顧客のアプリケーションコードが動く マルチテナント環境における課題とEKSにたどり着くまで

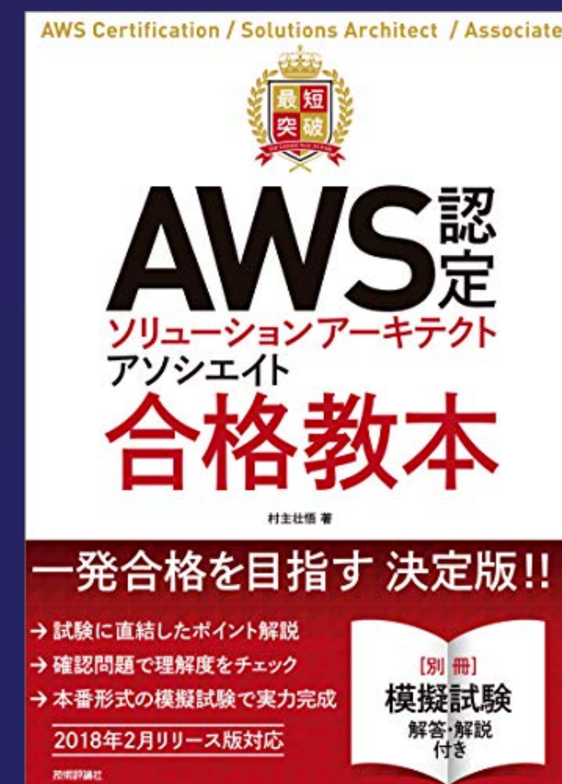
春のAWS コンテナ祭り with Amazon EKS

ABEJA, Inc.
Shogo Muranushi

Shogo Muranushi

ABEJA, Inc.

- Site Reliability Engineer Tech Lead



アジェンダ

- 話の背景となる事業紹介
- アーキテクチャの進化とマルチテナントの工夫点

ABEJA

2018.12.06 プレスリリース

「ABEJA Platform」、国内で初となる 「AWS Machine Learning コンピテンシー」パートナーに認定

ディープラーニングを活用したAIの社会実装事業を展開する株式会社ABEJA(本社：東京都港区、代表取締役社長：岡田陽介、以下 ABEJA（アベジャ）)は、アマゾン ウェブサービス（以下、AWS）の「AWS Machine Learning コンピテンシー」パートナーに、2018年11月に認定されましたので、お知らせいたします。なお、本コンピテンシーの”プラットフォームソリューション”カテゴリーにおいて、ABEJAは、グローバルでは8社目、国内では初となる認定組織となりました。

「AWS コンピテンシー」とは、習熟した技術を持ち、専門的なソリューションエリアでAWSの顧客を成功に導いた実績を持つAWS パートナーネットワーク (以下、APN)パートナーに付与される資格です。また、「AWS Machine Learning コンピテンシー」とは、新たに公開された機械学習領域の卓越した技術と導入実績を持つパートナー向けのAWSコンピテンシーであり、この度の認定においては、ABEJAの提供する「ABEJA Platform」が、”プラットフォームソリューション”カテゴリーで評価を受け、国内初となるパートナーとして認定されました。

大量データの取得に必要なAPIや負荷分散の仕組みや準備、セキュリティ担保

データ、モデル、結果のバージョン管理

冗長性やGPUリソースの担保、エッジ側との連携プロセス構築

教師データの作成に必要なツールと人材の準備

データ取得 → データ蓄積 → データ確認 → 教師データ作成 → モデル設計 → 学習 → 評価 → デプロイ → 推論 → 再学習

データウェアハウスの準備と管理

0からのモデル設計

開発環境から本番環境への引き渡し

データのバリデーション（正確性）の確認

GPU環境の準備と高度な分散化

統計的に本番にデプロイした瞬間から精度が下がることを担保

AI活用までに数多くの課題が存在

大量データの取得に必要なAPIや負荷分散の仕組みや準備、セキュリティ担保

データ、モデル、結果のバージョン管理

教師データの作成に必要なツールと人材の準備

冗長性やGPUリソースの担保、エッジ側との連携プロセス構築

データ取得 → データ蓄積 → データ確認 → 教師データ作成 → モデル設計 → 学習 → 評価 → デプロイ → 推論 → 再学習

データウェアハウスの準備と管理

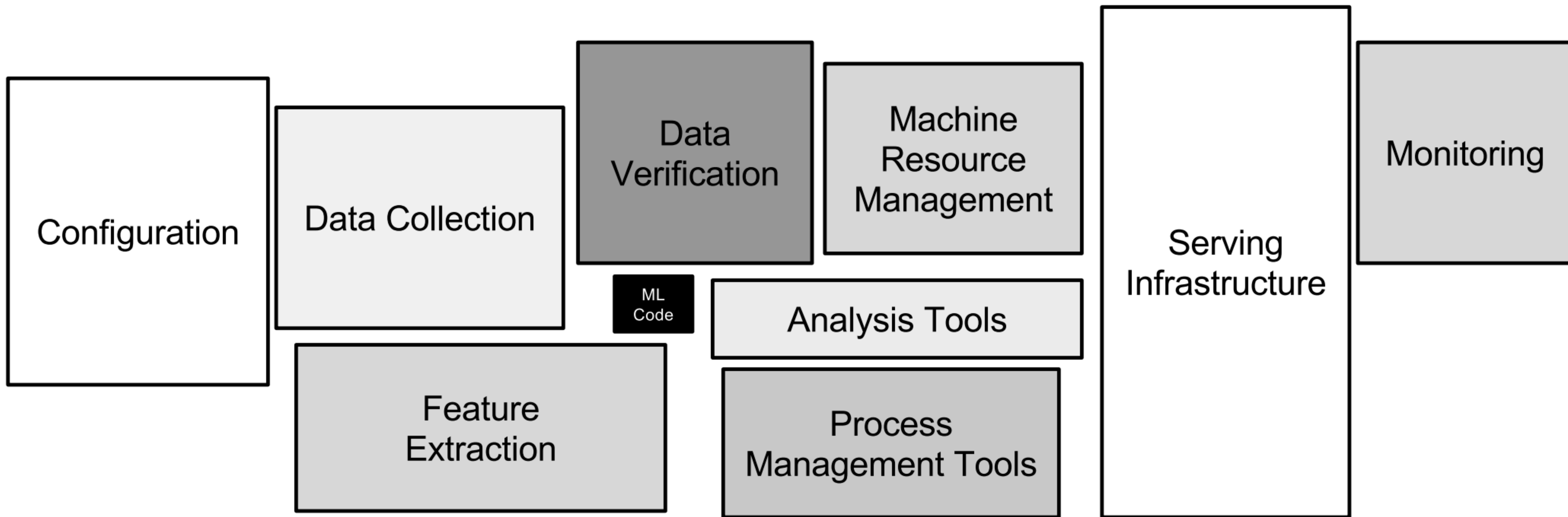
0からのモデル設計

開発環境から本番環境への引き渡し

データのバリデーション（正確性）の確認

GPU環境の準備と高度な分散化

統計的に本番にデプロイした瞬間から精度が下がることを担保



Ref: https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

“ As the machine learning (ML) community continues to accumulate years of experience with live systems ”

“ 開発およびMLシステムを導入することは比較的高速で安価ですが、時間をかけてそれを維持することは困難かつ高価である”

ABEJA
PLATFORM

アーキテクチャ

アーキテクチャ

現状はEKSベースでしたが、開発初期はECSをベースで
色々工夫しながら今の構成にたどり着きました

アーキテクチャ

そこに至るまでのアーキテクチャの進化の過程で
苦労した点・工夫した点と、

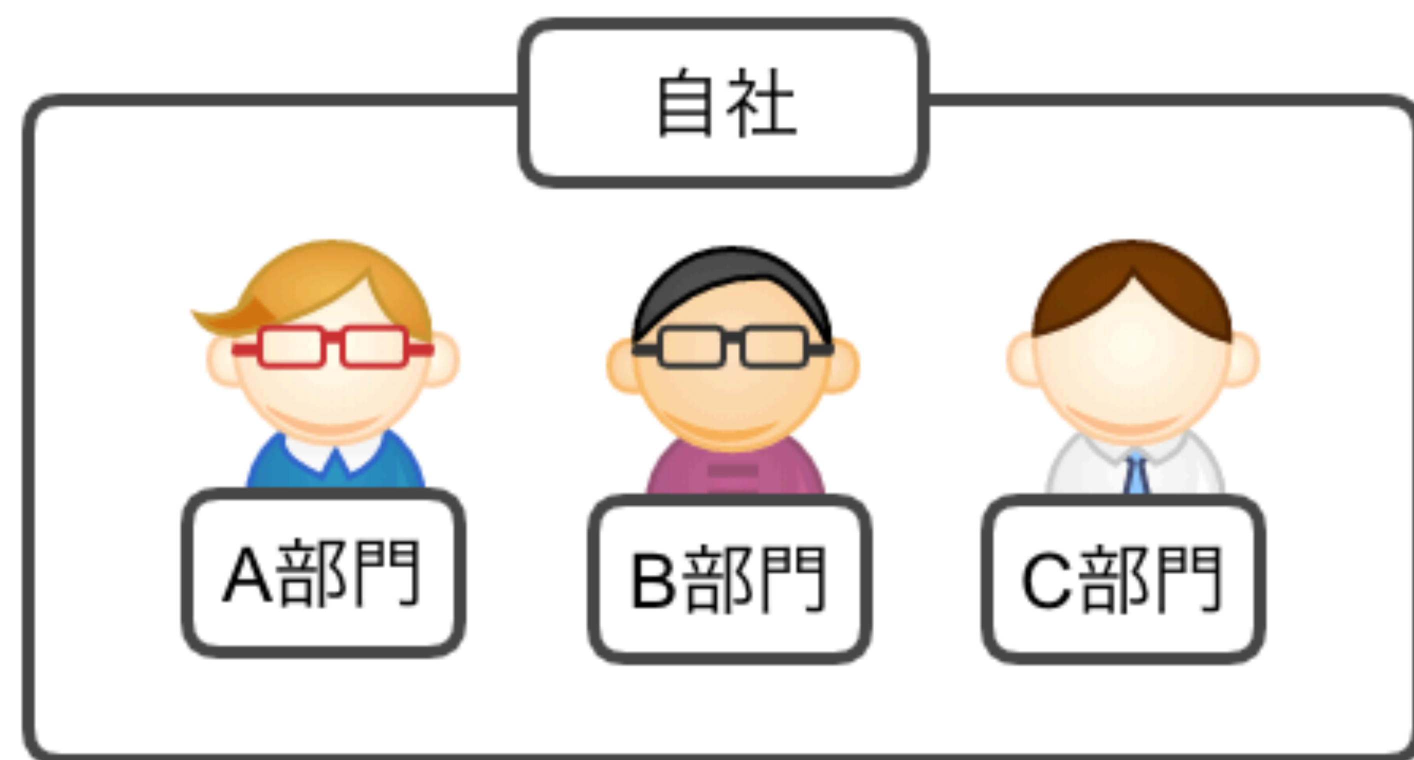
「顧客のコードが動く」マルチテナント（？）環境下で
Kubernetesを用いてプラットフォームを構築する際の
苦労した点・工夫した点を話します

