名古屋市中村区名駅3-28-12 大名古屋ビルヂング

不動産業界のDXを推進

統合不動産業務ソリューション

不動産コミュニケーション プラットフォーム

不動産業界向けの
様々なソリューションを
SaaS・サブスクリプ
ションモデルで展開

MAU
1,447 法人
4,447 店舗
14,728 人

詳細は弊社サービスサイトへ
https://www.es-service.net/

Contents

- ことの起こり
- 今稼働しているモノ
- 稼働までの苦労
  - コンテナを作る、 CI/CD、監視、Envoy つらい
- 稼働してからの苦労
  - HPA、スループット改善、 Pod のスケーリングとの格闘、監視系運用のつらみ
- これからの課題

ことの起こりは 1年前

CTO「4月から新規プロダクト
よろしく、EKS で」

僕「？！」

ともあれ、4月からプロジェクトは始動

- チームメンバーは PL の自分入れて3人でスタート
  - うち2人はアプリケーションエンジニアとDBエンジニア
  - がっつりクラウド・コンテナでの開発経験のあるメンバーは無し
- EKS の採用は内定状態
  - クラウド × コンテナで進めたかった
  - 社内的な事情も有り

プロダクトの概要

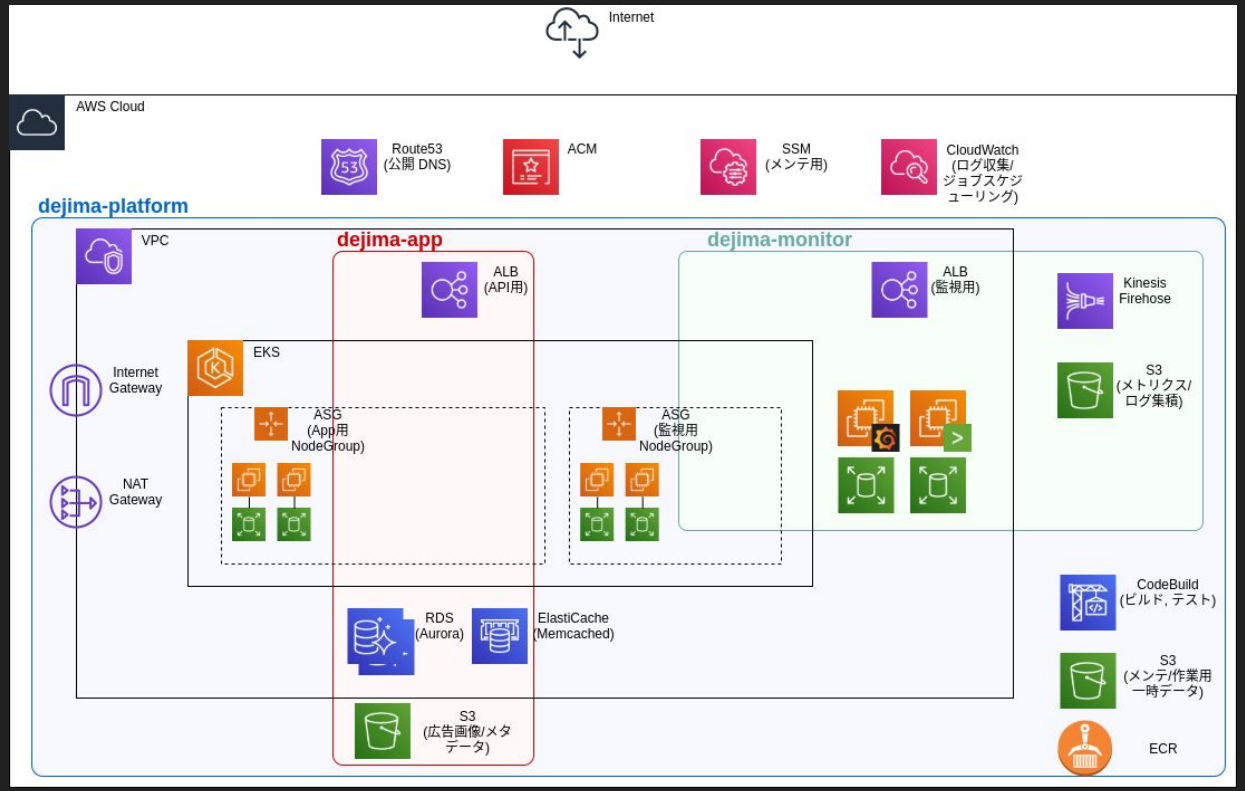
- 当社の主力製品（いい物件One）から/への データ連携を行う外部連携用システムの新規構築
  - 物件広告情報（テーブルデータ、画像データ）の取り出しと、エンドユーザからの問い合わせ情報を取り込み
  - いい物件One のバックエンド API (Python) は**デスクトップアプリケーション用**に作り込まれた（レガシーな）システムのため外部公開するには使いづらく、かつ月一回程度メンテナンスタイムが存在してしまうため、**より可用性の高い**システムを別途構築
  - 画像部分とユーザからの問い合わせ部分は既存プロダクトがあったため、新規に作り込んだのは広告情報の部分
- さっくり言えば、**鎖国**気味の既存システムに接続する**オープンな使いやすい**APIを作ろう！という話

