ニコニコ生放送 Web フロントエンド Kubernetes 移行ハンドブック 2022

@Himenon

Last Update 2022 年 6 月 5 日

目次

1 ニコニコ生放送 Web フロントエンドの Kubernetes 移行ハンドブック 2022 3
- 1.1 はじめに . . . 3
- 1.2 背景 . . . 3
- 1.3 読み方 . . . 4

2 移行前・移行中・移行後のネットワーク設計 4
- 2.1 移行前のネットワーク構成 . . . 4
- 2.2 移行中のネットワーク構成 . . . 4
- 2.3 移行後のネットワーク構成 . . . 5
- 2.4 Apache を移行時のロードバランサーとして選定した理由 . . . 6

3 Kubernetes の Manifest 管理 6
- 3.1 移行後の各 Component のファイル数の規模感 . . . 7
- 3.2 問題意識 . . . 7
- 3.3 TypeScript で Kubernetes を書くための支援ライブラリと Schema . . . 7
- 3.4 他のライブラリ . . . 8

4 TypeScript で Kubernetes の manifest を記述する 8
- 4.1 基本的な書き方 . . . 8
- 4.2 TypeScript で記述する特徴 . . . 10

5 TypeScript で Manifest を生成する Generator のアーキテクチャ 13
- 5.1 アーキテクチャが解決すること . . . 13
- 5.2 具体的な実装例 . . . 14
- 5.3 特徴的なこと . . . 15
- 5.4 Manifest 生成はどう使うのがよいか？ . . . 16

6 Argo CD 17
- 6.1 他チームとの棲み分け . . . 17
- 6.2 フロントエンドのチームが管理するマイクロサービスのためのリポジトリと ArgoCD の Application 設定 . . . 18
- 6.3 フロントエンドが管理するマイクロサービスのための Manifest のリポジトリ . . . 19
- 6.4 `infrastructure` リポジトリのブランチ設計 . . . 20
- 6.5 デプロイの速度が重要な理由 . . . 21

7 Argo Rollouts の導入 22
- 7.1 Argo Rollouts とは . . . 22
- 7.2 Istio + Argo Rollouts . . . 22
- 7.3 Canary Deploy を実施する . . . 22
- 7.4 DataDog でのモニタリング例 . . . 24
- 7.5 これから . . . 24

8 Slack Bot による自動化 25

8.1 バージョンアップのシーケンス図 . . . 25
8.2 既存のアプリケーションの CI と Kubernetes Manifest のリポジトリの連携 . . . 26
8.3 Slack Bot によってデプロイ作業を最小工数で終わらせる . . . 26

9 BFF と Istio 27
9.1 Istio の利用 . . . 27
9.2 Istio と Envoy の関係 . . . 28
9.3 Ingress Gateway をセットアップする . . . 28
9.4 istio-proxy のサイドカーが不要なケース . . . 30
9.5 BFF でサービスメッシュが有効だと何が良いか . . . 30

10 アクセスログ 30
10.1 Ingress Gateway とアクセスログ周りのアーキテクチャ . . . 31
10.2 具体的な設定 . . . 32
10.3 これから . . . 37

11 Istio Ingress Gateway の設定 37
11.1 hosts でルーティングを分ける . . . 38
11.2 Header や QueryParameter でルーティングする . . . 41

12 Global RateLimit 42
12.1 Global Ratelimit の設定 . . . 42
12.2 istio-proxy と Rate Limit のマイクロサービスとの連携 . . . 44

13 Local RateLimit 46
13.1 envoy と nginx の Rate Limit アルゴリズムの違い . . . 46
13.2 envoy と nginx の設定例 . . . 47
13.3 どちらを選ぶか . . . 48
13.4 今後どうするか . . . 49

14 RateLimit で負荷の上昇を防げないパターン 49
14.1 発生メカニズム . . . 49

15 水平スケール 51
15.1 Horizontal Pod Autoscaler . . . 51
15.2 リソースの値をどうやって決めるか . . . 52
15.3 バースト性のあるリクエストをどうやって耐えるか . . . 52
15.4 水平スケール設計の今後 . . . 53
15.5 参考文献 . . . 53

16 負荷試験 53
16.1 負荷試験の目的 . . . 53
16.2 何を試験するか . . . 54
16.3 試験方法 . . . 54

17 モニタリング 56
17.1 利用しているモニタリングツール . . . 56

17.2 BFF サーバーの何を観測するか . . . 57
17.3 DataDog の例 . . . 57
17.4 モニタリングの次にやること . . . 59

18 負荷対策 59
18.1 nodeSelector . . . 59
18.2 PodAntiAffinity . . . 60

19 アプリケーションの移行 61
19.1 Graceful Shutdown . . . 61
19.2 静的リソースのアップロード . . . 62
19.3 ルーティングの切り替え . . . 63

20 通信経路の切り替え 64
20.1 移行の流れ . . . 65
20.2 ロールバック設計 . . . 66
20.3 スケジュールの振り返り . . . 66
20.4 移行中・移行後の運用 . . . 67
20.5 まとめ . . . 67

1　ニコニコ生放送 Web フロントエンドの Kubernetes 移行ハンドブック 2022

1.1　はじめに

2021 年 6 月から 2022 年 3 月の 9 ヶ月間でニコニコ生放送のフロントエンドに関係するマイクロサービスを Docker Swarm から Kubernetes へ移行しました。その移行前から移行期間中、移行後にかかる設計や運用方法を実際に実施した事例を元に整理して紹介して行きます。