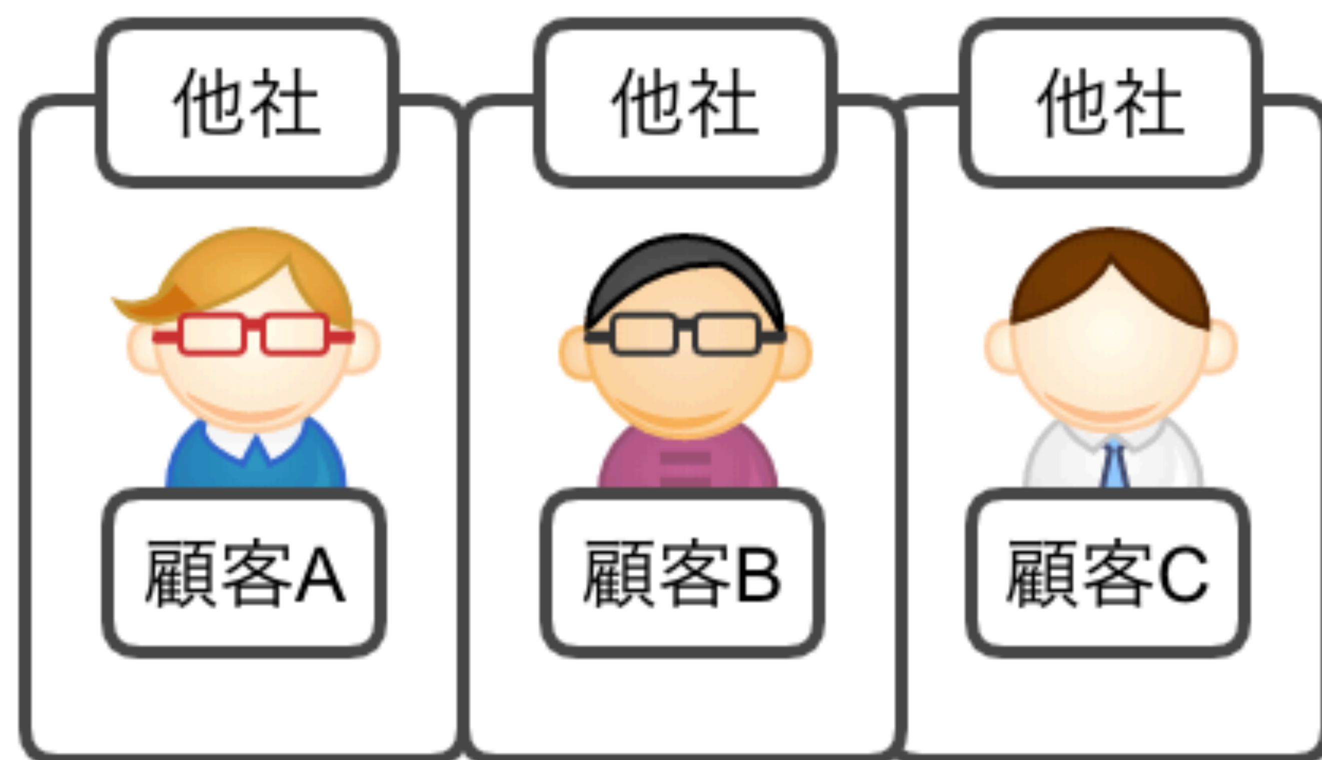
【前提】マルチテナントとは？

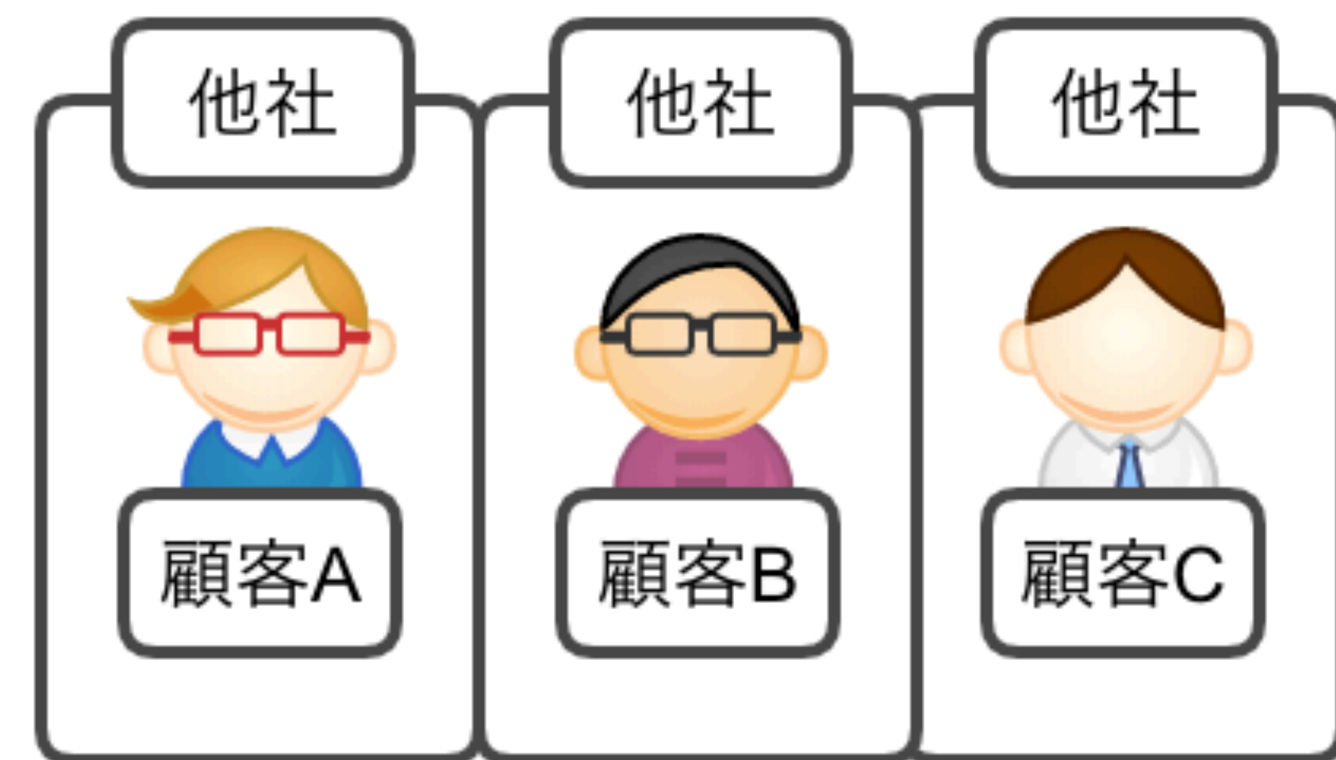
自社部門でマルチテナント
アプリは自社

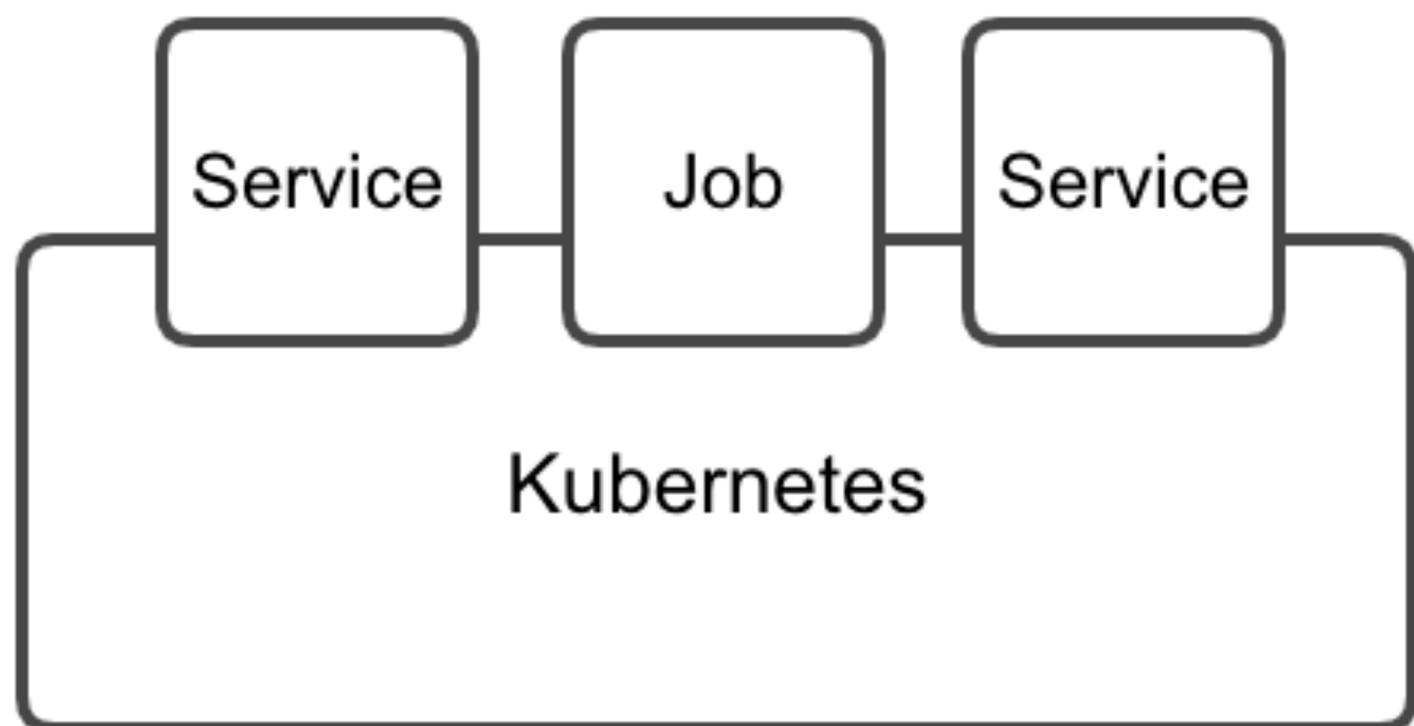
顧客毎に論理DBでマルチテナント
アプリは自社

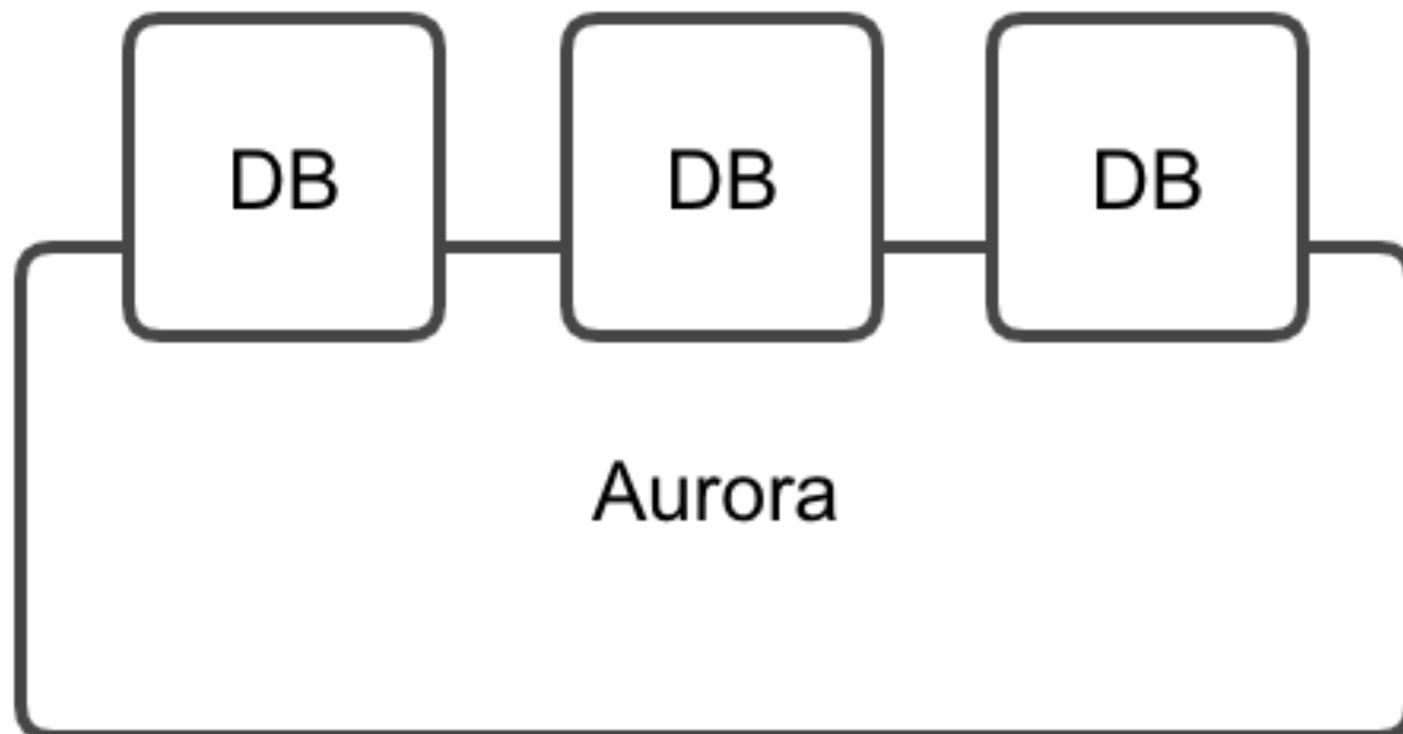
顧客のアプリケーションコードが
動くマルチテナント
アプリは顧客

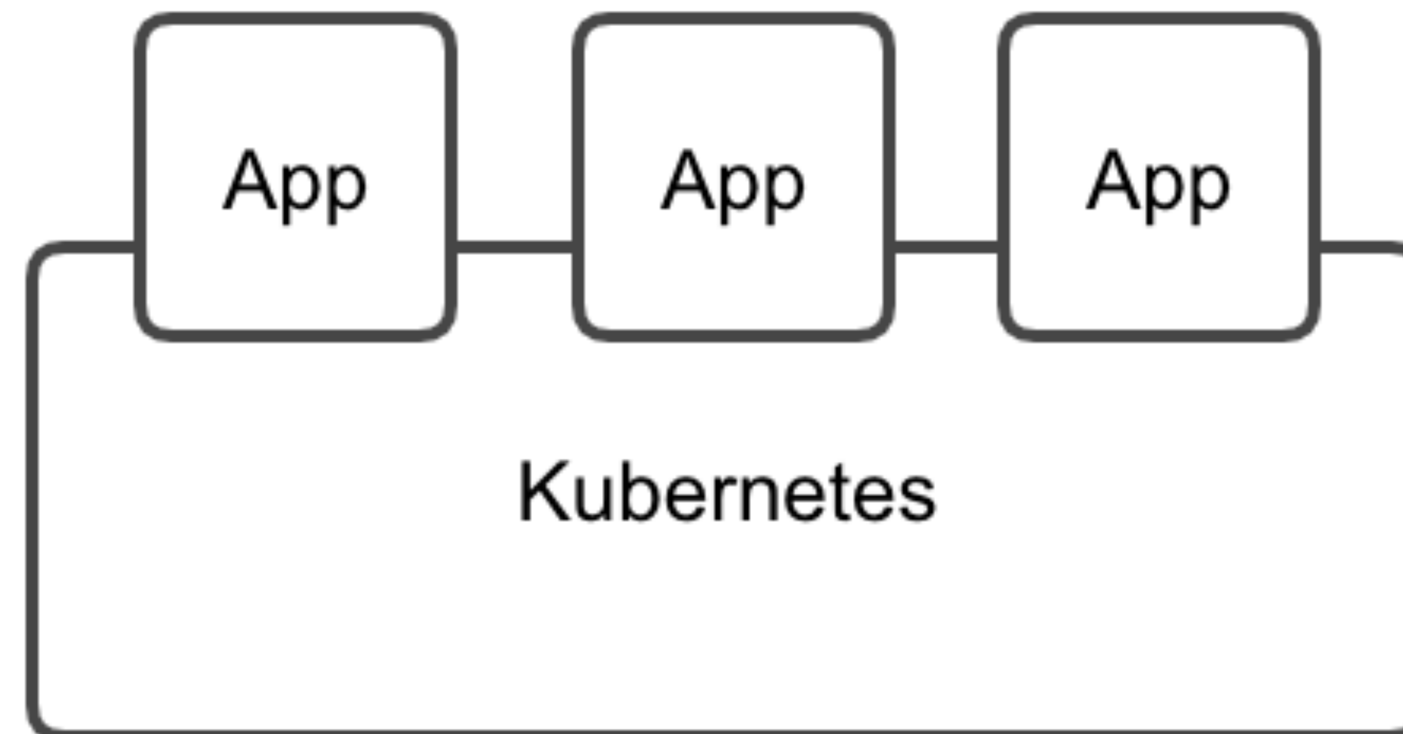




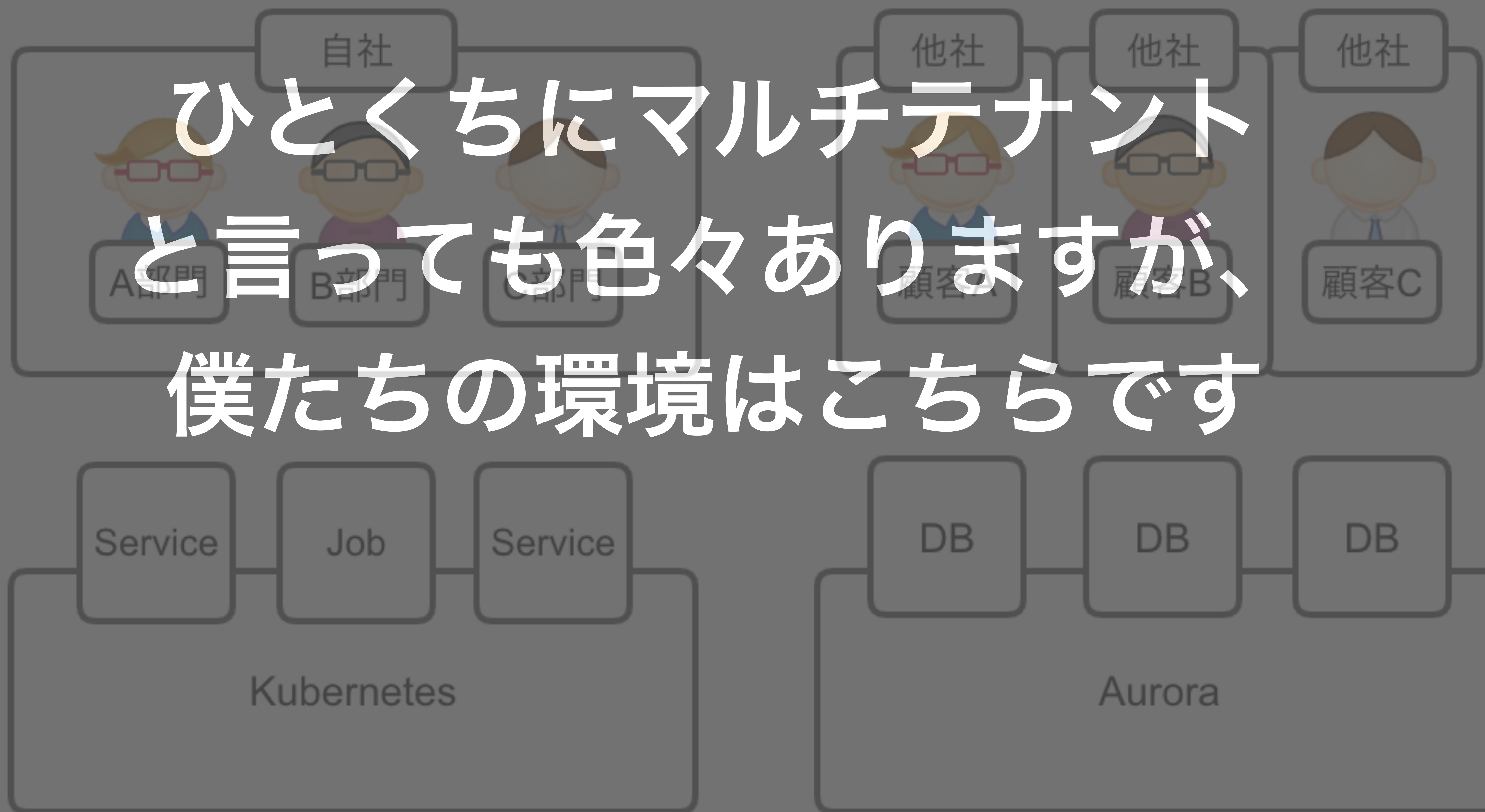