dejima（出島）と名付けた

dejima の現在

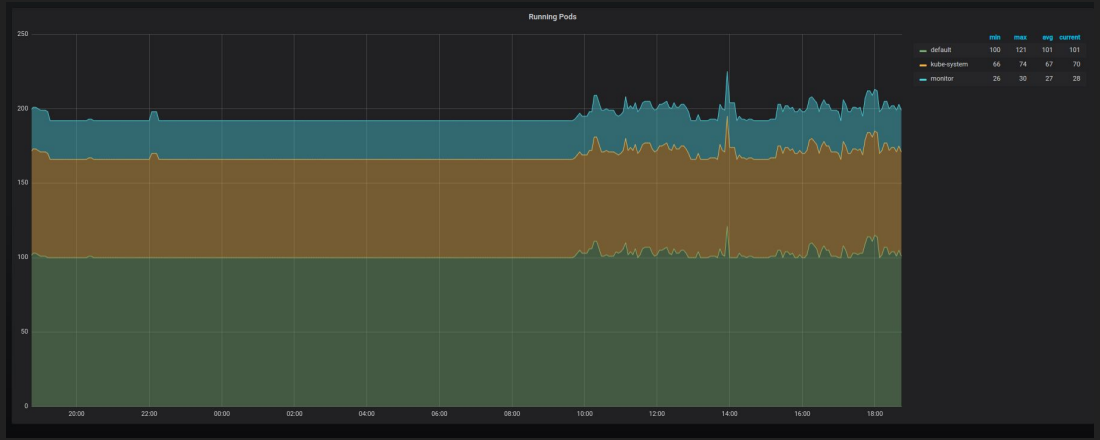
アーキテクチャ

- ノードグループはそれぞれ 3AZ で構成



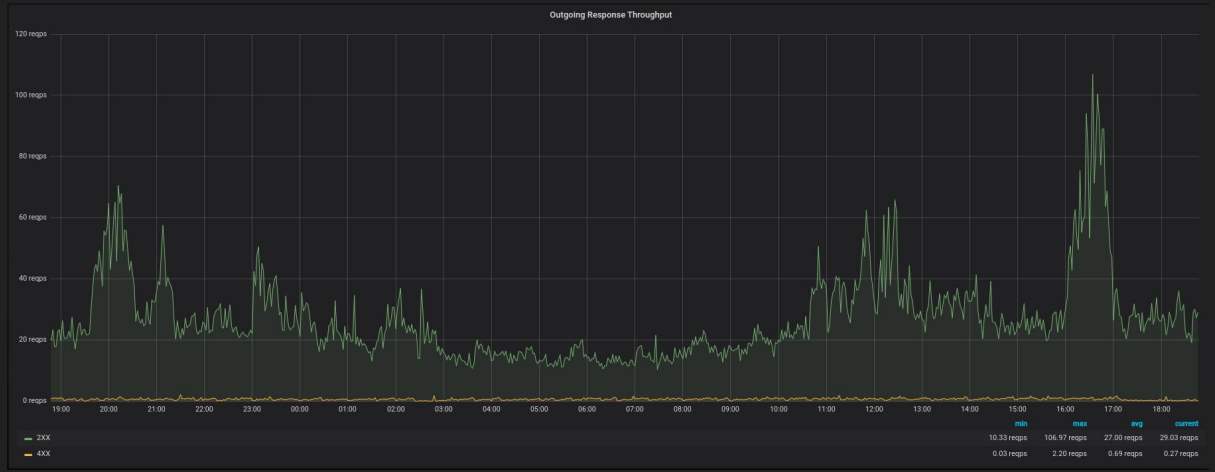
ガンガンスケールしている

- 21ノード（最大）
- 13 マイクロサービス
- 200pods ~



とあるマイクロサービス

- 通常 50-100RPS くらいだが **500**RPS くらいにスパイクすることも
- レイテンシーはまちまち



形になるまで
（〜1st リリース）

EKSでデプロイできるまで

- eksctl はあまり使わずクラスタは **CloudFormation** で作成
  - クラスタ作成までは当社 CTO の *素振り* を後追いしたのでスムーズだった
- 「とりあえずデプロイできる」段階までも苦労はあまりなかった
  - アプリケーションをコンテナで包んで deployment 書くだけならば、
    正直見様見真似でもどうにでもなったので、そこから改善していった
- サービスの公開には **ALB Ingress Controller** を利用
  - k8s の外側のリソースを（あまり）気にせずに LB 立てられて便利だった

使いやすいコンテナに仕上げる工夫

- 設定値を注入可能にする
  - アプリケーションの設定値を環境変数から読むようにする定番戦略
    - アプリケーションのコードレポジトリ側にデフォルトの設定値を持たせた
    - 変数なくてもコンテナ単体で動作する状況の方が開発でも使いやすい
  - k8s 側から環境変数や起動時引数与えることで上書きできるよう設計
    - 非機能テスト段階でのチューニングが容易（アプリケーションコードを触らずに済む）
  - 設定値が無理なく自然にコード管理されている喜び・・・
- コマンドクエリ責務分離 (CQRS) パターン を拡張して適用
  - **同じ**データモデルを扱うサービスでも更新系（コマンド）と参照系（クエリ）を **別のマイクロサービス** ≒ Deployment として扱う
  - 更新と参照で別の言語・フレームワークを使うことが比較的容易に可能
  - 更新と参照では負荷傾向が全く違うこともあるので、分離することで細密なチューニングができるようになった

CI/CD

- 実行環境は既存の社内 CI サーバ（Drone） と AWS CodeBuild を併用
  - コードレポジトリ（GItLab）がオンプレにあり既存のものを使うほうが楽な場面とそうでない部分があるため
- nightly ビルド + デプロイ
  - `develop` ブランチに対してビルド&プッシュを定時実行
  - `ImagePullPolicy: Always` にして `kubectl rollout` を CodeBuild から実行しステージング環境に自動デプロイ
  - タグ付け起因の stable バージョンのビルド＆プッシュ
- 機能テスト（ふるまいテスト）と性能テストを開発環境対して定期実行
  - テストコード自体は Drone でビルドして ECR にアップし、 CodeBuild で実行
- リリースサイクルの高速化に貢献

監視系

- メトリクスは Prometheus で収集、 Grafana で可視化
  - **promethus-operator** by Helm でサクッと構築
  - デフォルトでとれるメトリクスは限られているため、 Prometheus 側のパラメータ変更や各種リソースにアノテーションして Pod 関連のメトリクスも収集
- ログは Fluent Bit + Firehose + Splunk
  - Splunk App で CloudWatch Logs 経由でも取得できるが、遅延があるため Splunk Http Event Collector(HEC) で送信してほぼリアルタイム連動する構成に
- Prometheus のメトリクス長期保存は挫折
  - InfluxDB（時系列DBで、 Prometheus の Long-Term Storage として使える） を EC2 に構築して試した
  - 大量のメトリクスを1台で捌き切れず、開発環境での検証段階で死亡
  - クラスタ化などを考えている余裕がなかったため、断念
  - Thanos?

Envoy つらかった

- Pod を app + sidecar(Envoy) で構成しサービスメッシュを実現、 Istio は使っていない
  - はじめはコンフィグの勘所がわからず
    - Envoy のメリットを頭ではわかっていていても書くのがつらい
    - そもそも設定項目が膨大で初見だと心が折れる
    - yaml でかける点、ドキュメントは結構充実している点、最悪 Envoy のソースコード見ればなんとかなる点は救いだった
  - やってるうちになんとか読み書きできるようになっていったので、慣れ
    - サービスメッシュについては、ちょうどこのあたり考えているときに聞いて大変参考になりました
      - サービスメッシュは本当に必要なのか、何を解決するのか | AWS Summit Tokyo 2019