1.2　背景

本題に入る前に、移行前のニコニコ生放送の Web フロントエンドチームが管理している「開発からリリース、運用」までのインフラを軽く紹介します。

- js や css の静的リソースビルド
- express.js を搭載した Web Application Server の開発（Backend for Frontend）
- Docker Image のビルド
- 静的リソースのデプロイ
- Apache や nginx、OpenResty といった L7 ロードバランサーによるトラフィックマネジメント
- Jenkins を利用した Docker Swarm クラスタの Blue/Green デプロイ
- ログの送信
- 監視（DataDog など）

1.2.1　移行に主に携わった人たち

Web フロントエンドのチームから主に 2 人で移行作業を進めていきました。他のマイクロサービスも別のチームが担当し、知見やリソースの使用状況などを共有しつつ作業を行いました。背景で紹介したようなインフラの歴史的経緯や、移行後にあるべき状態を整理しつつ、Kubernetes に素早く移行するための方法を設計し実現していきました。

1.2.2　移行後の Kubernetes の規模感

Kubernetes 上でのニコニコ生放送の**フロントエンドに関係するマイクロサービス**の Manifest の数は 1 つのクラスターに対して以下の規模で Component があります。

Component	ファイル数
v1/Deployment	22
v1/Service	60
v1/Config Map	15
batch/v1/Job	14
argoproj.io/v1alpha1/Rollout	18
networking.istio.io/v1beta1/VirtualService	20
networking.istio.io/v1alpha3/EnvoyFilter	20

※ 規模感と具体的な用途がわかりやすい Component を記しています。

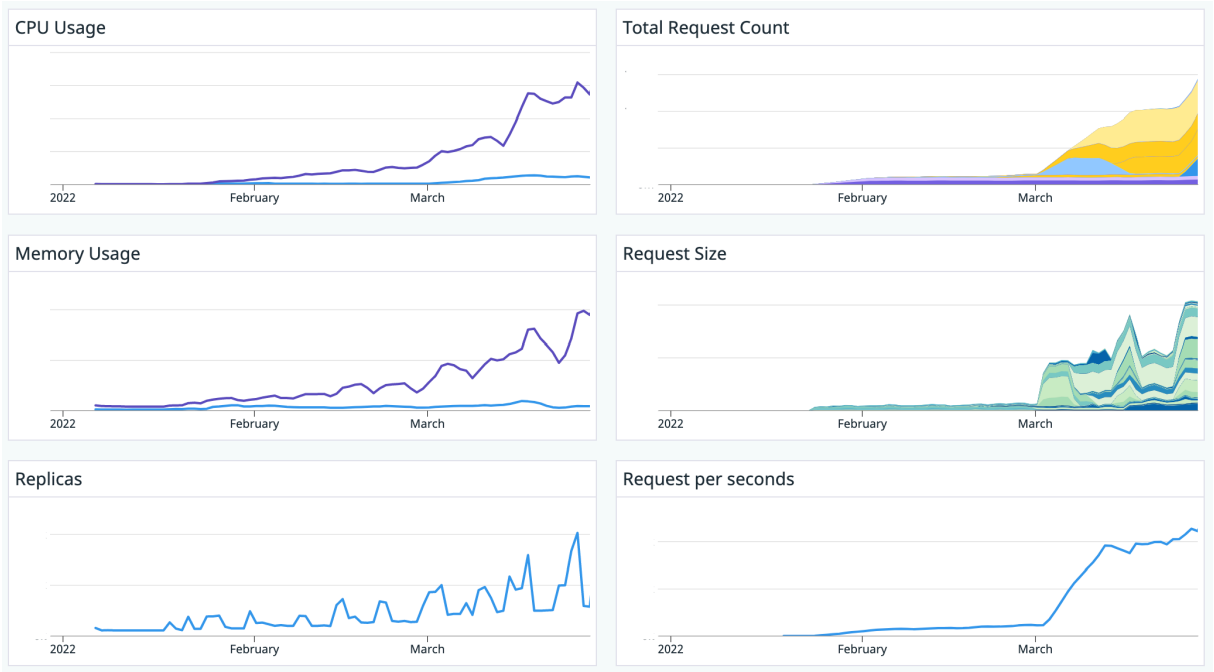


図 1: 移行期間中の各種メトリクス

1.3 読み方

Backend For Frontend のサーバーを管理する、Web フロントエンドの開発者視点から Kubernetes 上アプリケーションを稼働させるためのプラクティスを整理しました。基本的はステートレスなアプリケーションですが、非機能要件にあたる部分が本内容で紹介する主要な部分となります。ただし、ゼロベースから築かれたプラクティスではなく「移行」という前身のあるアーキテクチャ構成となっており、必ずしも「ベスト」とは言えない構成も存在します。

したがって、Web フロントエンドの開発をしている読者は非機能要件にあたる部分の運用や費用のコスト感覚を意識しつつ、それらのコストを下げるためにアプリケーションレベルで何ができるか考えるとインフラを触るときに役に立つかもしれません。逆に普段インフラを触っている読者は Web フロントエンドのインフラを管理するときに考慮する内容の整理として参考になるかもしれません。

2 移行前・移行中・移行後のネットワーク設計

2.1 移行前のネットワーク構成

移行前の Docker Swarm のネットワーク構成図のようになっています。

Traffic を最上位のロードバランサーで受けた後、Apache や nginx、OpenResty といった複数の L7 ロードバランサーで受けた後に Docker Swarm のクラスターに到達し、コンテナネットワークに接続されて Web アプリケーションがレスポンスを返す様になっています。