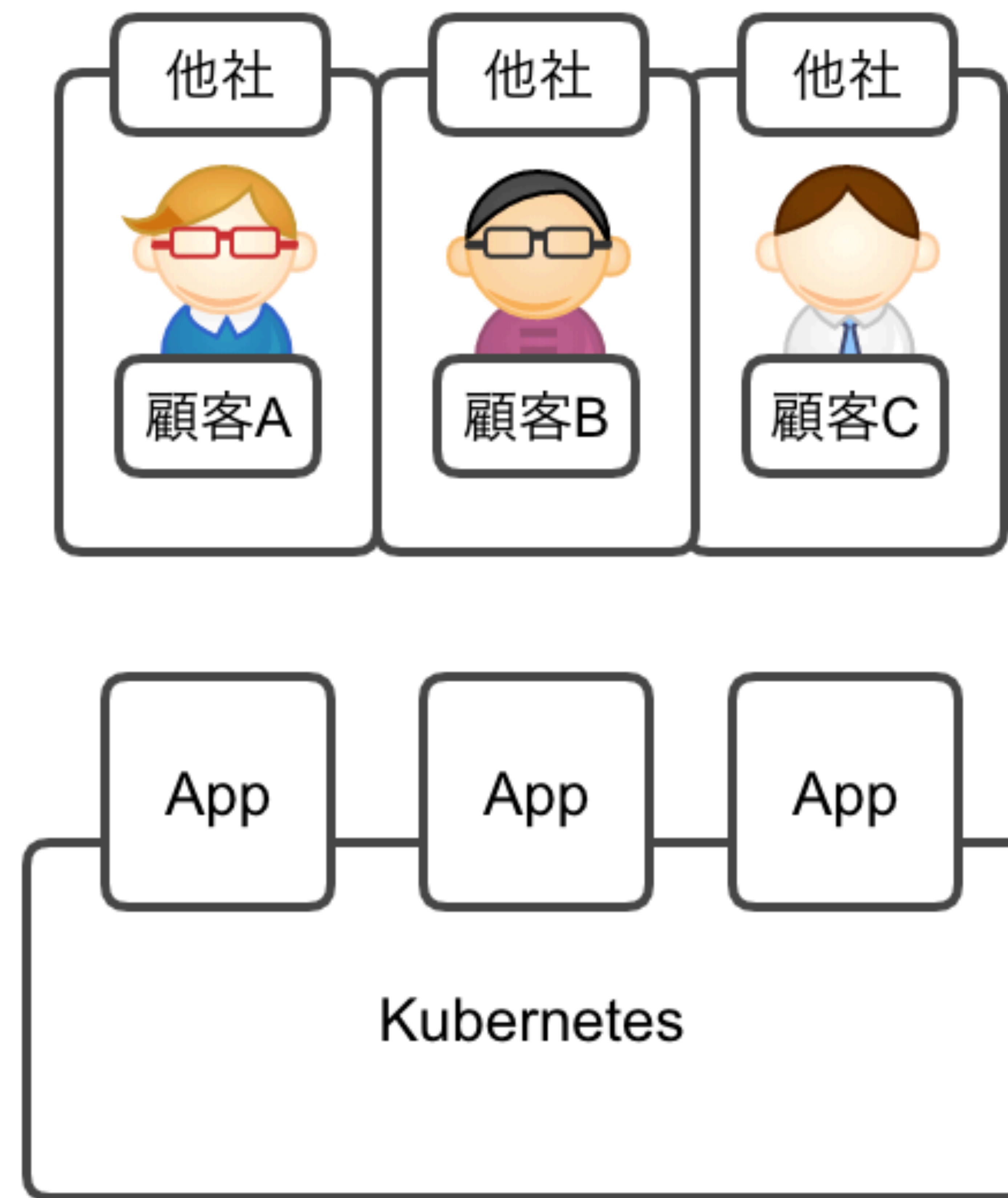






ひとくちにマルチテナント
と言っても色々ありますが、
僕たちの環境はこちらです





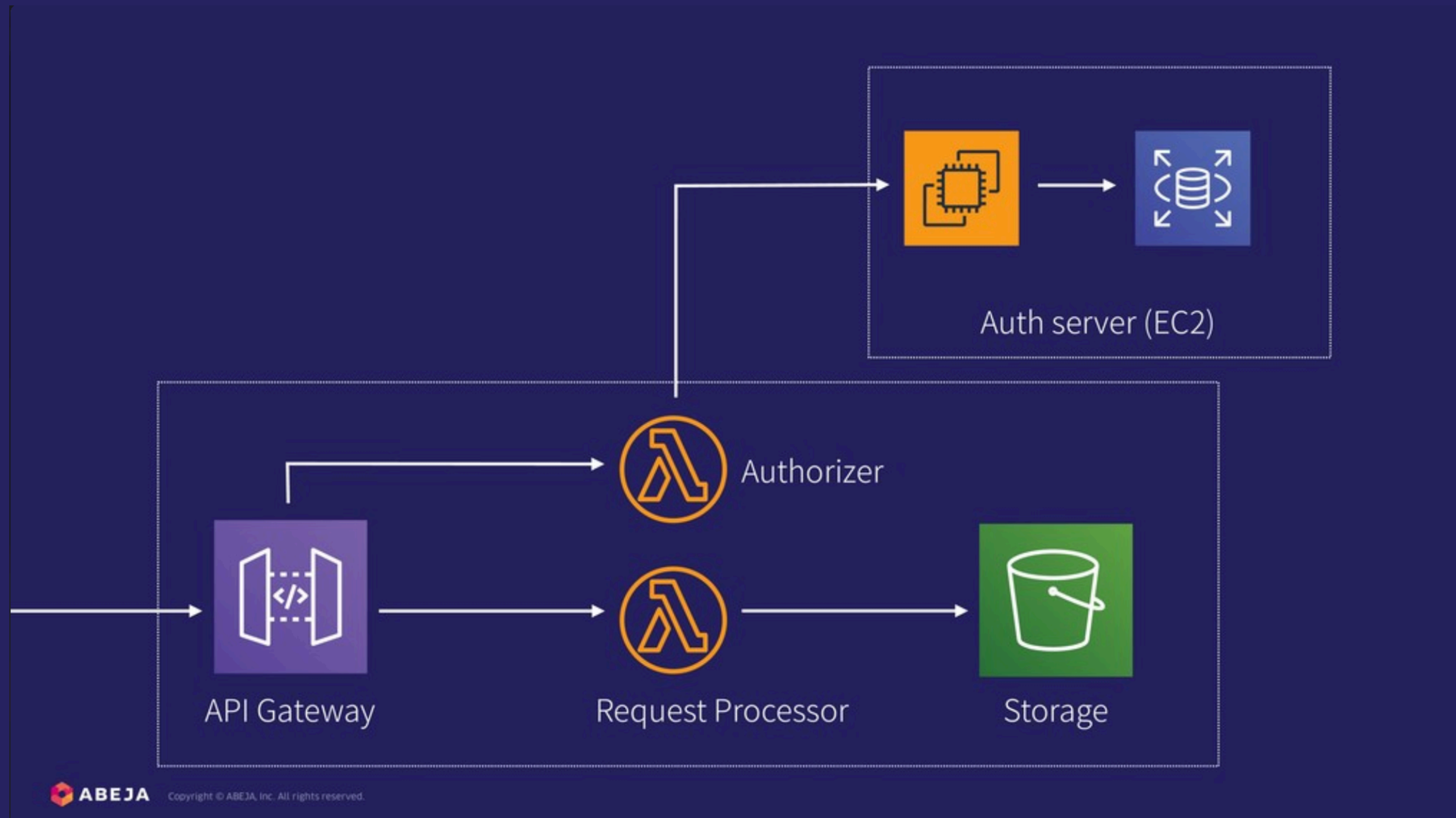
アーキテクチャ

Datalake

第一世代アーキテクチャ of Datalake

- Raw データを蓄積するためのサービス
- ストレージは S3
- 初期バージョンは API Gateway と Lambda で REST API で提供
- 当初はファイルの保存、取得、一覧などのオペレーションのみ
- 構築にはServerless Frameworkを利用
- Signed URLを発行しアップロードしてもらう仕様