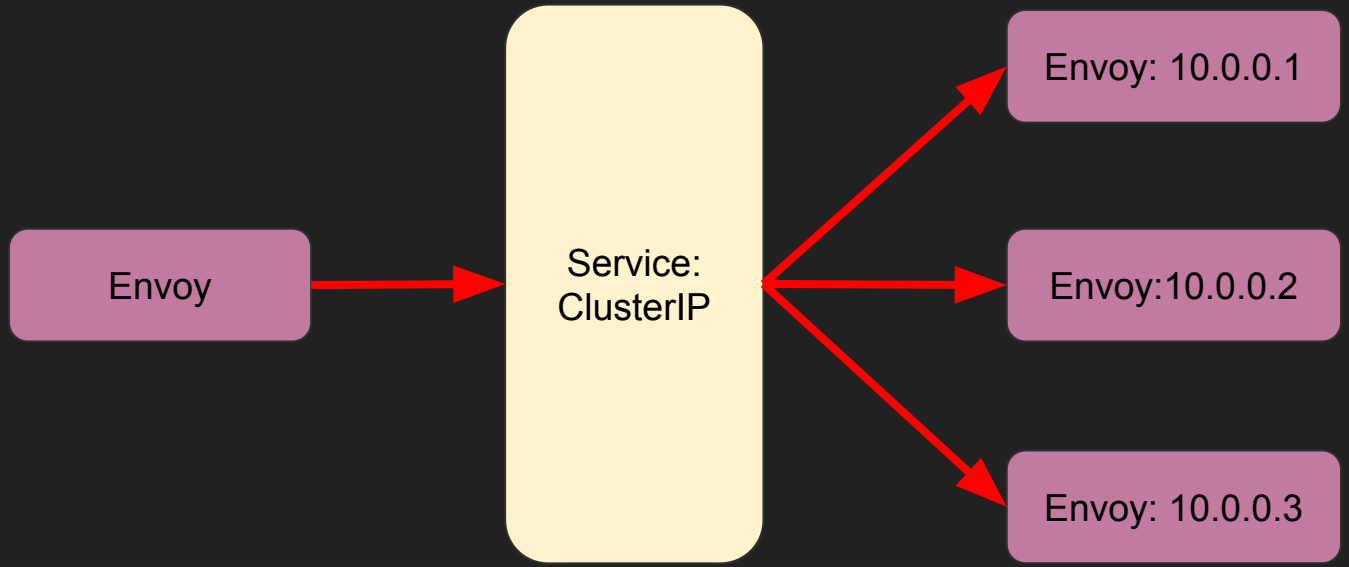
Service: ClusterIP の罠

Envoy to Envoy の通信が Service: ClusterIP だと接続が不安定になる問題

- Gateway 系のエラーが頻発

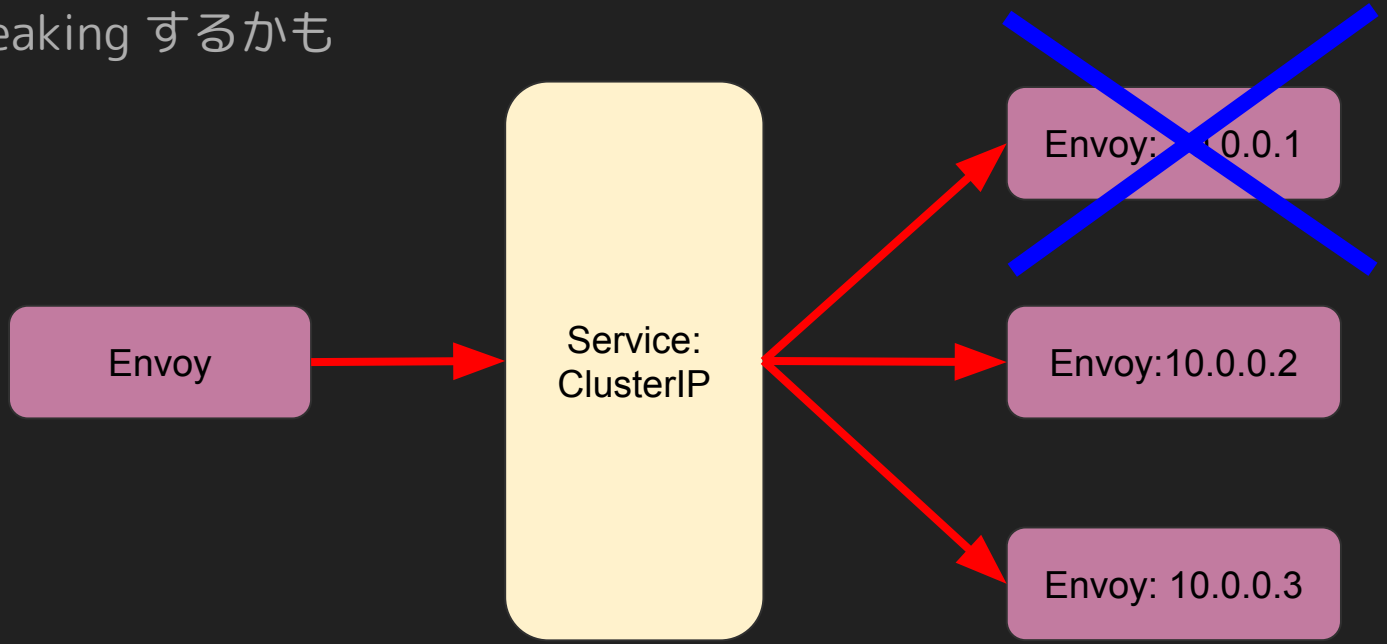
ClusterIP の場合①

- Egress の Envoy は PodIP を直接は知らない（Service が LB する）



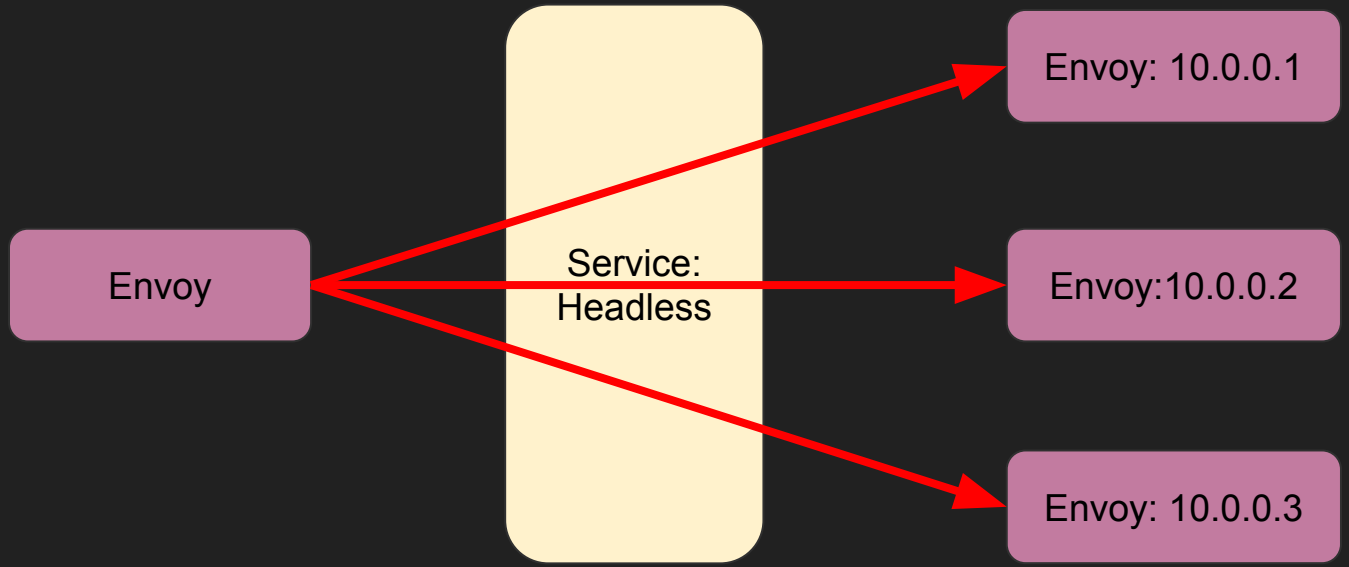
ClusterIP の場合②

- Pod が死んでも Service が unhealthy と判断するまではルーティングされる
- 「偶に」リクエストが失敗する & Egress Envoy が Sevice 自体を Circuit Breaking するかも