2.2 移行中のネットワーク構成

図は Docker Swarm から Kubernetes に移行するにあたってネットワークのネットワーク構成になります。基本的な移行方法としてはクラスター外のロードバランサーから新旧の環境に `PATH` 単位でしていくことで段階的に移行することが可能です。

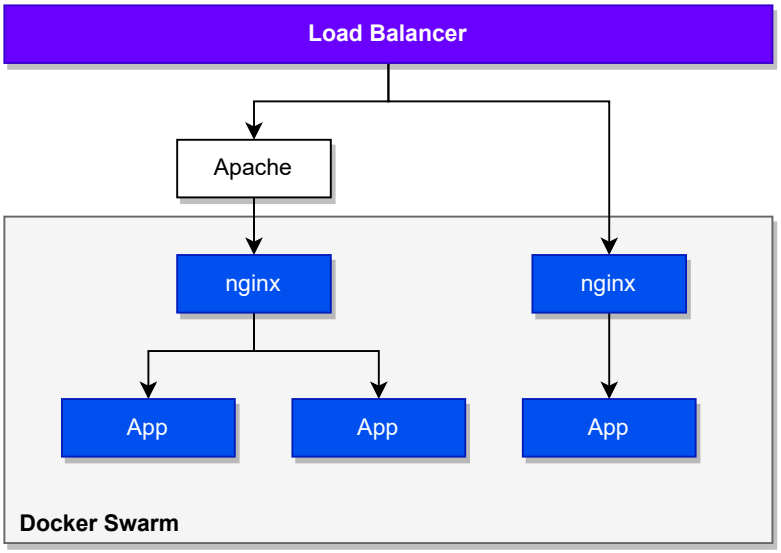


図 2: 移行前のネットワーク構成

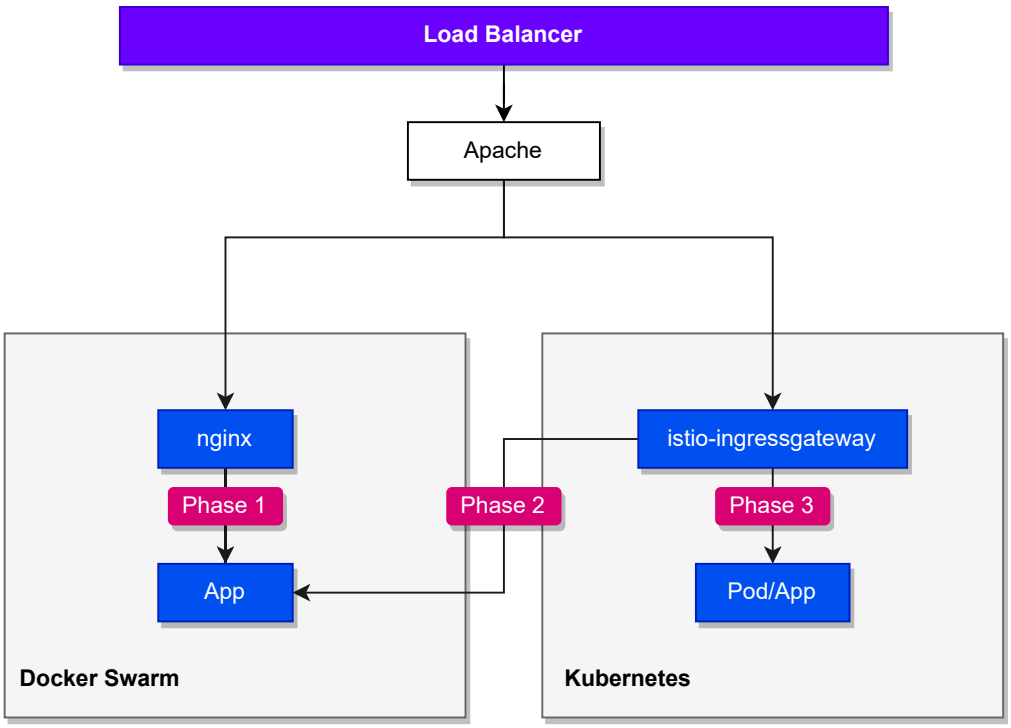


図 3: 移行中のネットワーク構成

今回 Apache をそのハンドルに選んだ理由は後述しています。

2.3 移行後のネットワーク構成

Kubernetes 上でのネットワーク構成は図のようになりました。Apache による Load Balance は移行前の Docker Swarm と同じですが、全てのアプリケーションは一度全て Istio の Ingress Gateway を経由するようになりました。