第一世代アーキテクチャ of Datalake



Good

- メンテナンスフリー

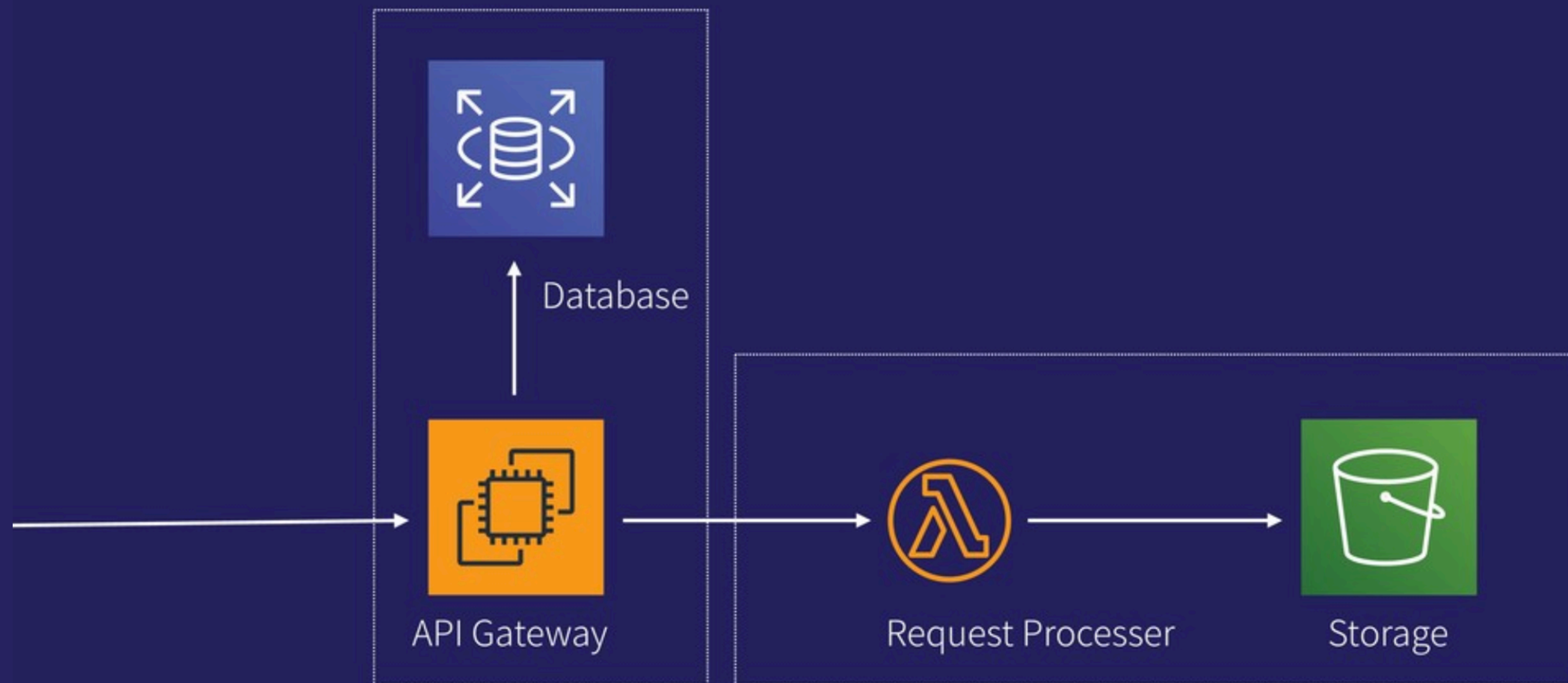
Bad

- API Gatewayつらい
  - ローカルの再現性が低く開発効率悪い
  - ペイロードサイズ
- Serverless Frameworkまぁまぁ辛い
- 今後他のサービスも同じスタイルで開発？

第二世代アーキテクチャ of Datalake

- AWSのAPI Gatewayはシンドいので、API Gatewayを内製
  - 開発効率が悪い、大事なエンドポイントなので障害時のコントロールはしたかった
- API Gatewayが分離されたことで、認証・ルーティングが共通に
  - 沢山のAPI Gatewayとはおさらば
- DatalakeはLambdaとS3のみに
- 当面はこの構成

第二世代アーキテクチャ of Datalake



Good

- 認証・ルーティングを分離し共通に
- Datalakeの責任範囲を分離

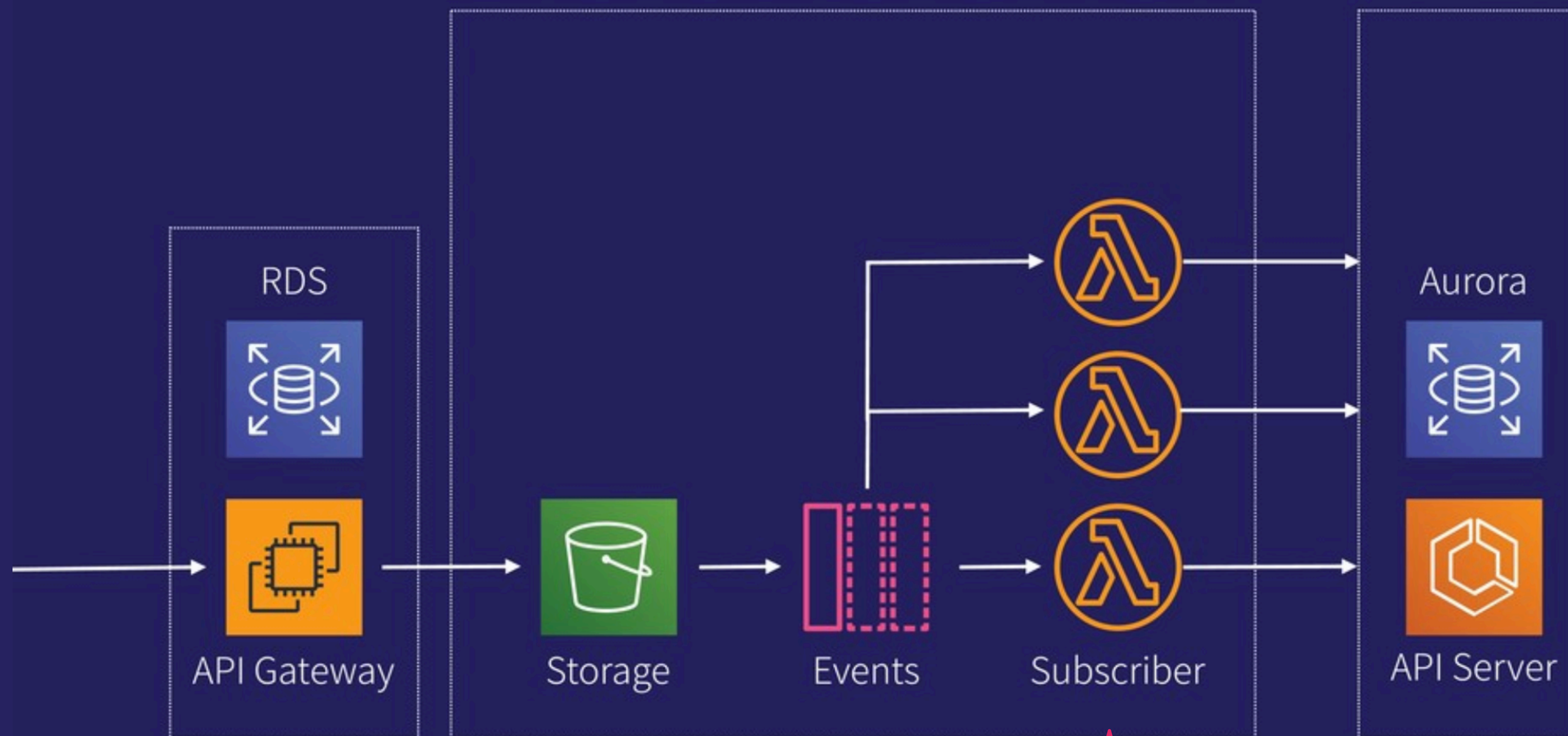
Bad

- 自前API Gatewayの面倒を見る必要あり

第三世代アーキテクチャ of Datalake

- 「検索やカウントがしたい」
  - メタデータ検索やカウント機能用のDBを構築
  - 検索はPostgreSQL(Aurora)のJSON+GIN Indexで実装
    - 柔軟なメタデータの付与と検索をできるようにし過ぎてIndex爆発
    - Cassandra…? Or 柔軟性を軽減させることを検討中
  - S3 Event + SQS + Lambda から Datalake API を呼び出す
    - バックエンドの負荷が耐えきれなかった、CW Logsが高くなったのでバックエンドをLambdaからECSに移行
- 「S3のSigned URLは手間なので一回でDatalakeにアップロードしたい」
  - S3へのPutはAPI Gatewayで担うように仕様変更

第三世代アーキテクチャ of Datalake



ABEJA Copyright © ABEJA, Inc. All rights reserved.

バックエンド負荷、CWLogsコスト増加により
ECSに変更

Good

- Signed URLは不要になりUX改善
- メタデータ検索できるように

Bad

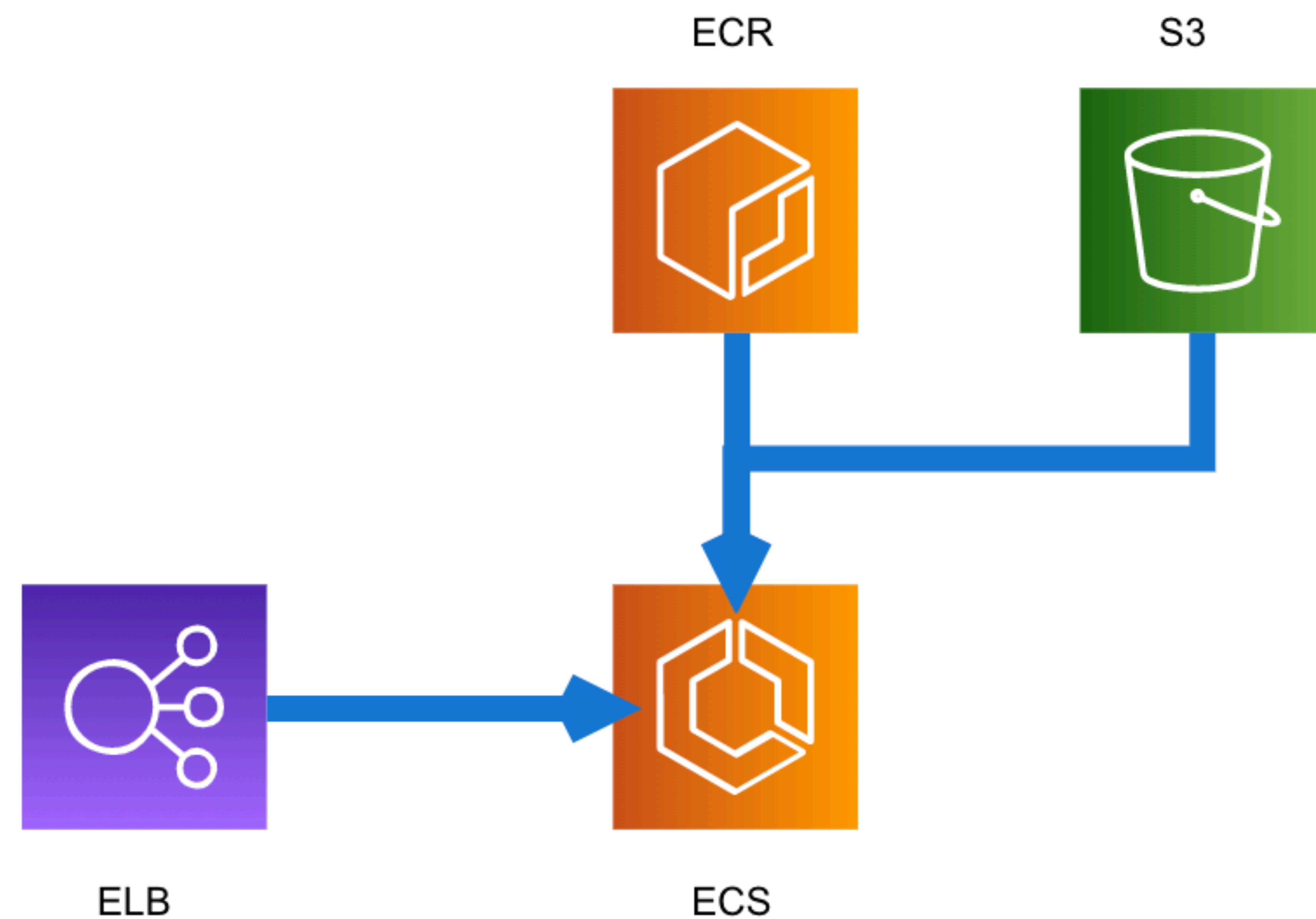
- もはやサーバレスは無くなったのでメンテコスト増加
- メタデータ機能が自由過ぎてIndex肥大化で辛い

Serving

第零世代アーキテクチャ of Serving

- 推論用APIをホスティングするサービス
- コンテナ
- 初期バージョンはプロトタイプとしてCloudFormationで ELB + ECS Service を作成

第零世代アーキテクチャ of Serving



Good

- シンプル
- 一括で関連リソースを作成・削除できる

Bad

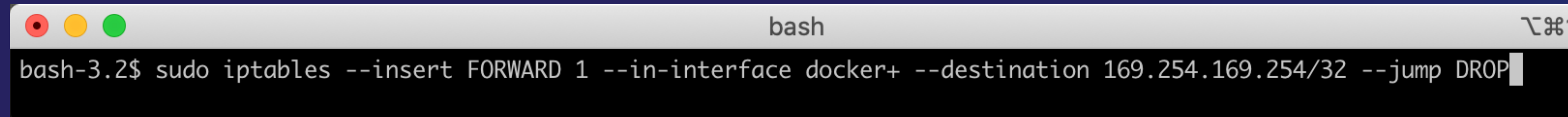
- 1Service毎に1つのELBは無駄
- CFnで作成されるのに数分かかり遅い
- CFnは非同期なのでエラー検知が難しい