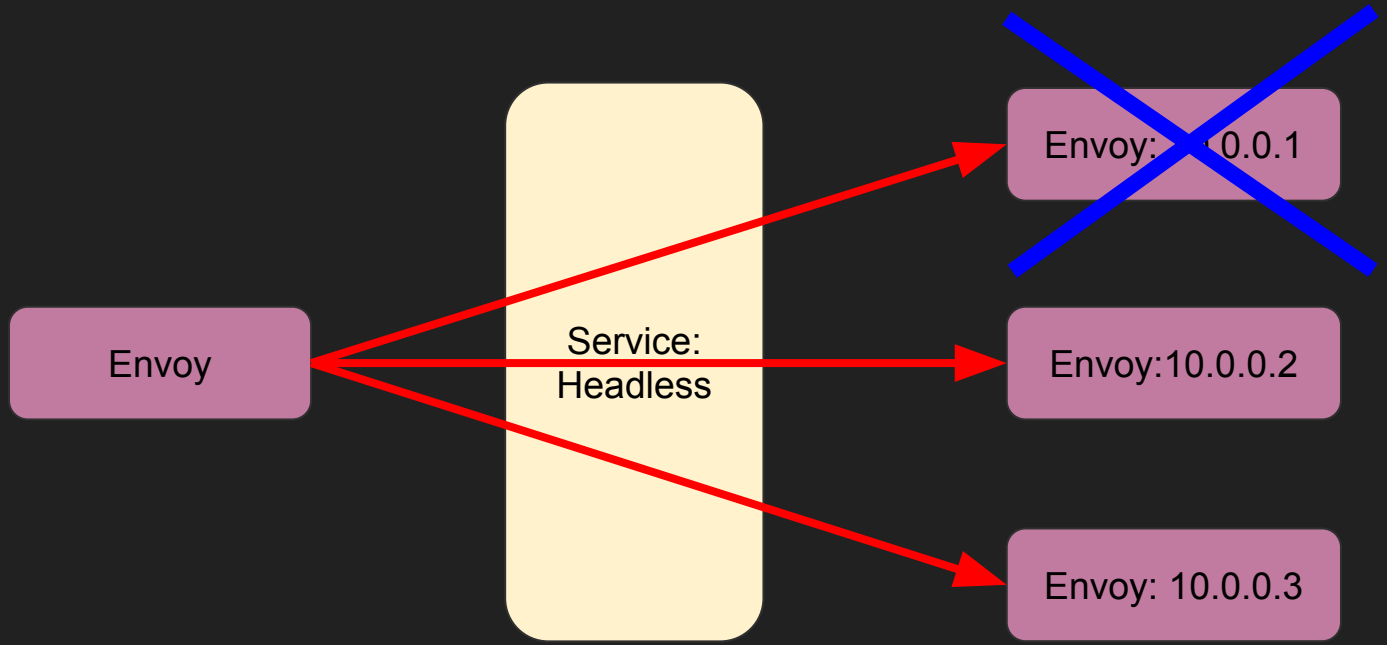
Headless Service の場合①

- Headless Service では直接 Envoy が IP アドレスを知っている



Headless Service の場合②

- Pod が死んでも Envoy 自身が検出して即 Circuit Breaking できる



Service: ClusterIP の罠②

- まとめると、
  - 接続性の問題は LB が2段あることだった
    - LB が Envoy と k8s Service の2段あり、Envoy の方が細かく health check しているものの、 Service の遅い health check 律速でしか配送先の切り替えが起きないため
    - Service を Headless 化することで LB を Envoy に任せることで安定した
- Envoy to Envoy の接続以外でも同じことが起こる
  - 例えば RDS へ Endpoint のドメイン名で接続するとき、実際の IP アドレスを Envoy が掴んだままになったりする（再解決をしてくれない）
  - 結局 Envoy は通さずアプリケーションでコネクション管理している
  - Amazon RDS Proxy にちょっと期待

複数の LB を挟んでしまう場合、必要な相手にいかに**死を素早く伝達**できるか

コンテナ化の恩恵

- プロジェクトが動き出してから、同じ EKS 内に移設することが決まったサービスがいくつかあった
  - Elastic Beanstalk で動作していた Python 製 API
  - オンプレ環境で動作していた Python 製 API
- コンテナ化さえやってしまえばなんとかなる！で実際乗り切れた
  - 新規開発していたプロダクトとは利用しているフレームワークやバージョンの違いなどもあったが、最小限の調整でやりきることができた

プロジェクト開始から4ヶ月、
なんとか1stリリース

本番稼働しないとわからないこともある

Pod へのリソース割当①

HPA(Horizontal Pod Autoscaler) 大暴れ

- 同時実行数の微増減ですぐにスケールアウト/インするピーキーな状態
  - Pod の CPU 割り当てが不足 -> すぐにスケールアウトの閾値を超えてしまっていた
  - Pod の限界性能までは余裕があったので無駄にスケールしていた
    - テスト段階でリソース使用傾向や限界性能をきちんと把握できていなかった
- 最大キャパシティをしっかりテストして測る
  - 十分なCPUを割り当てた上で、性能劣化が起きるより前にスケールするよう調整

Pod へのリソース割当②

- 例）cpu使用量 200m くらいから性能劣化するケース
  - 早めにスケールアウトさせ、上限は余裕をもたせる

```
apiVersion: apps/v1
kind: Deployment
..
          resources:
            limits:
              memory: 128Mi
              cpu: 700m
            requests:
              memory: 64Mi
              cpu: 150m
---
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
spec:
  targetCPUUtilizationPercentage: 80
```

Pod へのリソース割当③

- CPU 不足は気づきにくい
  - CPU 不足は**性能劣化**という形で現れる
  - 性能評価を cpu: limits 設定した状態でやってしまうとベースラインを勘違いする
  - 見ているメトリクスの解像度が足りていないこともある
    - exporter の設定次第だが、15s 程度の解像度だと、もっとショートタイムの CPU バーストが観測できない = 本当はもっと CPU 必要なことに気づけない
- memory: limits をケチるとすぐに OOM Kill を食らう
  - コンテナが死ぬため気づきやすくはある
  - 「あれ、このコンテナ手元では起動したのに・・・」の原因の8割はメモリ不足（経験則）
  - threading や async で処理をしている場合、同時実行数増やすことで同様の問題が起きることも