また、Rate Limit など nginx で実施していたシステムの防衛処理は Kubernetes クラスター全体に対する

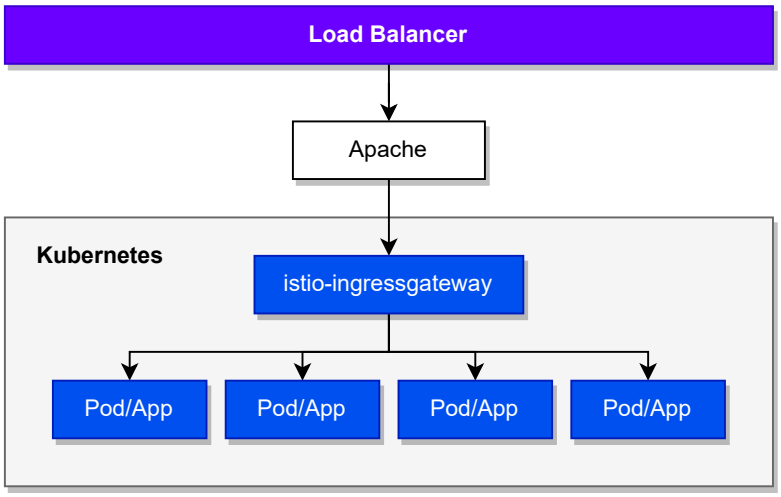


図 4: 移行とのネットワーク構成

Global Rate Limit と Pod 単位の Local Rate Limit に機能を分割しました。これらの詳細は別のページで紹介しています。

- Rate Limit に関して
- Istio Ingress Gateway に関して
- アクセスログに関して

2.4 Apache を移行時のロードバランサーとして選定した理由

1. Apache という L7 ロードバランサーは Request / Response Header などの処理をすでに抱えており、短期間で移植することは困難であることが想定されました。
2. Apache よりも Down Stream 側にあるロードバランサーをチームの権限でコントロールすることはできないため変更のリードタイムが長くなることが想定されました。
3. Apache ではなく nginx や HAProxy だったら移植できたかというとそういう問題でもなく、直接 Kubernetes クラスタで Traffic を受けたときに Kubernetes 自体や Istio Ingress Gateway が Traffic の負荷に耐えられるか、信頼できる負荷試験の結果が存在しないため、旧環境に切り戻すリスクが高いと判断しました。

すなわち、運用実績とオペレーションの容易さから Apache で移行することが移行時に取り得る選択肢として最良と判断しました。

※結果は言わずもがなですが、書いてあるとおりスケジュール通りに移行を完遂させています。

3 Kubernetes の Manifest 管理

ここでは Manifest の管理をどのように実施しているかについて紹介します。結果から言えば、Kubernetes で利用する Manifest を生成する Generator を TypeScript で構築しました。

どのように構築して運用しているのか説明していきます。

3.1 移行後の各 Component のファイル数の規模感

導入でも提示していますが改めて、**フロントエンドに関係するマイクロサービス**に関係する Manifest は以下の規模で存在しています。これは簡単に管理するとは言えないコンポーネント数があり、これからも増えていきまうｓ.

Component	ファイル数
v1/Deployment	20
v1/Service	60
v1/Config Map	15
batch/v1/Job	15
argoproj.io/v1alpha1/Rollout	20
networking.istio.io/v1beta1/VirtualService	20
networking.istio.io/v1alpha3/EnvoyFilter	20

3.2 問題意識

移行前の段階ですでにファイル数は YAML で保守するには困難な量が発生することはわかっており、ツールによる補完支援やテスト無しでは必ず破綻することが容易に想定されました。また、これらは最初に定めた 2 つの目標に反します。

- デプロイが素早く簡単にそして安全に実施できる
- Web フロントエンド開発者が更新に必要な最低限の設定の変更を簡単に実施できる

3.2.1 TypeScript で Manifest の保守面の問題を解決する

これらを網羅的に解決する一つの方法として TypeScript により Kubernetes の YAML を生成することです。TypeScript 自体の利点は各種記事に譲るとして、Kubernetes を運用するチームの背景として TypeScript を日常的に利用している Web フロントエンドのエンジニアたちです。

したがって、TypeScript で Manifest を記述すること自体は非常に障壁がほとんど皆無という状態です。また Manifest 自体のテストも TypeScript から YAML を生成するタイミングで `Exception` を投げてしまえば良いだけなので、テストの方針も非常に単純になります。

仮に TypeScript で書くのを辞めたいといった場合は生成された YAML を持っていけば良いので、TypeSCript 自体を切り捨てることも簡単になります。

以上の理由から TypeScript で記述しない理由が移行の設計段階で存在しないため、Manifest を YAML で書くことを初手で捨て、TypeScript で記述するようにしました。

3.3 TypeScript で Kubernetes を書くための支援ライブラリと Schema

Kubernetes は `CustomResourceDefinition` を定義する際 OpenAPI Schema V3 で記述できます。 これによって Schema が Apply 時に Validation されています。逆に言えば、OpenAPI Schema を TypeScript の型定義に書き起こしてしまえば Validation を TypeScript の静的型付けに変換することができます。

幸いにして筆者は OpenAPI Schema と TypeScript の話にはちょっとだけ詳しいので、手前味噌ですが@himenon/openapi-typescript-code-generator を利用して Kubernetes の型定義を生成しました。

- @himenon/kubernetes-typescript-openapi
- @himenon/argocd-typescript-openapi
- @himenon/argo-rollouts-typescript-openapi

もちろん他にも同じようなアプローチで型定義を提供しているものもありますが、以下の点で見送りをしています。

- TypeScript の Object に対してシンプルに型定義を当てたい
  - これはライブラリ側のメンテナンスが滞っても自分たちで書き直すことができるため
- ArgoCD や ArgoRollouts、Istio など他の Custom Resource 利用時も同じライブラリの使い勝手になるようにしたい
- 最新だけでなく任意の古いバージョンもサポートするようにする

これらを考えたときになるべくライブラリは薄く実装されているのが望ましく、型定義ライブラリを Fork したときも簡単にメンテナンスできる実装ベースが必要でした。これらの条件を満たす設計コンセプトで豊富な知見があるライブラリは@himenon/openapi-typescript-code-generator でした。
次の節でより詳細に紹介します。