第一世代アーキテクチャ of Serving

- 1 Loadbalancer = Muliti Serviceに変更 **Multi Tenant**
  - ALBのルーティングルールは100個がハードリミット
  - 顧客のAPIがどんどん乗るためすぐに破綻することが見えてた
  - またもや自前でGatewayを開発（ECS + Lambda + DynamoDB） orz
- 顧客のアプリケーションが常時エラー発生。コンテナが再起動しまくりEBSバーストクレジットを食いつぶす **Multi Tenant**
  - ECS Seviceは常に起動する仕様でサーキット・ブレイクは無い
    - システム起因のエラーでのサーキット・ブレイクは後日実装されたが、アプリ起因のサーキット・ブレイクはまだ無い
  - なので再起動繰り替えしIO食いつぶす（圧倒的ノイジーネイバー）
  - 自前（Lambda）でヘルスチェックして、一定回数Failの場合はルーティングしないように自前GatewayのDynamoDBに保存

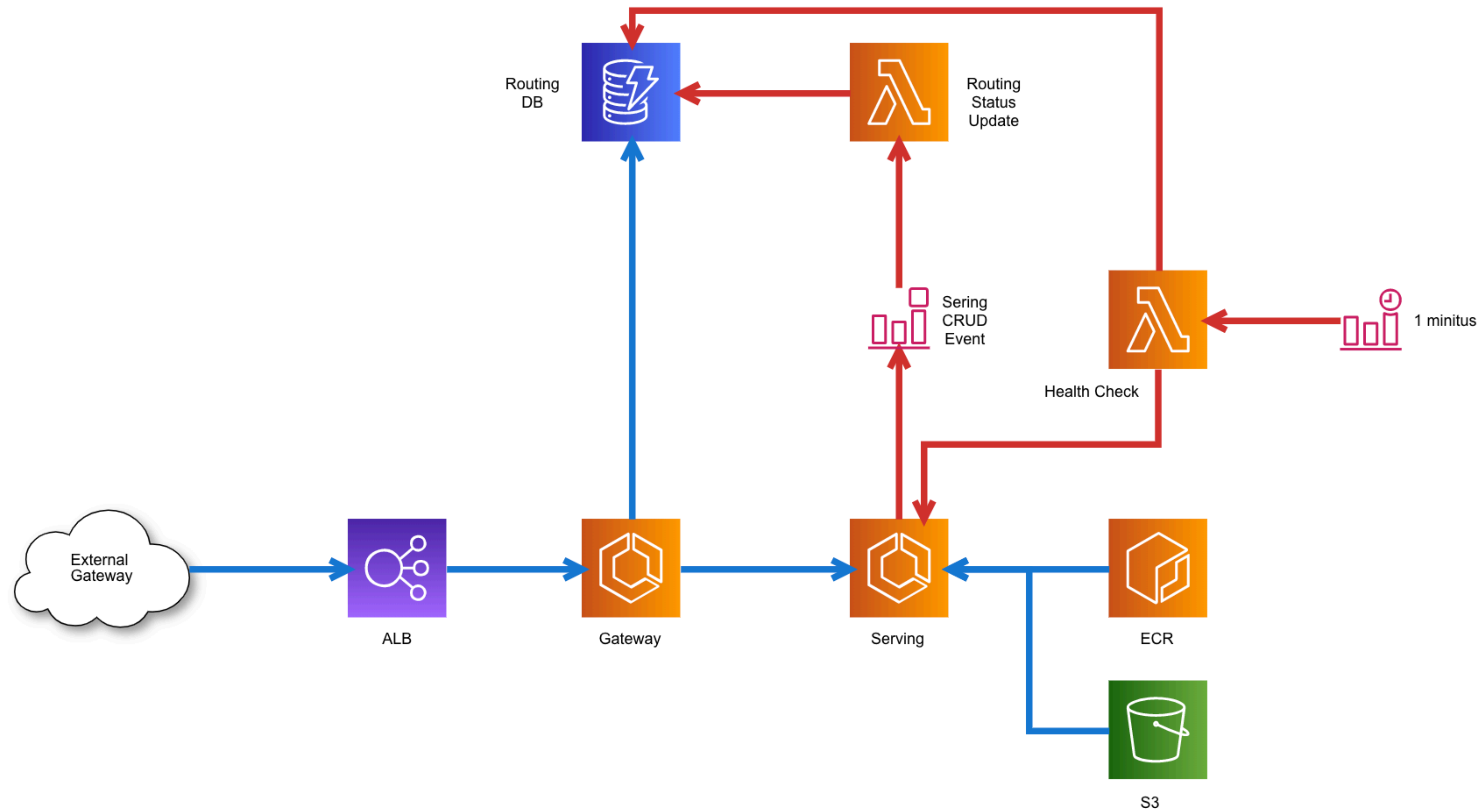
第一世代アーキテクチャ of Serving

- Blue/Green機能の実装
  - APIのエンドポイントの向き先となるモデルを簡単に切り替えられる機能
- ECSタスクからホストのIAMロールが触れてしまう問題 **Multi Tenant**
  - iptablesでmetadataへのアクセスをぶった切る



  - 最近は、awspvc ネットワークモードの場合の場合は以下のように簡単
    - ECS_AWSVPC_BLOCK_IMDS: true

第一世代アーキテクチャ of Serving



Good

- 1推論API = 1ELBでは無くなるためコストダウン
- Blue/Greenができるように
- EBSバースト問題解決
- EC2 metadataアクセスできないように

Bad

- 自前実装が多くなってきてメンテコストがかが。とはいえ、AWSの機能ではカバーしてない課題は自前実装しかない

第二世代アーキテクチャ of Serving

- ECSとEC2の相性が悪い問題
    - AutoScaling発動時にコンテナを無視してEC2をTerminateする。Drainしてよ（今は解消されている？）
    - AutoScaling発動条件はCPU予約量ベースが基本なので、コンテナサイズのスケールがかなり手間だし、計算ロジックもめんどくさい
    - 空きリソースがあっても集約されない
    - インスタンスを入れ替える際のBlue/Green Deploymentは自前実装
- 顧客の推論APIを沢山載せるとコストが肥大化する
    - スポットインスタンスをうまく扱えないか検討
    - インスタンスタイプやゾーンを散りばめたり、スポットインスタンスが売り切れた時にオンデマンドに切り替えたりする必要がある
- 検証の結果、Spotinstというサービスを入れることに

Spotinstについてはこちらを

2018-02-26

スポットインスタンスを効率的に管理するSpotinstを使おう

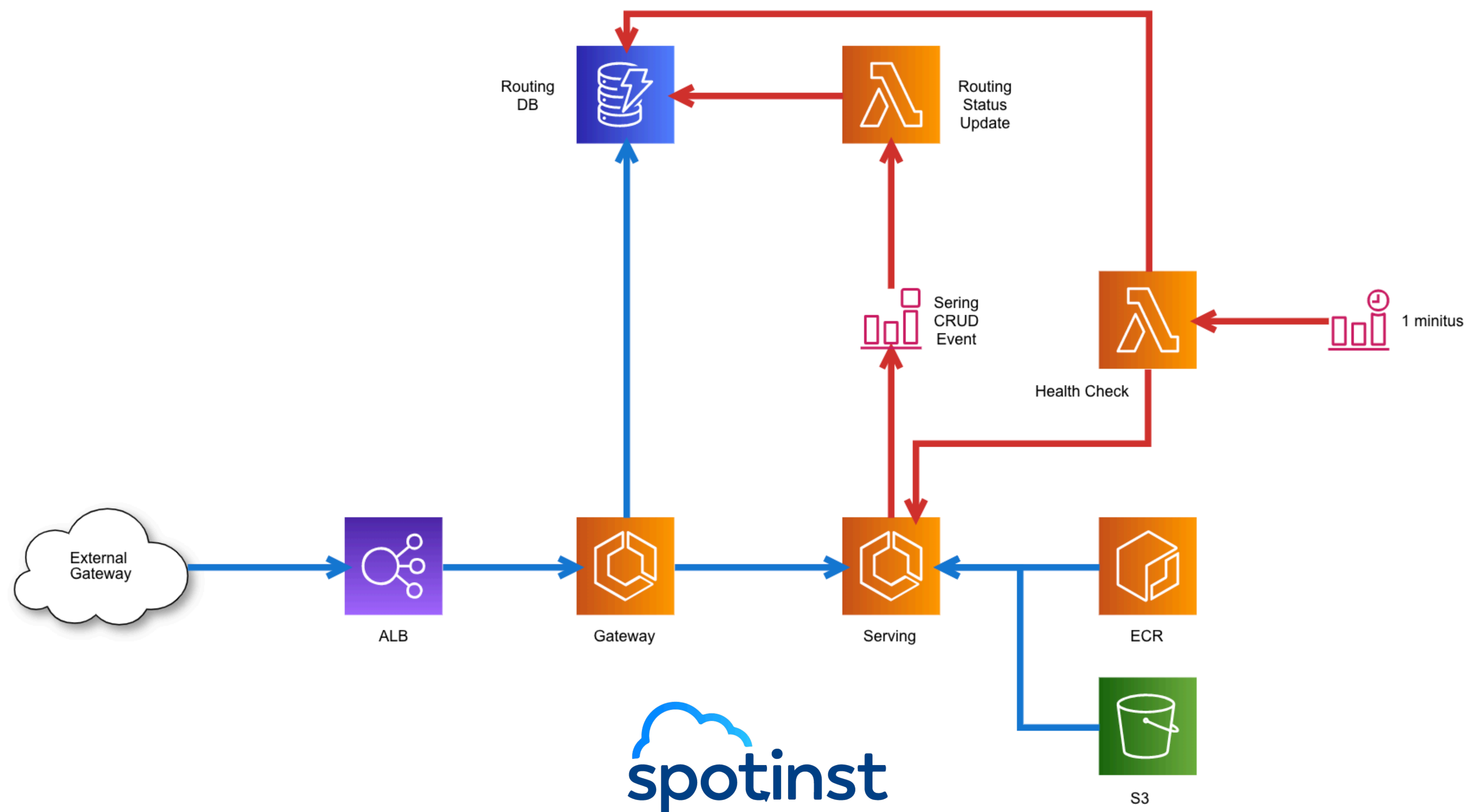
aws infra

63 B!ブックマーク

0 シェア

ツイート

第二世代アーキテクチャ of Serving



Good

- Spotinstを利用することによりECSが足りないところをカバー
- Blue/Green Deploymentが強制される
- 集約効率が上がり、60-70%コストダウン

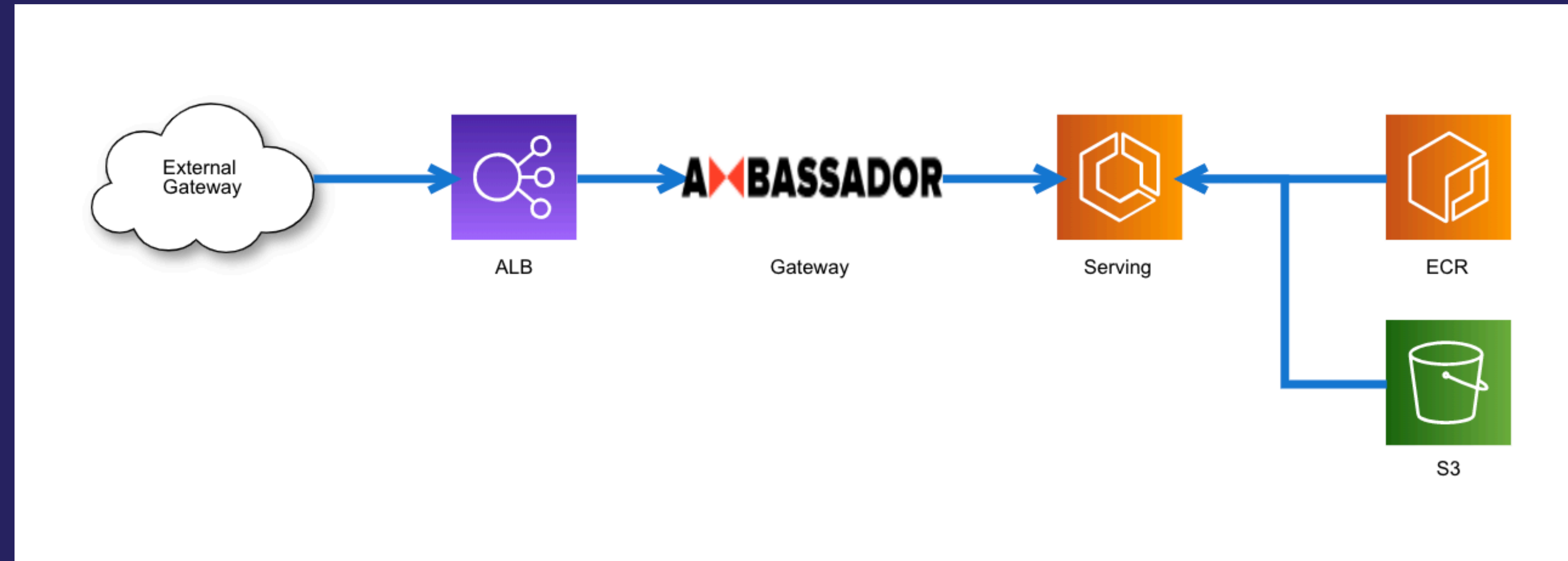
Bad

- スポットインスタンスの適用範囲を間違えると意図しない停止が発生する（Jupyter Notebookとか）

第三世代アーキテクチャ of Serving

- 1 Serviceで 300コンテナを動かすツワモノが出てきた **Multi Tenant**
  - DynamoDBのパーティショニングの偏りが発生しスロットリングが大量に
- Kubernetes機運が高まってきた。Kubernetesを活用することで
  - API Gateway相当の機能はAmbassador(Envoy)で代用可能
    - しかし、Service/Podが大量に存在するとEnvoyのルーティングルールの更新に数十秒かかることも
  - Service Discovery、ヘルスチェックは標準機能で任せられる
- Service環境もKubernetes化とともに自前実装部分の減らす方向に