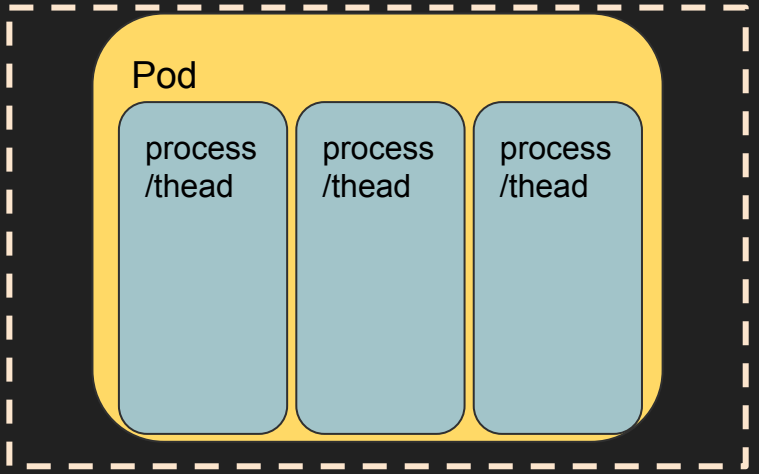
まずは limits 設定せずに性能評価してみること

同時実行数をいかに稼ぐか①

1podシングルスレッド×大量の Pod vs 1Pod マルチプロセス/スレッド

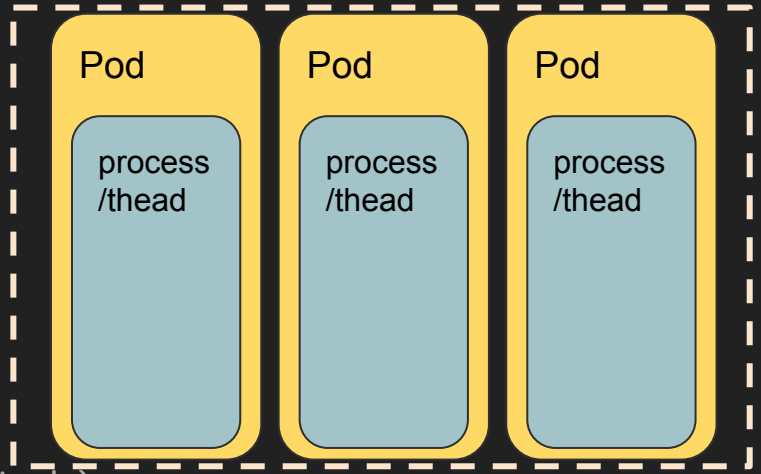
- アプリケーションコード自体を修正して、レイテンシを向上させるのが一番効くのは事実
- k8s のレイヤでできることもある

同時実行数をいかに稼ぐか②



- 起動設定で素朴にマルチプロセス/スレッドにできるフレームワーク/処理系であれば、 1Pod の許容量をあげる手が使える
  - 1Pod に対するリソース割当量は増加するのでコストは増えるかも
  - マルチ化や非同期化はオーバーヘッドで性能劣化する可能性もある
  - コンテナでマルチプロセスすることの良し悪し

同時実行数をいかに稼ぐか③



- Pod自体をスケールさせる（≒プロセスを増やす）
  - Pod が増えるのでコストはかかる、プロビジョニングまでの時差もある
  - デフォルトの CPU 使用率を元にしたスケールだと機能不足な場合も多い
    - リクエスト着弾数やDBコネクション数でスケールさせるには、カスタムメトリクスを利用する必要がある
    - ↑の状況を CPU バウンドに落とし込めているとチューニングしやすい
- とりあえずで Pod 数にものを言わせた解決できるのは k8s の強み

Podは死んでもリクエストは来る①

パターンはいろいろある

- アプリケーションの処理中に Pod Termination になり、 Envoy が先に死亡した結果 Egress 通信ができず処理失敗する
- コンテナ作成後のアプリケーション初期化処理の途中でリクエストが配送され処理失敗する
- Pod の死亡検知が間に合わず ALB がリクエストを配送され GateWay Error

Podは死んでもリクエストは来る②

- 【基本】アプリケーションの状態を Readiness Probe に対応させる
- コンテナ起動/停止順序を制御する
  - コンテナの起動順序は非自明（大事）
  - ライフサイクルフック(postStart/preStop) + Volume Mount を活用
    - 起動/停止完了するまで Sleep させる
    - 秒数指定する場合 Sleep は 5-10s くらいが無難、preStop フックの猶予時間は 30s
    - Volume Mount を使ってコンテナ間で状態を共有する
  - Istio 使ってもできるわけではなさそう
  - k8s v1.18 でサポートされそう
    - https://github.com/kubernetes/enhancements/issues/753
    - https://banzaicloud.com/blog/k8s-sidecars/

ALB の反応が遅い①

- NodePort ではなく HeadlessService で SVC を構成していたため、 ALB が PodIP を持っている
  - 「ClusterIP の罠」を回避するため
- Pod が死んだときの検知が間に合わない
  - ALB のヘルスチェック間隔は 最小 5s
  - deregistration_delay（登録解除までの待ち時間）の罠もある
- Gateway Error が起きたときにリトライする機能が ALB にはない

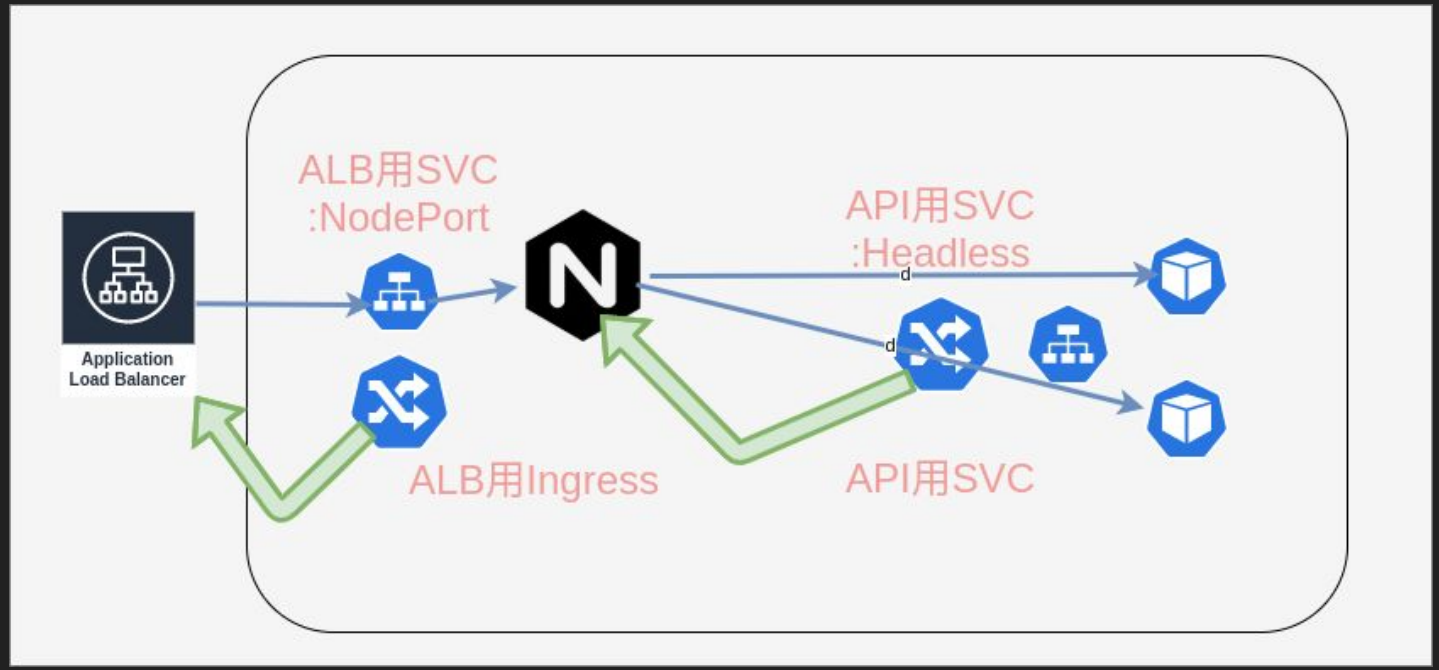
ALB の反応が遅い②

ダブルIngressモデル

- ALB -> AwesomeIngress -> マイクロサービス用 ingress の2段構成にする
  - ALB： internet facing な TLS 終端として利用
  - リソースルーティングは間に仕込んだ Ingress（LB） にやらせる
    - Pod のより詳細な死活監視やリトライ機構の活用を期待
  - 選択肢としては Nginx Ingress Controller か Envoy Ambassador あたり
  - Nginx を採用することにした（現在試験中）