- TypeScript で Kubernetes の manifest を記述する
- TypeScript で Manifest を生成する Generator のアーキテクチャ

3.4 他のライブラリ

Kubernetes 向け TypeScript のライブラリ

- cdk8s
- kosko

Kubernetes の Definition が定義れている Component は CustomResourceDefinition があります。

4 TypeScript で Kubernetes の manifest を記述する

ここでは基本的な書き方について紹介します。

4.1 基本的な書き方

NodeJS で動かすスクリプトとして次のようなに記述してきます。これを `ts-node` などで実行すると `deployment.yml` が出力され、`kubectl apply -f deployment.yml` とすることで Kubernetes 上に Pod が起動します。

```
import * as fs from "fs";
import * as yaml from "js-yaml";
import type { Schemas } from "@himenon/kubernetes-typescript-openapi/v1.22.3";

const podTemplateSpec: Schemas.io$k8s$api$core$v1$PodTemplateSpec = {
  metadata: {
    labels: {
```

```
      app: "nginx",
    },
  },
  spec: {
    containers: [
      {
        name: "nginx",
        image: "nginx:1.14.2",
        ports: [
          {
            containerPort: 80,
          },
        ],
      },
    ],
  },
};

const deployment: Schemas.io$k8s$api$apps$v1$Deployment = {
  apiVersion: "apps/v1",
  kind: "Deployment",
  metadata: {
    name: "nginx-deployment",
    labels: {
      app: "nginx",
    },
  },
  spec: {
    replicas: 3,
    selector: {
      matchLabels: {
        app: "nginx",
      },
    },
    template: podTemplateSpec,
  },
};

const text = yaml.dump(deployment, { noRefs: true, lineWidth: 144 });
fs.writeFileSync("deployment.yml", text, "utf-8");
```

4.2 TypeScript で記述する特徴

TypeScript で記述したときの特徴を紹介します。

4.2.1 YAML の記法に悩まれなくて済む

まず一番わかりやすいのは YAML の記法のブレがなくなります。YAML は出力された結果であり、その結果を出力する処理が記法を規格化するため YAML の記法に関する一切のレビューが不要になります。

1. space か tab indent か
2. indent は space 2 か 4 か
3. 複数行コメントは | か>のどちらで初めるか
4. アルファベット順にソートするか

など。これらのことを一切考える必要がありません。

4.2.2 コメントが書きやすい

TypeScript のコードコメントがそのまま利用することができます。エディタ上で変数名などをホバーしたときにコメントが見えるなどの可視化支援を受けることができます。

また、そのままドキュメントになるためマニフェストとドキュメントの乖離を防ぐことができ、ロストテクノロジーになることに対する予防措置が同時に実施できます。

```
/** podTemplate に対するコメント */
const podTemplateSpec: Schemas.io$k8s$api$core$v1$PodTemplateSpec = {};

const deployment: Schemas.io$k8s$api$apps$v1$Deployment = {
  apiVersion: "apps/v1",
  kind: "Deployment",
  metadata: {
    name: "nginx-deployment",
    labels: {
      /** このラベルを付ける理由.... */
      app: "nginx",
    },
  },
  spec: {
    /** replicas が 3 で妥当な理由... */
    replicas: 3,
    /** この Selector を付ける理由.... */
    selector: {
      matchLabels: {
        app: "nginx",
      },
    },
    template: podTemplateSpec,
```

```
  },
};
```

4.2.3 「変数」が依存関係を表す様になる

Kubernetes で基本的な Service と Deployment というセットを考えたとき、Service 間通信するためには Service の Selector を Pod の Label と一致させる必要があります。これを TypeScript で表現する場合、Selector と Label の部分を変数化してしまえば確実に疎通ができる Service と Deployment のマニフェストを生成することができます。

他にも推奨されるラベルにある `app.kubernetes.io/version` なども漏れなく適切に指定されるようになります。

```
const Namespace = "mynamespace";

export const generateService = (applicationName: string, applicationVersion: string):
↪  Schemas.io$k8s$api$core$v1$Service => {
  return {
    apiVersion: "v1",
    kind: "Service",
    metadata: {
      name: applicationName,
      namespace: Namespace,
    },
    spec: {
      type: "ClusterIP",
      selector: {
        app: applicationName,
        "app.kubernetes.io/name": applicationName,
      },
      ports: [
        {
          name: `http-${applicationName}-svc`,
          port: 80,
          targetPort: 80,
        },
      ],
    },
  };
}


export const generateDeployment = (applicationName: string, applicationVersion:
↪  string): Schemas.io$k8s$api$apps$v1$Deployment => {
```

```
  return {
    apiVersion: "apps/v1",
    kind: "Deployment",
    metadata: {
      name: applicationName,
      namespace: Namespace,
      labels: {
        app: applicationName,
        "app.kubernetes.io/name": applicationName,
      },
      annotations: {},
    },
    spec: {
      selector: {
        matchLabels: {
          "app.kubernetes.io/name": applicationName,
        },
      },
      /** 省略 */
    },
  };
}

const applicationName = "my-nginx";
const applicationVersion = "1.14.2";

generateService(applicationName, applicationVersion);
generateDeployment(applicationName, applicationVersion);
```