第三世代アーキテクチャ of Serving



Good

- ヘルスチェック・サービスディスカバリを自前実装からKubernetesの一般的な機能に置き換え
- Gatewayの機能をKubernetesの拡張機能（Ambassador）に置き換え

Bad

- Ambassador(Envoy)の理解が必要に
- 自前実装よりは様々な面で考慮されていてメンテコストは下がるがカスタマイズ性は落ちる

第三世代アーキテクチャ of Serving

- 現状の課題
  - 顧客毎のPodのアウトバウンドの転送量の計測方法がわからない **Multi Tenant**
    - GKEにはあるらしい。Istioの出番か・・？
  - 顧客毎にコストを可視化するためにKubernetesのeventをhookしまくらないといけない **Multi Tenant**
    - これGKEはusage meteringというので出来るぽいのでEKSでもぜひやってほしい。事業部マルチテナントでも必要と思う
  - Kubernetes の結果整合性の振る舞いの上に自分たちのシステムを構築する難しさ **Multi Tenant**
    - 例えば
      - Pod の STATUS が Running になる。疎通出来るかと思いきや Ready (readinessProbe) が 0/1
      - Ready が 1/1になる。疎通できるかと思いきや Ambassador (Envoy) のとある Pod は疎通できるが、とある Pod は更新待ちのため疎通できない。分散システムにおける結果整合なので「いつ」から使えるようになった状態かをユーザに知らせるのが難しい
      - それぞれの機能が結果整合を担保しているが故に、連動して欲しいところを埋める必要がある

第三世代アーキテクチャ of Serving

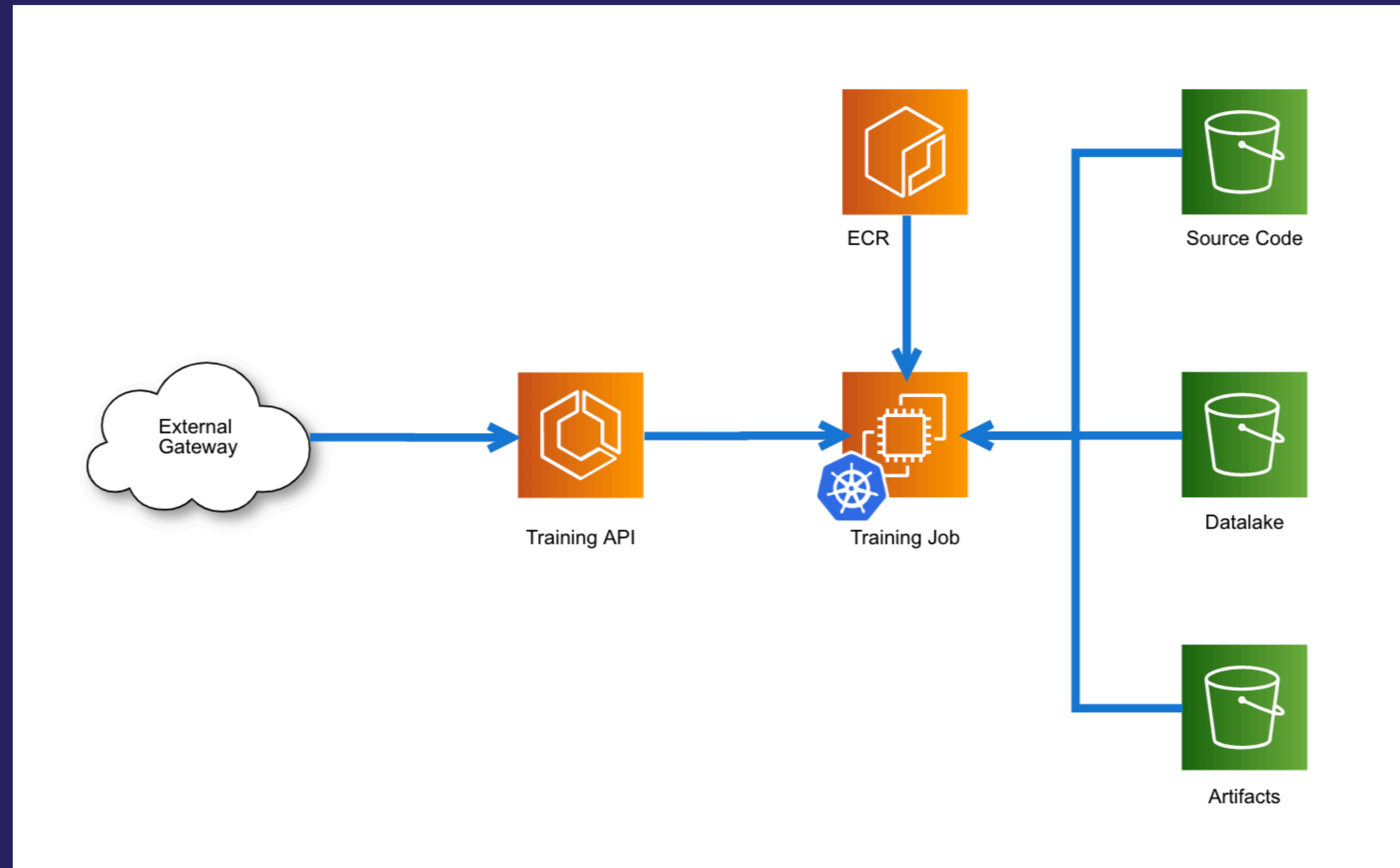
- 現状の課題
  - IPアドレスの枯渇 **Multi Tenant**
    - 1インスタンス辺り Serving = 20個、Training = 30個 を使用している。20個 * 400 Service が作られた時に8,000個のIPを使用。まさかの /16 サブネットが枯渇
  - 宣言的になりづらい **Multi Tenant**
    - SDKでデプロイしてるので「yamlをもう一度適用したら再現できるよ」状態（宣言的）にはなっていない。悲しい
  - 監視し辛い **Multi Tenant**
    - ユーザー起因の問題とプラットフォームの問題の切り分け方法が難しい
    - 顧客起因で死んでるpodが存在する。全てが正常に稼働している訳ではない
    - 独自Gatewayで5xx/4xxの監視ができない。バックエンドが顧客依存のため5xxは十分有り得る
  - マイクロサービスあるある **Multi Tenant**
    - LB/Proxyが多段なので調査しづらい。顧客にどこまでログ・メトリクスを出せば良いのか難しい

Training

第一世代アーキテクチャ of Training

- 学習ジョブを実行する基盤
- 初期バージョンからKubernetesベースで実装した
  - Job、Pod（artifact保存用コンテナ）などをECSで自前実装は非効率だったため
  - しかしEKSは無かったので on EC2 で
- 顧客のコンテナと管理系コンテナはPodを分けた **Multi Tenant**
  - Training後のモデルをs3に自動保存するコンテナ、ログを収集するコンテナは、ユーザのコードと同居させるとIAMの権限的に良くないので、別Podとしてagent的に起動
  - kube2iam を用いてPod毎にIAMロールをアタッチ（今は公式が出ている？）
- privilegedは何があってもoff **Multi Tenant**
  - GPUドライバ周りは苦労するけど頑張る