Nginx Ingress Controller②

構成イメージ



Nginx Ingress Contoller③

- ドキュメント・実例が充実していることからこちらを採用
- upstream 側の Pod 検知・解除は k8s の Endpoint API を利用しており高速
- Gateway エラーに対するリトライ機構も完備
- クロスネームスペースのルーティングも簡単
  - ネームスペース毎に Nginx を準備して冗長化（推奨）
- Nginx に対する SVC は NodePort にして ALB に公開
- ALB と Nginx 併用する場合は それぞれに大して Ingress 定義する
  - external-dns 利用している場合、 host ベースでのルーティングを複数書いてしまうと競合してしまうので注意

ログとメトリクス、多すぎ

- k8s はコンポーネントが多いのでログも莫大
  - 現行本番で 4GB/day
  - Splunk は ログ取得量/day でのライセンスなのでログ量を収める必要があった
  - FIrehose + Lambda で正常応答系ログの一部をフィルタリング
    - fluent-bit でもフィルタ可能
- メトリクスも膨大
  - Prometheus のキャパシティ不足
  - いったんはスケールアップで対応した
  - 分散構成という手はあるが・・・

インフラのバージョンアップ

運用フェーズに入って落ち着いたことでようやく検討することができた

- このときのバージョンは 1.13.7（1.13 系のファーストバージョンだった

AMI と EKS バージョンの頻繁な更新に追従する

- AMI
  - セキュリティパッチ等を含む場合も多いため見逃せない
- EKS プラットフォームバージョン ≒ k8s バージョン
  - EKS バージョン は k8s バージョン に追従する形で上がっていく
    - およそ3カ月毎の k8s マイナーバージョンのアップデートは見逃せない
  - Managed Node Groups や Fargate が利用できるのは k8s v1.14 相当のプラットフォームバージョンから

AMI のバージョンアップ

ワーカーノードのインスタンス入れ替え操作を行う

- in-place な方法
  - ノードグループ内での Rolling Update
- out-of-place な方法
  - 別系を新しい AMI で準備してからトラフィックを切り替える Blue-Green Deployment
- 選択基準は Rolling Update でのダウンタイムを許容できるか
  - バージョン更新による影響やバージョンが混在することによる影響は事前検証可能なため、Rolling Update によるサービス影響が許容できるなら、そちらが楽

EKS のバージョンアップ

そもそも EKS のバージョンアップとは

- ≒ k8s マスターノードのアップデート
  - 一部 kube-system 系のコンポーネントの更新も作業としては必要

ワーカーノードの更新手順は AMI のときと同様

- マスターノードの方が新しい限りはバージョン互換性はサポートされているため、ワーカーノードへの反映は任意のタイミングで良い
- in-place か out-of-place か選択する

インフラのバージョンアップ②

in-place の更新を選択した

- 現状のシステム利用状況であれば アクセス頻度の少ない時間帯（深夜）であれば問題ないと判断
- 「攻めた」やり方試せるのはこれが最後の機会かも、という打算もあった

結果的に問題は全くなかった

- 公式のドキュメントをよく読み、試験リハーサルきっちり行い、過不足なく作業するのが大事
- dejima を利用する別システムの担当者（社内）からは「モニタリングしてもいつメンテナンスあったのか判断つかない」という嬉しい声もあった

いろいろあったが、
プロダクションでなんとか運用中

今の課題

可観測性

- 現状の課題
  - ログを一部削ってしまっている
  - メトリクスは Prometheus が不安定
  - トレーシングも取り入れたい
- メトリクス・ログ・トレーシングを三位一体で活用できるような監視系全体の再設計が必要
  - 自前での監視系運用を続けていくべきか？というを合わせて大きな悩み

Istio

- 最初期に比べるとマイクロサービスも増加し、サービスメッシュを動的に構成しないとつらい
  - 例えば今は全てのマイクロサービスで同一の envoy コンフィグを使っているが、サービスによって変更したい（Service discovery, Load Balancing）
- Canary Release を導入したい
  - Istio の Traffic Management は有力な選択肢
- Istio 向けに全体的に構成作り直すコストをいつ払えるか？

クラスターマネジメント①

- ノードのAMIやEKSバージョンの管理
  - どちらも高頻度なのでオペレーションを確立しておく必要がある
- 稼働中 Pod の激増に伴い、ノードが不足する未来が見えつつある
  - 3AZ × 最大 6 Nodes = 最大 18 Nodes となる Auto Scaling Group でクラスタを構成
  - Cluster Auto Scaler はスケールイン時にインスタンス配置が AZ 非対称になる可能性がある
    - im-running-cluster-with-nodes-in-multiple-zones-for-ha-purposes-is-that-supported-by-cluster-autoscaler
    - 単純に Node 数の上限値を増やす解決はしたくない
    - ノードのスケールアップを適宜行い、ノードの絶対数が増えすぎないように調整することも必要

クラスターマネジメント②

- Managed Node Group
  - スケールするクラスタを簡単に構築できる
  - ちゃんと CFn かいてクラスタを組んでいたので恩恵は少なめ
  - Cluster Auto Scaler のスケールイン時の問題は残る
- Fargate
  - 導入には Fargate 向けに根本的に構成見直す必要がある
    - CNI や PersistentVolume など（まだ）機能的に不十分なこともある
    - DaemonSet 使えなくなるのは痛い（監視系コンポーネントなど Sidecar 化する必要）
  - EC2 と Fargate 併用せざるを得ない部分はあるため、「on EC2 向けのもの」「on Fargate 向けのもの」を別々に管理・運用・教育していくことのコスト

チーム体制と教育①

- EKS できるエンジニアは社内でもほんのひと握り
  - どちらかといえば自習によって追いついてきている人々
- AWS も k8s もしっかり理解しないといけない
  - デプロイするだけなら k8s だけ勉強しとけばいいかもしれないが・・・

チーム体制と教育②

- まずは触るところを限定する
  - エンジニアの責務範囲を限定する
    - アプリケーションエンジニアコンテナ化までを考えればよい環境にしたい
      - 今日話したようなことまで完全に理解して手を動かせる、という必要はない
  - 結局 AWS や k8s のレイヤーまで理解しないと Production Ready なサービスにするのは難しい。完全に分離してしまうのもどうか？という個人的な思いはある
- 一歩目を踏み出しやすくする
  - EKS/k8s の魅力を発信する
    - メリットやノウハウを共有して「使ってみたい！」と思ってもらう
  - 簡単に触れる環境づくり
    - 一番の障壁は環境を作ること
    - playground 的な環境を準備しておく
    - マニフェストのテンプレートを配布する
    - CI/CD

今後

もっと Toil Free な世界を目指して

サービス価値提供のためのより良いプラットフォームでありたい

- 高可用性を突き詰める
  - ユーザにとっての価値 + 開発者を運用 Toil から守る
    - サービスが成長するほど新たな課題が現れるので、地道に一つ一つ解決する
- 監視機構の充実
  - 運用 Toil に陥ってもすばやく「サービス開発」に復帰できる体制づくり
    - メタリング、ロギング、トレーシング 三位一体の監視機構
- 一日一デプロイを目指して
  - 価値あるサービスをすばやく届けるのがエンジニアの本懐
    - CI/CD や 自動テストをより充実させ、安心して高速にリリースできる体制に