4.2.4 テンプレートの表現力が増す

例えば NodeJS や Go Lang、Scala など様々な言語で記述されているマイクロサービスの基本的な Deployment のテンプレートなども用意できるようになります。これは例えば/a と/b のエンドポイントが同じサーバーから提供されているが、水平スケールする単位や CPU/MEM などの各種リソースを分離して管理したい場合に Manifest を分割したい場合に大いに役立ちます。うまく Manifest の Generator が設計されていれば数分のオーダーで分割ができ、即日デプロイすることができます。

```
export const generateNodeJsDeployment = ():Schemas.io$k8s$api$apps$v1$Deployment =>
↪  {};
export const generateRubyOnRailsDeployment = ():Schemas.io$k8s$api$apps$v1$Deployment
↪  => {};
export const generateScalaDeployment = ():Schemas.io$k8s$api$apps$v1$Deployment =>
↪  {};
```

4.2.5 Generator 内部で Error を throw することがテストになる

Manifest を Generate する際に立地なテストフレームワークは不要で、単純に Exception を発生させることがテストになります。例えば Service や Job などのリソースタイプは metadata.name に指定可能な文字列や文字数が決まっています（参照）。

大きな変更が入った後に kubectl apply を実施して、この問題が発覚するとトラブルシュートの時間が掛かるため、Manifest を Generate する際に具体的なエラーメッセージを出力して処理を中断してしまえば悩む時間が最小限にできます。手元で Generate せずに Pull Request 投げた場合は CI で Generate を再度走らせてテストを実施することができます。

```
export const validateMetadataName = (text: string, throwError?: true): string => {
  if (throwError && text.length > 63) {
    throw new Error(`May not be deployed correctly because it exceeds 63
↪   characters.\nValue: "${text}"`);
  }
  return text.slice(0, 63);
};

export const generateJob = (applicationName: string): Schemas.io$k8s$api$batch$v1$Job
↪   => {
  return {
    apiVersion: "batch/v1",
    kind: "Job",
    metadata: {
      name: validateMetadataName(applicationName, true),
    },
  };
};
```

5 TypeScript で Manifest を生成する Generator のアーキテクチャ

5.1 アーキテクチャが解決すること

そもそも Generator そのものが解決することは manifest をドキュメントの乖離を防ぎ、YAML の記法のぶれなどを防ぐことです。アーキテクチャが解決しなければいけないことは、具体的には次のようなことが挙げられます。

1. マニフェスト自体のスケーラビリティを確保する
2. 実際に運用する際に必要最小限の変更だけで Manifest を更新できる ≒宣言的な変更で済むようにする
3. マイクロサービス単位で設定の変更ができる（CPU/MEM/replicas など）
4. 管理しているマイクロサービス全体のリソース量、変更時の増減が把握できる
5. Manifest ファイルの命名規則、出力先のディレクトリ・ファイルツリーなどを意識しなくても良い
6. Generator 自体の保守性を高める

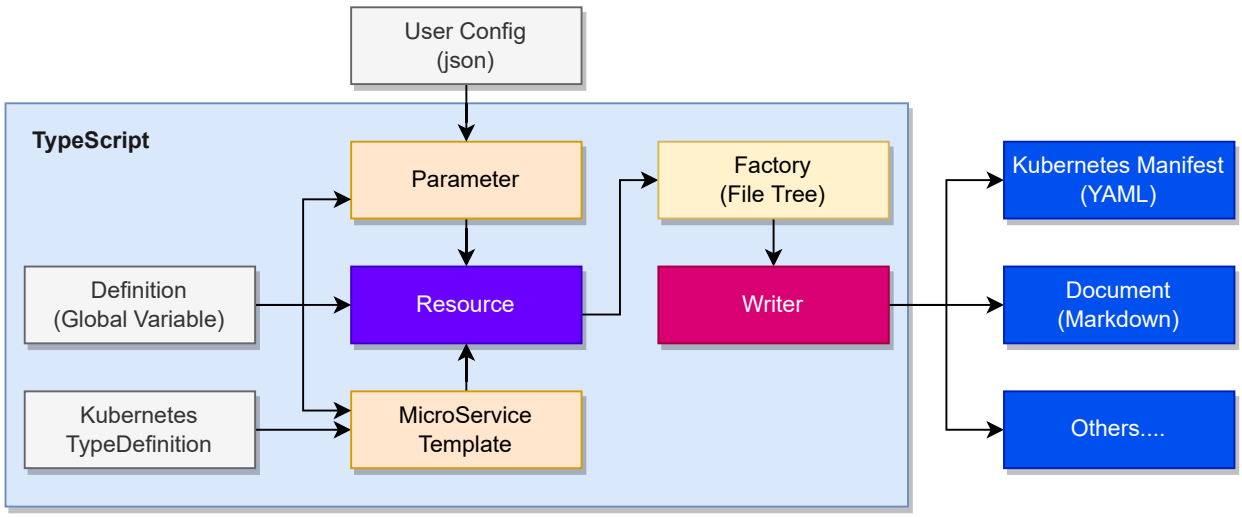


図 5: Manifest 生成のためのアーキテクチャ

これらを表現するためのアーキテクチャは Static Site Generator や Yeoman、Cookiecutter、Rails Scaffold などたくさん事例があります。これらの基本的な骨格を Kubernetes の Manifest Generator として応用し次のようなアーキテクチャが設計しました。

それぞれの役割を紹介します。

名称	役割
User Config	バージョン変更など最小限の変更を与えるファイル
Kubernetes TypeDefinition	TypeScript の型定義
MicroService Template	マイクロサービスの種類に応じたテンプレート
Definition	`Namespace` 名や `Port` 番号、`Gateway` の Host 名などの不動値の定義
Resource	`Parameter` と `MicroService Template` を Kubernetes のリソースコンポーネント単位で結合する
Factory	`Resource` をどのファイル名でどのグループで出力するか定義する
Writer	Factory から与えられた情報から Kubernetes の Manifest や、CPU Requests などのレポートを生成する