第一世代アーキテクチャ of Training



Good

- ECS上で車輪の再発明が不要になった

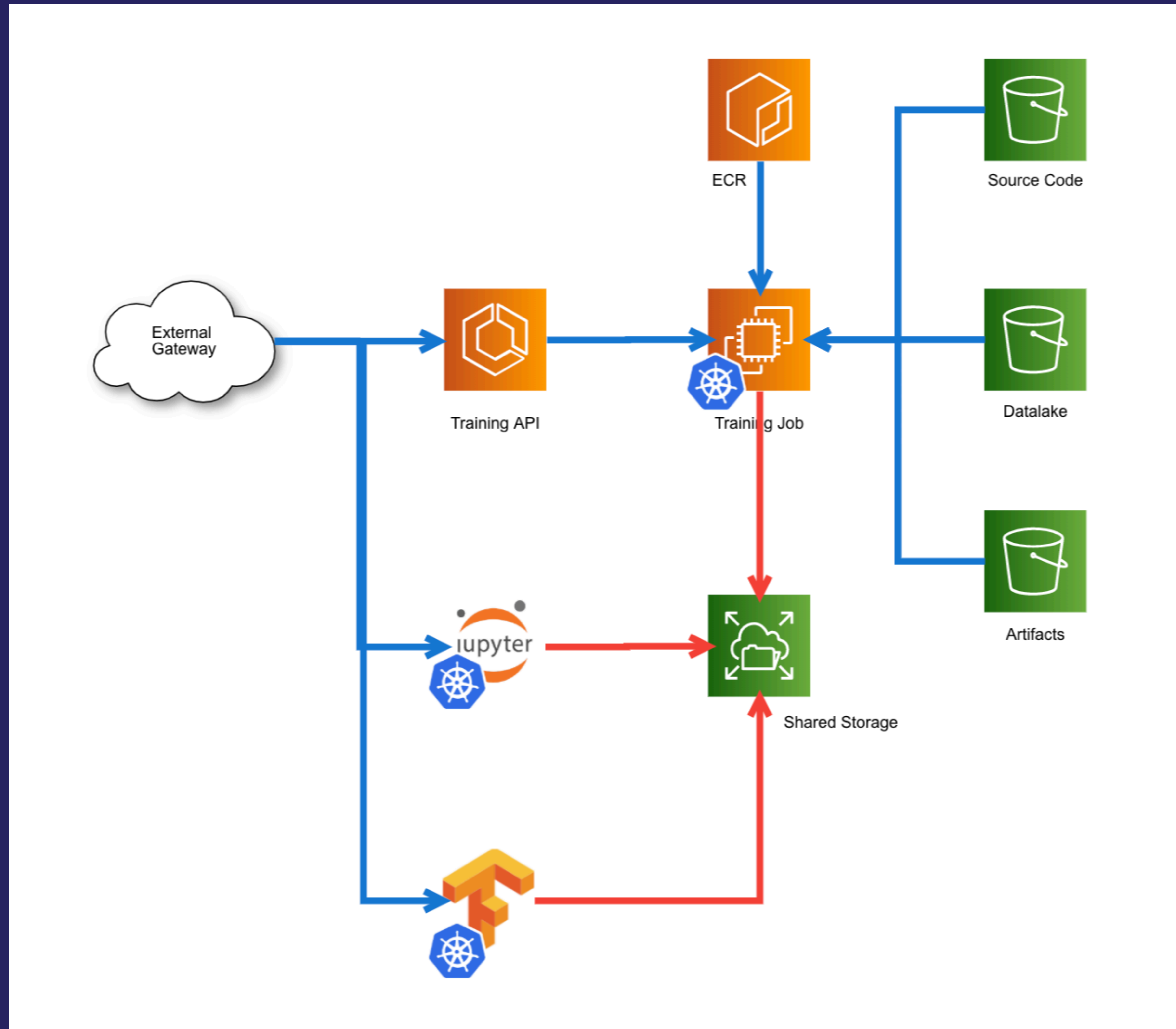
Bad

- Kubernetes on EC2 は運用が結構辛い

第二世代アーキテクチャ of Training

- Jupyter Notebookが欲しい
- Tensorboardが見たい
- Job間やNotebook間でデータセットや結果を共有したい

第二世代アーキテクチャ of Training



Good

- Jupyter Notebook、Tensorboradを提供
- 共有ストレージの提供

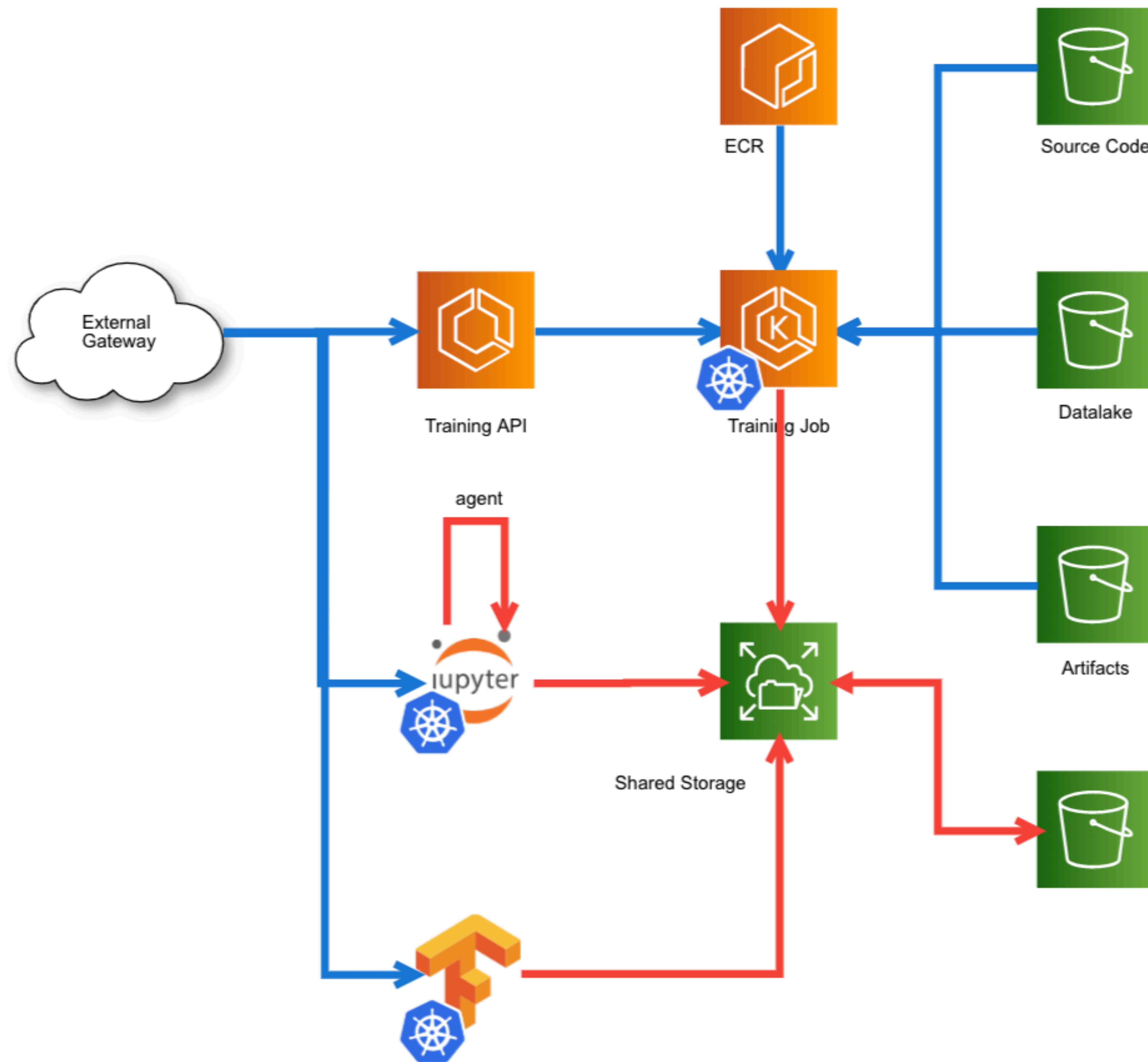
Bad

- Kubernetes on EC2 は運用が結構辛い
- Jupyter未使用時の稼働コストが無駄
- 共有ストレージが高くてNFSなので遅い

第三世代アーキテクチャ of Training

- EKSが出たので載せ替えたい
- Jupyterは使ってなければ自動で落として欲しい
- EFSのコストが肥大化するのでS3に自動で戻し、EFSの中身は自動消したい

第三世代アーキテクチャ of Training



Good

- EKS移行により運用負荷減る
- EFSからS3への書き込み・戻しによりコストダウン
- Jupyterの自動停止によりコストダウン

Bad

- EFSがNFSなので遅い

第三世代アーキテクチャ of Training

- 現状の課題
  - 年間計画の売上・原価 **Multi Tenant**
    - 精緻化する必要があるが、顧客次第なので原価を予測するとかもはやよく分からん。AWSさんどうやって管理しているのだろう
  - バグ
    - p3.16xlargeが停止せずにP万位かかってる時もあった
  - OS
    - nvidia-driverがサポートするOSである必要がある。つまりUbuntu or Amazon Linux2。「Bottlerocket」のnvidia-driverサポート期待

第三世代アーキテクチャ of Training

- 現状の課題
  - コンテナ間の依存関係
    - ログを漏れなく収集するために Affinity を駆使して log collector pod -> platform agent pod -> training job という優先順位を付けてPodを起動させると 、必要なリソースが足りないEventの発火が遅れるため Autoscaler への通知が遅れ、インスタンスの起動が遅くなる
  - Dockerイメージ・パッケージの互換性、サポート
    - 提供するDockerイメージの互換性維持が難しい
    - サポート対象
      - DLライブラリの種類 x バージョン x Pythonバージョン x CUDAのバージョン … 😷

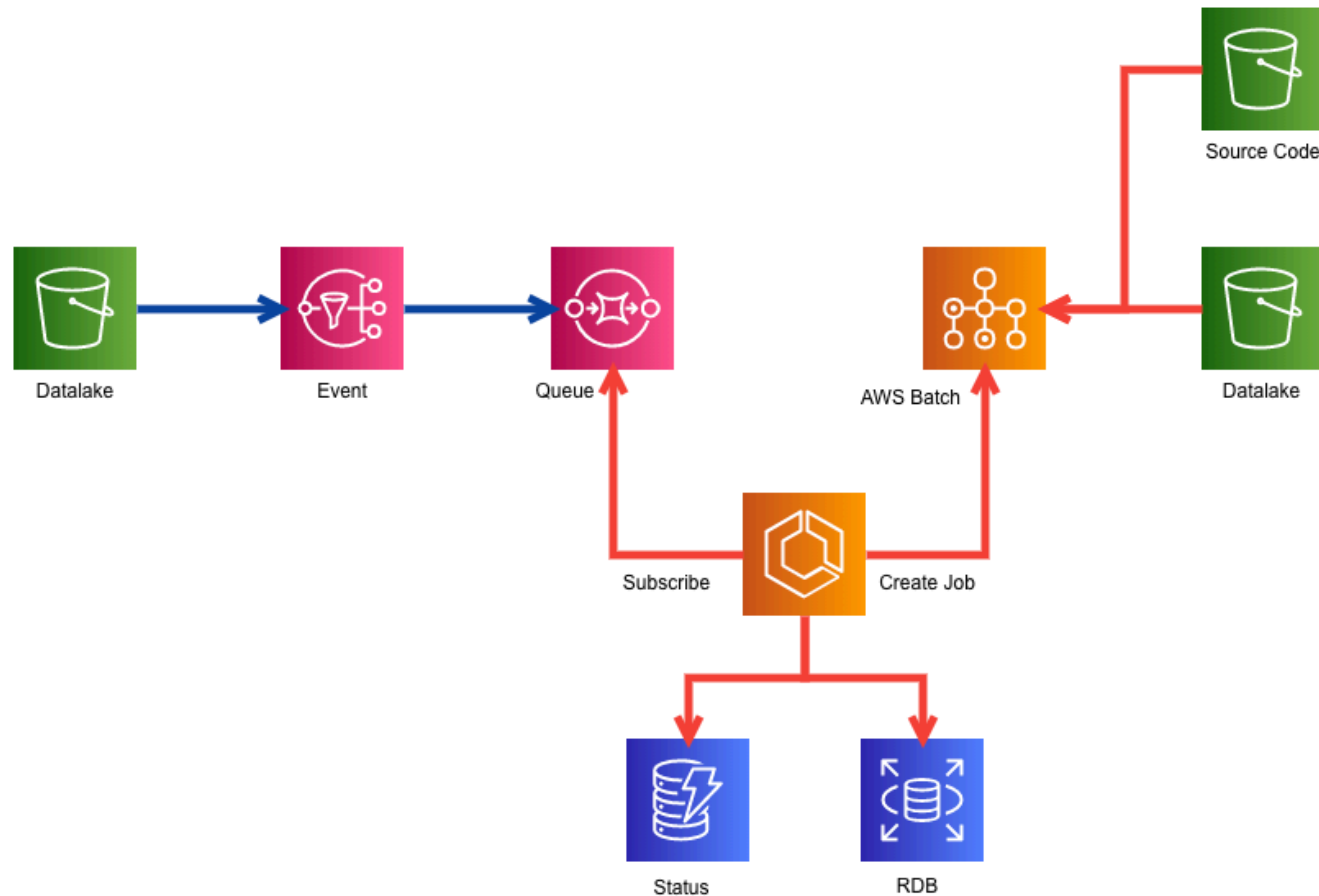
Trigger

第一世代アーキテクチャ of Trigger

- 推論ジョブを実行する基盤
- Datalakeにデータが投入されたことをトリガーに発動
- S3->SNS->SQSで、一旦キューイングする
- SQSのQueueからSubscriberが取得し、AWS Batchへタスクを投げる
- ジョブが終わればインスタンスが自動停止する

- 蓋を開ければ1分に何百とジョブが投げられる

第一世代アーキテクチャ of Trigger



Good

- 急激な負荷はSQSが吸収
- SubscriberはQueueの数でスケール
- AWS Batchは投げたら良いだけ

Bad

- 起動が遅い、メトリクス無い、ログが集約されてJob単位で見れない
- AWS Batchのユースケースに合ってなかった
- AZ間のネットワーク転送量がヤバい

Logging for Customer

第一世代アーキテクチャ of Logging for Customer

- 顧客に提供するログ基盤
- ServingとTraining、Triggerのログを提供
- ログはコアコンピタンスではないため、出来る限り自前で運用したくない

第一世代アーキテクチャ of Logging for Customer

Good

- サーバレス系（CWLogs / Kinesiss / Lambda / DynamoDB）なので運用管理不要

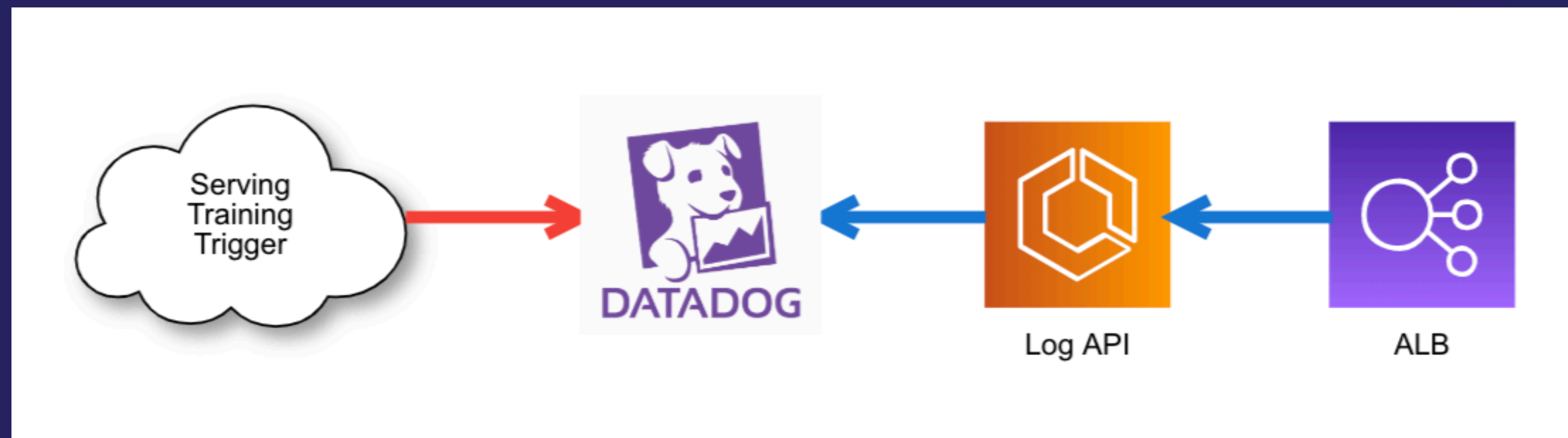
Bad

- ログ量が急増するとLambdaかDynamoDBでスロットリング起きる
- CWLogs実質使ってない割に結構高い

第二世代アーキテクチャ of Logging for Customer

- ログのバックエンドをDatadog Logsに変更
    - Service、TrainingのログをDatadog Logsに保存
    - Datadog Logs のAPIからログを取得し顧客に提供
    - ElasticSearchにするか考えたが
        - 検討次点でログ量は5億レコード/月
        - 事業の成長とロギング対象を増やすことを考えると3ヶ月毎に倍々に増える
        - 倍々に増えるElasticSearchを運用したくなかったし、ログはコアコンピタンスではないので運用コストを掛けたくなかった