主力システムのクラウドシフト

オンプレミスで稼働中のコアプラットフォームの移行が決まっている

- dejima はこのための壮大な PoC だった
  - これまでの知見をもとにスムーズかつできる限りモダナイゼーションするのが dejima チームに課せられた使命

まとめ（所感）

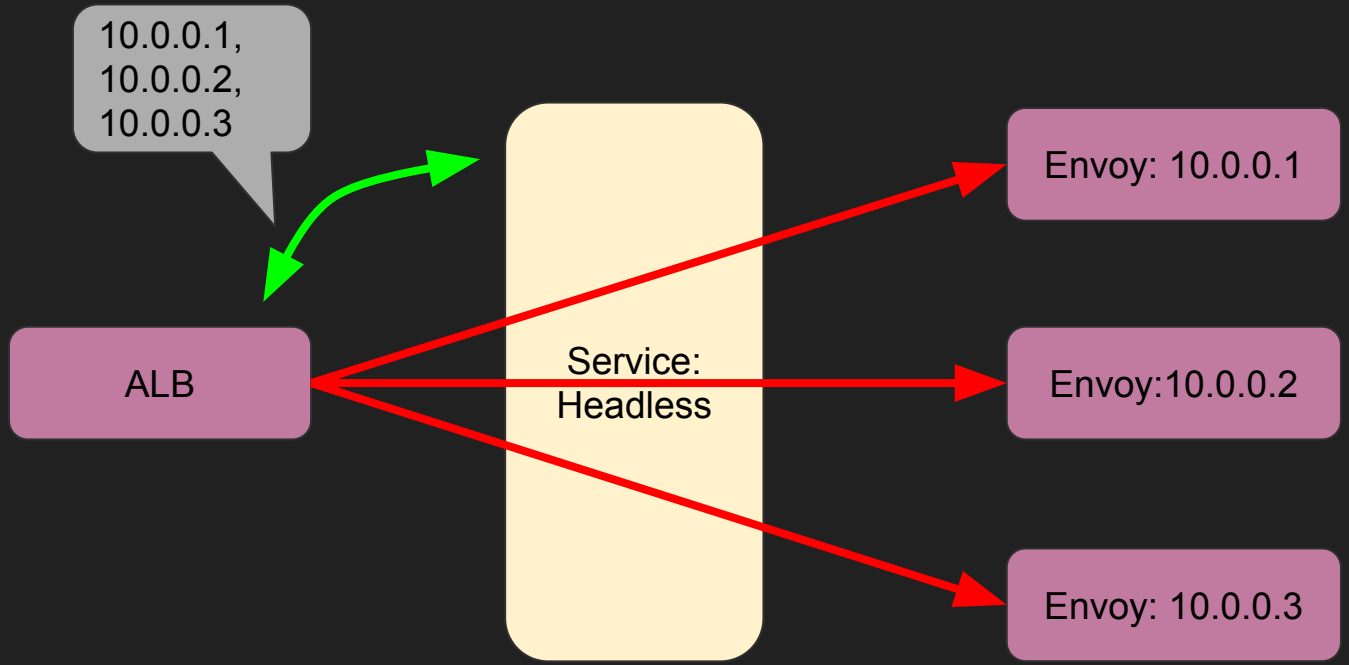
- コンテナ化すればなんとかなる世界は幸せ
  - 間違いなく開発サイクルは高速化していると感じる
- プロダクション運用してみないとわからないつらみもたくさんある
  - マイクロサービス化含めてコンポーネントを細かく分割したことによる数の暴力に泣きがち
  - 今の課題感の多くは、EKS にしたことによる苦労よりも、より良いアーキテクチャや可観測性を目指した結果の苦労なので、辛くも楽しい
- ただのアプリケーションエンジニアだったころに比べて圧倒的に多くの経験値を得られた