5.2 具体的な実装例

実装サンプルを以下のリポジトリに用意しました。`nodejs` と `pnpm` を利用したサンプルとなっています。Docker Swarm を利用すれば Argo Rollouts + Istio がデプロイできるところまで確認しています。

- https://github.com/Himenon/kubernetes-template

Name	PATH
User Config	config/*.json

Name	PATH
Kubernetes TypeDefinition	src/k8s/*
MicroService Template	src/templates/*
Definition	src/definitions/*
Factory	src/factory/*/index.ts
Resource	src/factory/*/resource.ts
Writer	src/writer/*

依存関係は sverweij/dependency-cruiser のカスタムルールによってテストしています。

- https://github.com/Himenon/kubernetes-template/blob/main/.dependency-cruiser.js#L4-L94

Writer が出力するファイルは以下の通り。

- `kubectl apply -k overlays/[env]/`が可能なディレクトリ群
  - https://github.com/Himenon/kubernetes-template/tree/main/overlays
- `production` で利用するリソースのレポート
  - https://github.com/Himenon/kubernetes-template/blob/main/report/resource-table.md

5.3 特徴的なこと

5.3.1 ConfigMap の更新した後に Pod を再起動する

例えば Deployment が ConfigMap の設定によって動作を変化させるような場合、ConfigMap だけを更新してもロールアウトは発生しません。

> Deployment のロールアウトは、Deployment の Pod テンプレート (この場合.spec.template) が変更された場合にのみトリガーされます。例えばテンプレートのラベルもしくはコンテナーイメージが更新された場合です。Deployment のスケールのような更新では、ロールアウトはトリガーされません。
> 引用: Deployment の更新

これを対処するには例えば ConfigMap の ContentHash を計算して、 それを Pod Template の Annotation に付与することで Config Map と Deployment の関係性を作れます。

```
import { createHash } from "crypto";

export const createContentHash = (text: string): string => {
  const hash = createHash("md4");
  hash.update(text);
  return hash.digest("hex");
};
```

```
kind: Deployment
spec:
  template:
```

```
  metadata:
    annotations:
      # Content Hash の値は依存する ConfigMap に対して計算する。
      dependency.config-map.content-hash: bf84e528eaedf3fd7c3c438361627800
```

これの Apply の順番は Argo CD の Sync Wave を利用すると簡単に制御できます。

5.3.2 Report を作成するスクリプトは NonNull Assertion を許可する

TypeScript で書いてると厳密に型定義を守ることが開発の安全に繋がりますが、**Writer** の種類によっては 2 つの理由でこれを許容します。

1. 細かく定義するコストが高い
2. レポートとして自動生成するようなパラメーターは” そもそも必須” であるため、マニフェスト生成時にエラーになってくれたほうが良い

前者は消極的な理由ですが、後者は先程紹介した**実装内で** `Exception` **を発生させる**と同じ意味合いを持っています。つまり、`obj.a?.b?.c` で参照するよりも `obj.a!.b!.c!`で参照すると、型チェックして `throw new Error` をする手間が省ける算段です。もしくは、生成されたレポートがおかしな状態になるのでレビューで簡単に防ぐことができます。

- 実装例

5.4 Manifest 生成はどう使うのがよいか？

5.4.1 リポジトリ運用について

`namespace` 単位で管理するのが楽でしょう。ただし、機密情報がある場合は `secret` だけまとめたリポジトリを別途切るのは必要です。`namespace` 内は基本的に競合する `.metadata.name` を作ることはできません、加えて仮に同じ名前にしても管理が複雑になります。

5.4.2 ツールについて

ここで紹介したのは、愚直に Kubernetes などが提供している OpenAPI Schema から型定義を生成したものを利用した例でした。Kubernetes のドキュメントには REST API 経由で Kubernetes API を Call する Client Library としていくつか紹介されています。

- https://kubernetes.io/docs/reference/using-api/client-libraries/

これを純粋に REST API の Client として使うだけでなく、Manifest を生成するために役立てることも可能でしょう。YAML で書くには複雑になりすぎた場合に、チームで使い慣れた言語で記述する選択肢も用意されているので一考する価値はあるでしょう。