第二世代アーキテクチャ of Logging for Customer



Good

- フルマネージドなので運用フリー

Bad

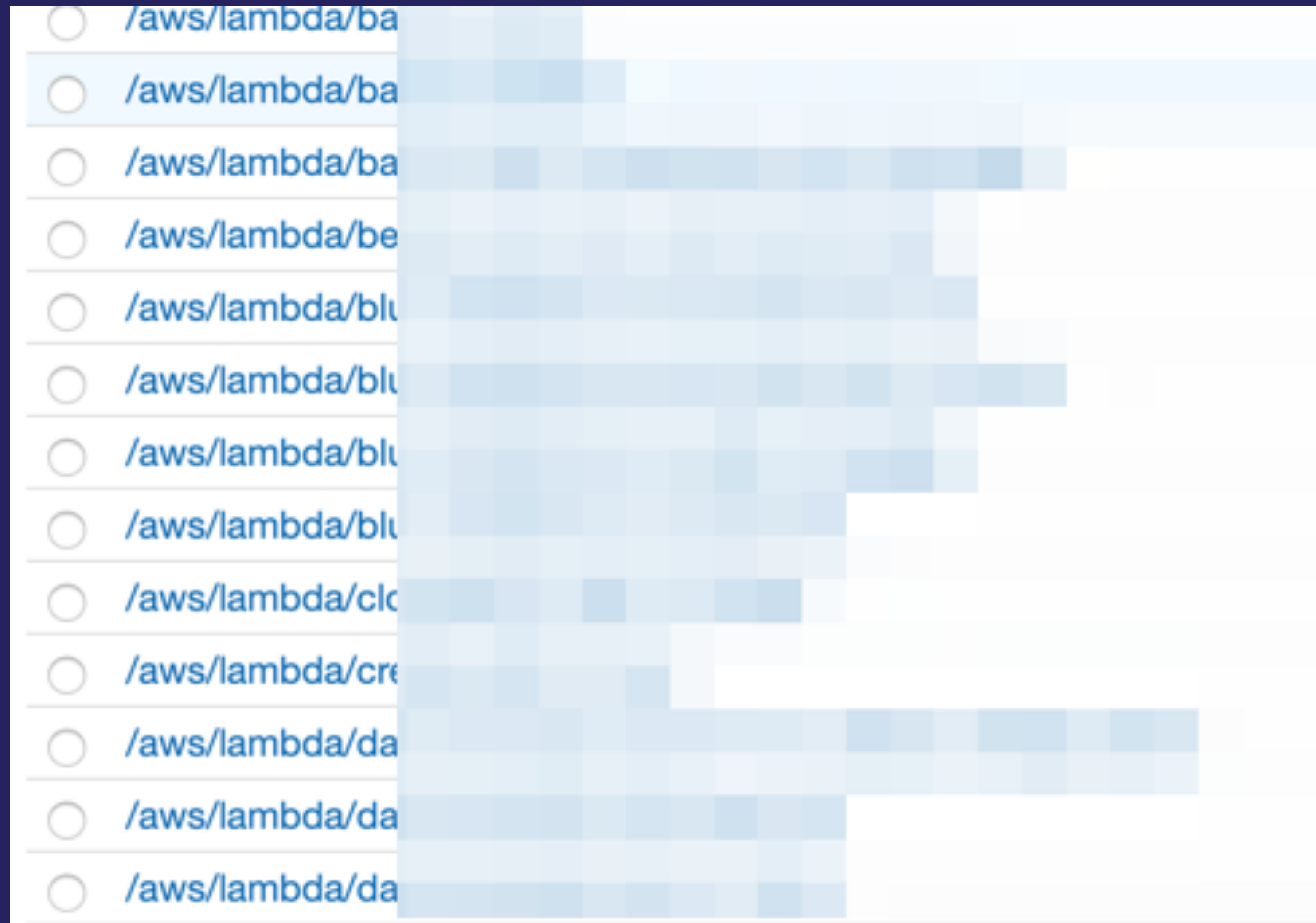
- お金がかかる。とはいえ、CWLogsより安く自前で運用するよりマシ
- Datadogの仕様に引っ張られる
  - 結果整合、順序保証無し
  - datadog-agentの挙動・仕様

Logging for System & Application

第一世代アーキテクチャ of Logging for System & Application

- 社内で利用するシステムログ・アプリケーションログの収集基盤
- ログはコアコンピタンスではないため、出来る限り自前で運用したくない
- ECSやLambdaを中心に利用していたためCloudWatch Logsにログは格納されていた
- とりあえずCW Logsを利用した

第一世代アーキテクチャ of Logging for System & Application



イベントのフィルター　すべて　2019-10-23 (03:58:45)

時間 (UTC +00:00)	メッセージ
2019-10-24	
	いまのところ古いイベントはありません。 再試行。
03:43:44	START RequestId: 1d490e6e-1d25-41f3-9d4f-5cc3094f5b7a Version: $LATEST
03:43:45	END RequestId: 1d490e6e-1d25-41f3-9d4f-5cc3094f5b7a
03:43:45	REPORT RequestId: 1d490e6e-1d25-41f3-9d4f-5cc3094f5b7a Duration: 897.76 ms Billed Duration: 900 ms Memory Size: 128 MB Max Memory Used: 91 MB Init
03:53:57	START RequestId: 02672c28-d19a-4e33-a9dc-80aa5d297661 Version: $LATEST
03:53:58	END RequestId: 02672c28-d19a-4e33-a9dc-80aa5d297661
03:53:58	REPORT RequestId: 02672c28-d19a-4e33-a9dc-80aa5d297661 Duration: 503.23 ms Billed Duration: 600 ms Memory Size: 128 MB Max Memory Used: 93 MB
03:53:59	START RequestId: 12827fc5-8df2-4f5c-a4d5-21282671b56c Version: $LATEST
03:54:00	END RequestId: 12827fc5-8df2-4f5c-a4d5-21282671b56c
03:54:00	REPORT RequestId: 12827fc5-8df2-4f5c-a4d5-21282671b56c Duration: 373.50 ms Billed Duration: 400 ms Memory Size: 128 MB Max Memory Used: 94 MB
03:58:44	START RequestId: 4d96766f-c069-4c91-9831-2a9edf652a63 Version: $LATEST
03:58:45	END RequestId: 4d96766f-c069-4c91-9831-2a9edf652a63
03:58:45	REPORT RequestId: 4d96766f-c069-4c91-9831-2a9edf652a63 Duration: 134.65 ms Billed Duration: 200 ms Memory Size: 128 MB Max Memory Used: 94 MB
	いまのところ新しいイベントはありません。 再試行。

Good

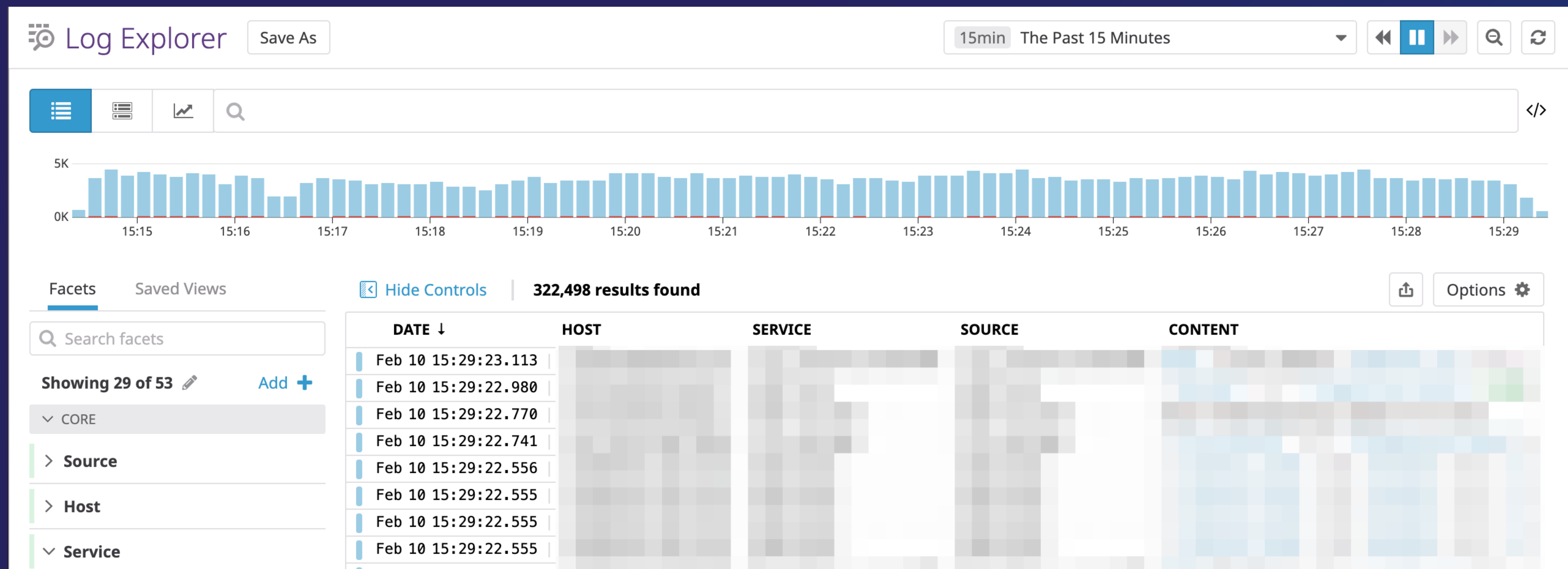
- インフラのことは考えなくて良い

Bad

- どこに格納されているか分からない
- 検索性皆無。目grep力上がる
- マイクロサービス間のログ調査とかかなりツラミしかない
- 結果、問題あった時しかログを見ない
- 以外にCWLogsは高い

第二世代アーキテクチャ of Logging for System & Application

- CWLogsへの転送は辞めて、Datadog Logsに集約した



第二世代アーキテクチャ of Logging for System & Application

OTHERS

Availability zone

- ap-northeast-1c 68.6k
- ap-northeast-1a 52.1k
- ap-northeast-1d 6.55k
- us-west-2a
- us-west-2b

Elapsed Microsecs

Http Host

Uri

Uagent

Http Status

Filter values

- 200
- 403
- 503 127k
- 500

ATTRIBUTES

authority	intern 2.elb.
bytes_received	0
bytes_sent	0
downstream_local_address	10.0.6
downstream_remote_address	10.0.8
duration	0
istio_policy_status	-
method	POST
path	/deplo
protocol	HTTP/1
request_id	62b054
requested_server_name	-
response_code	404
upstream_host	-
upstream_local_address	-
upstream_service_time	-
upstream_transport_failure_reason	-
user_agent	curl/7
x_forwarded_for	10.0.8

Add column for @response_code

Create facet for @response_code

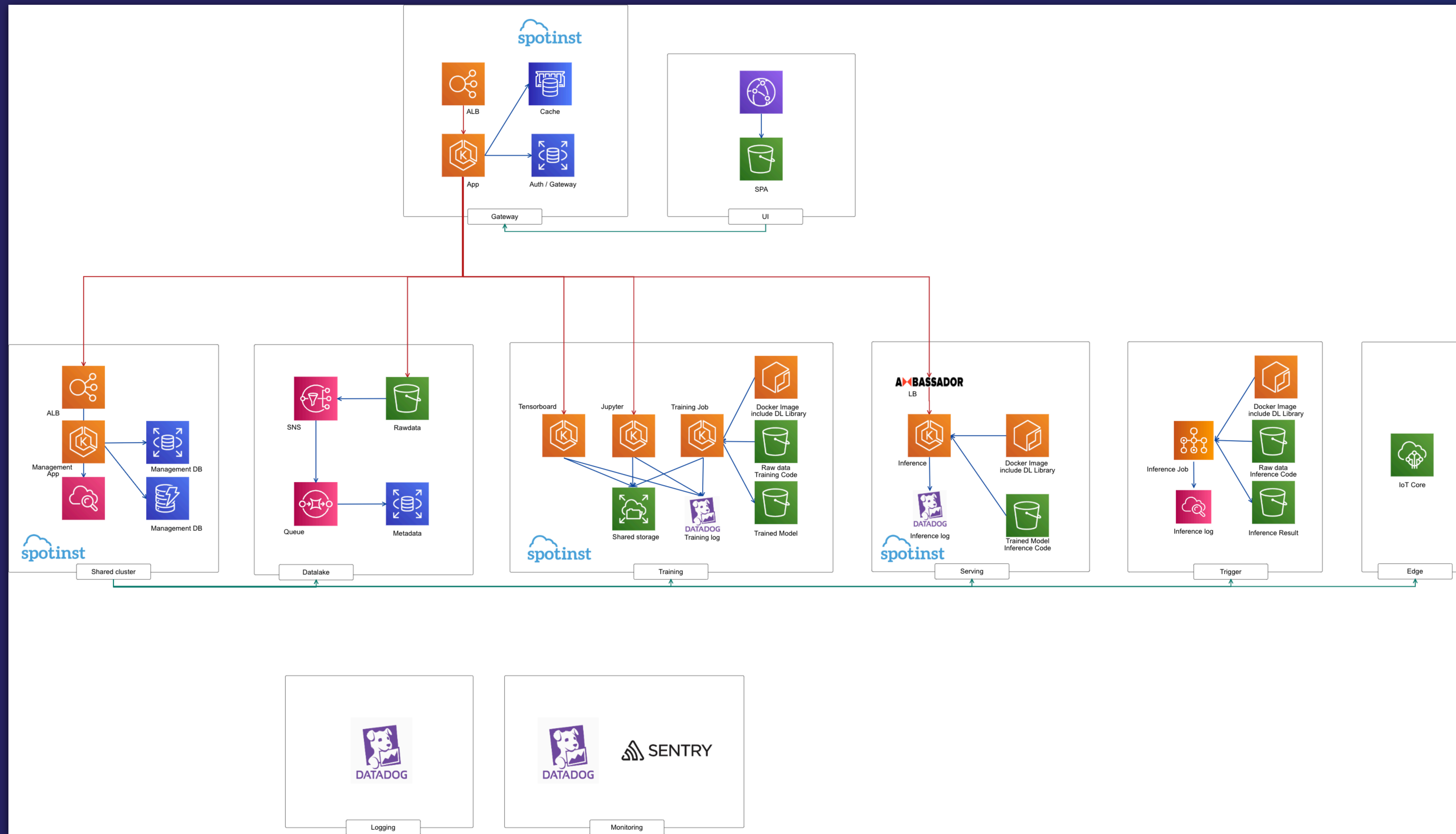
Good

- インフラのことは考えなくて良い
- 一箇所で全部検索出来る
- Tag, Attributeに対してIndex張れて検索対象にできる
- どの項目に何件あるか一目瞭然
- マイクロサービスなので、Tracing IDなどを埋め込んで追いやすくしたり
- APIの傾向分析したり、影響範囲調べたりと活用が広がった

Bad

- お金がかかる。とはいえ、CWLogsより安く自前で運用するよりマシ

で、現在に至る



結論

- 顧客のアプリケーションコードが動くプラットフォームを作ってるAWSさんスゴイ

以上