エンジニア積極採用中です

Appendix

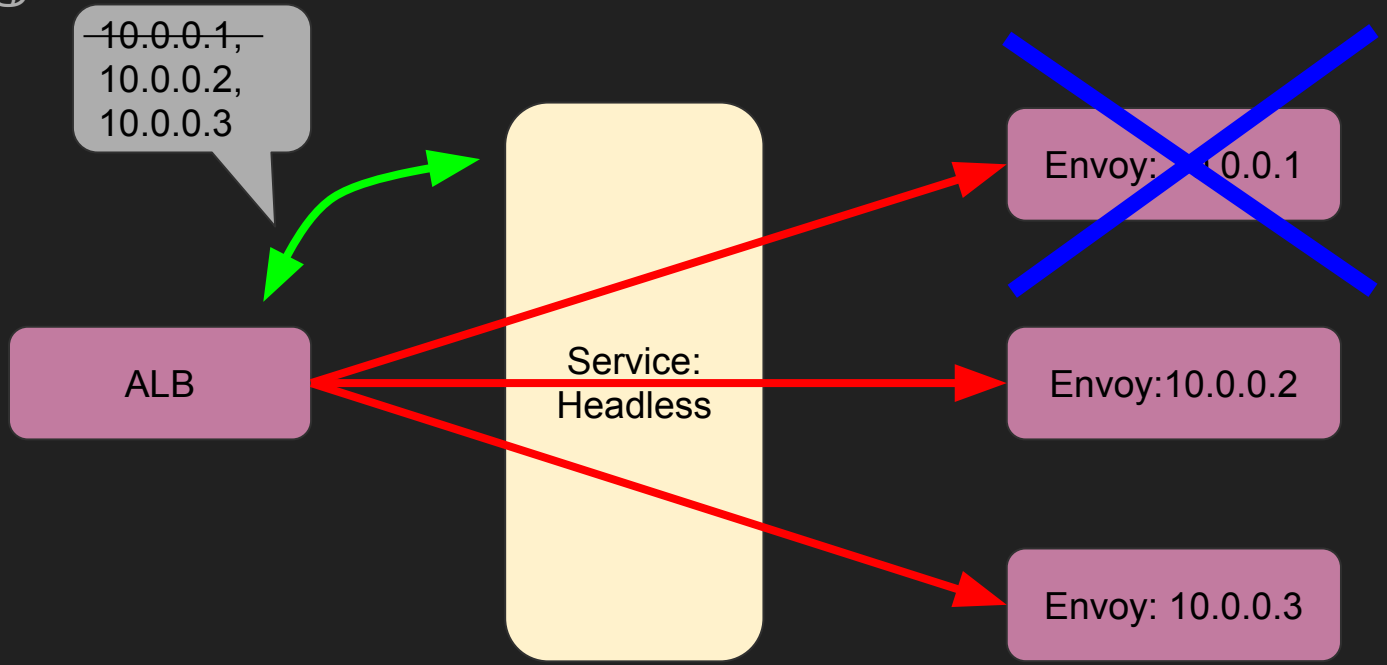
ALB の Pod 死亡検知遅れ①

- ALB は Pod の IP を直接保持し、自身でヘルスチェックする



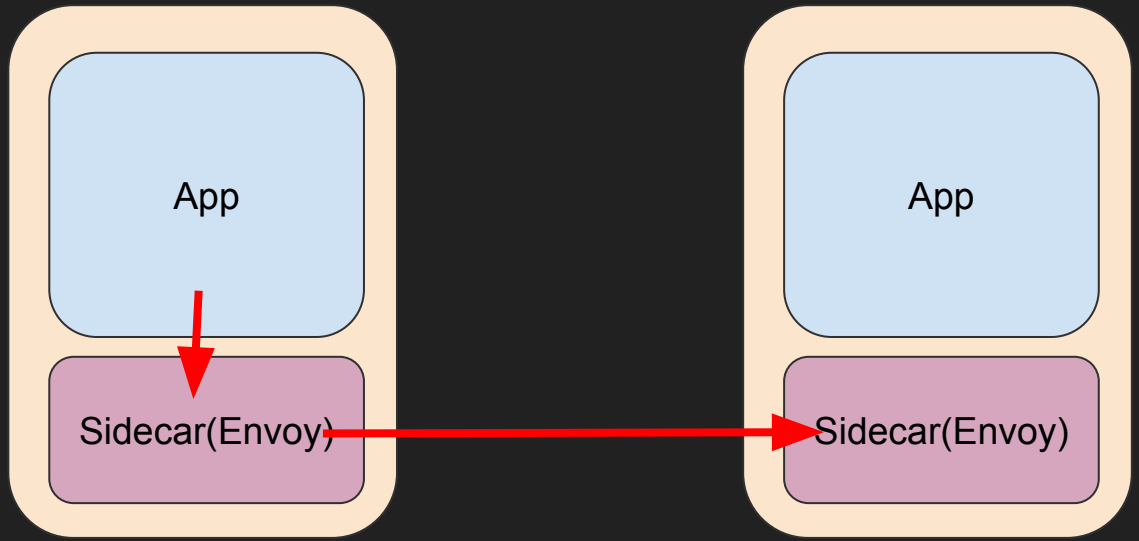
ALB の Pod 死亡検知遅れ②

- Pod 死亡時に ALB のヘルスチェック=検知が間に合わずリクエストが配送される



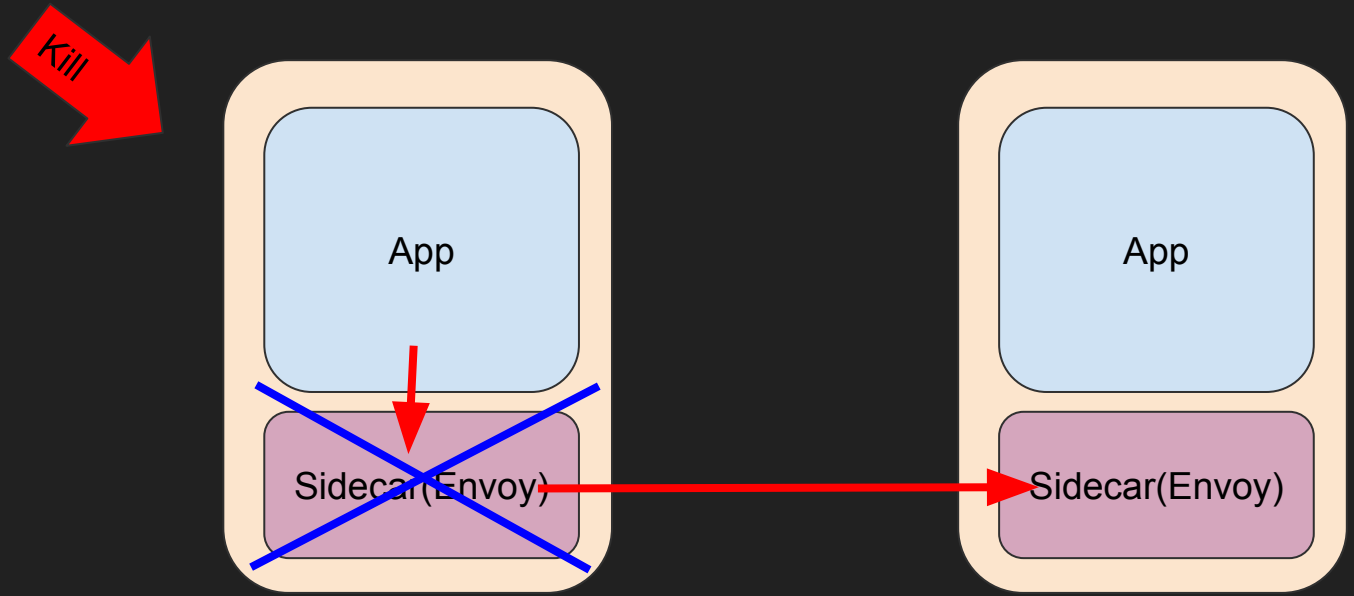
Sidecar が先に死んで通信不能になる①

- Sidecar を通した Egress の通信をする



Sidecar が先に死んで通信不能になる①

- Pod 死亡時に Sidecar が先に死ぬと、外部通信が必要な App では処理中だった場合リカバリできない



ログの奔流①

- aws-for-fluent-bit で全ログ収集していた
- k8s はとにかくログが多い
  - 現行本番環境で 4GB/day くらい
  - アプリケーションログに加えて envoy のログ
  - 監視系 agent や k8s-system 系のログも馬鹿にならない
- 一日のログ取得可能量に制限があった
  - Splunk は日毎のログ量=ライセンス のため
  - Splunk でなくてもログ取り込み量はコストに直結する部分ではあるはず

ログの奔流②

- 本当に必要なログ以外はカットした
  - 5xx 応答やアプリケーションログが見れない状態はクリティカルにまずい
  - 一部の正常応答アクセスログなどをフィルタして落とす
- Firehose を利用していたため、 Lambda を挟んでログをフィルタリング
  - Chalice(Python) でさっと作ってデプロイ
    - 秒速で Lambda 関数作りたいときに Chalice は強い
    - 今のところ困っていないが Go で書き直したほうが性能は出る & バイナリにできるので取り回しはしやすいのかも
  - Lambda でやることで 仕組み自体は fluent-bit 経由以外のログにも適用でき汎用的に使える
- fluent-bit の plugin でフィルタ
  - k8sの中で完結させたいならこちらがおすすめ
- フィルタすることで落ちた可観測性は課題

不安定な Prometheus

- Prometheus から応答がない
  - マイクロサービスの種類、 Pod 数ともに増大しておりメトリクスの絶対量も膨大に
  - 特に Node がスケールしたタイミングは一気にメトリクスが増えるので不安定な傾向
  - これを1台の Prometheus で処理することに限界があった
    - 稼働中はよくても、なにかの拍子に Prometheus の Pod が死ぬと、起動時にはメモリ不足でいつまで経っても起動しなくなってしまう、という問題にも悩まされた
- 一旦はワーカノードのスケールアップで対応
  - 常にリソースが必要というよりは瞬間的なものなので t3 系がおすすめ
- Prometheus の場合、メトリクス収集対象分割という手段はある
  - Prometheus 自体を分散させることで負荷を軽減させる作戦
  - 設定はそれなりに複雑で作り込む必要がある
  - Prometheus 自前運用していくコストを払えるか？