5.4.3 Generator を辞めたくなったら

YAML だけ残して後の実装はさっぱり捨ててしまいましょう。YAML だけあれば Kubernetes にデプロイは可能ですから。

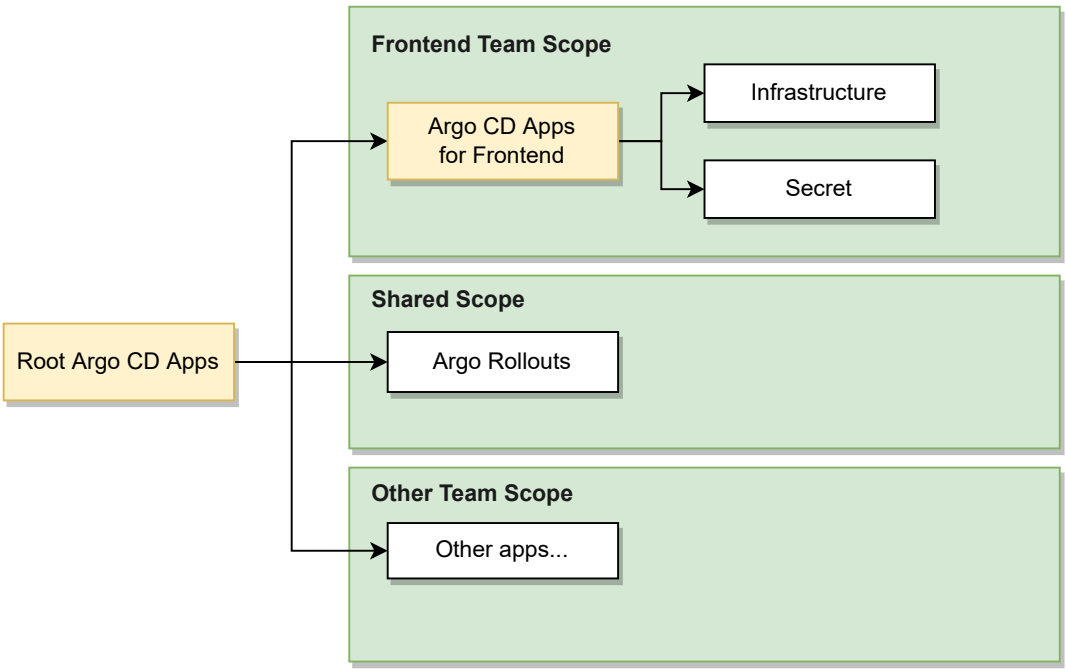


図 6: App of Apps の概略図

6 Argo CD

ニコニコ生放送のフロントエンドでは Continuous Delivery(以降 CD) ツールとして Argo CD と Argo Rollouts を利用しています。ここではその運用と設計について紹介します。

注意書き

- `argoproj.io/v1alpha1/Application` のことを「ArgoCD の Application」と表記します。

6.1 他チームとの棲み分け

Argo CD はフロントエンドのチームだけではなく、他のチームが管理するものも存在しています。したがってチーム横断で管理している部分が存在するとレビューコストが上がるため、App of Apps Patterns を利用して管理する Argocd Application をフロントエンドチームの namespace で分離しました。

具体的には app of apps を 2 段階で利用して次の図ように分離しています。

図中の `Root ArgoCD Apps` は他チームと干渉する部分になっています。ここに、フロントエンドチームが管理する ArgoCD Apps を配置します

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: frontend-apps
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
```

```
  project: default
  source:
    targetRevision: master
    repoURL: # フロントエンドチームが管理する app of apps パターンの親リポジトリ
    path: kubernetes/overlays/[環境名]
  destination:
    server: https://kubernetes.default.svc
    namespace: argocd
  syncPolicy:
    automated: {}
```

これにより、ArgoCD 上で他チームと干渉する場所が木の間にマニフェストファイルが絞り込まれました。

6.2 フロントエンドのチームが管理するマイクロサービスのためのリポジトリと ArgoCD の Application 設定

フロントエンドチームが管理するマイクロサービスの Manifest は 2 つのリポジトリで管理しています。

1. secret
2. infrastructure

secret は Kubernetes の Secret に対応するコンポーネントを格納するリポジトリで、機密情報が含まれるため権限がなければ Read/Write できないリポジトリ設定です。infrastructure のリポジトリはフロントエンドチームが管理するすべてのマイクロサービスの Manifest が集約されています。

infrastructure は具体的には次のようなディレクトリ構成になっています。

```
kubernetes/overlays
├── production
│   ├── app1
│   ├── app2
│   ├── ...
│   ├── ...
│   └── appN
├── [env2]
├── [env3]
├── ...
├── ...
└── [envN]
    ├── app1
    ├── app1
    ├── ...
    ├── ...
    └── appN
```

もう少し平たく書くと、

kubernetes/overlays/[デプロイ環境]/[マイクロサービス名]

したがって、フロントエンドのチームが管理する ArgoCD の app of apps の親は次のような ArgoCD の Application がそれぞれのパスを散所するようにずらっと並んでいます。

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: datadog
  finalizers:
    - resources-finalizer.argocd.argoproj.io
  annotations:
    notifications.argoproj.io/subscribe.slack: "Slack Channel Name"
    argocd.argoproj.io/sync-wave: "-100"
spec:
  project: default
  source:
    repoURL: [REPOSITORY URL]
    targetRevision: master
    # ここのパターンの分だけファイル数がある
    path: kubernetes/overlays/[デプロイ環境]/[マイクロサービス名]
  destination:
    server: https://kubernetes.default.svc
    namespace: frontend
  syncPolicy:
    automated:
```

一見するとファイル数はとても多くなりますが、TypeScript で Manifest を生成しているため、ファイル数の量は問題ではありません。

6.3 フロントエンドが管理するマイクロサービスのための Manifest のリポジトリ

前節ですでに先んじで登場していますが、infrastructure のリポジトリ内にフロントエンドチームが管理するマイクロサービスのすべてがあります。

まず、infrastructure 各マイクロサービスのリポジトリ（NodeJS などの具体的な実装）からは分離されています。これは Best Practice に則っています。

次に、1 つのリポジトリでチームが管理するすべての Manifest が存在している理由は次の点が挙げられます

6.3.1 メニーレポで管理しない理由

1. メニーレポ（複数のリポジトリ）で管理すると保守するときの更新が大変である

6.3.2 モノレポで管理する理由

1. 全体最適化が容易になる
2. Web フロントエンドのアプリケーションは 1 つの host に対してルーティングが存在するため全体を見て調整するケースが有る

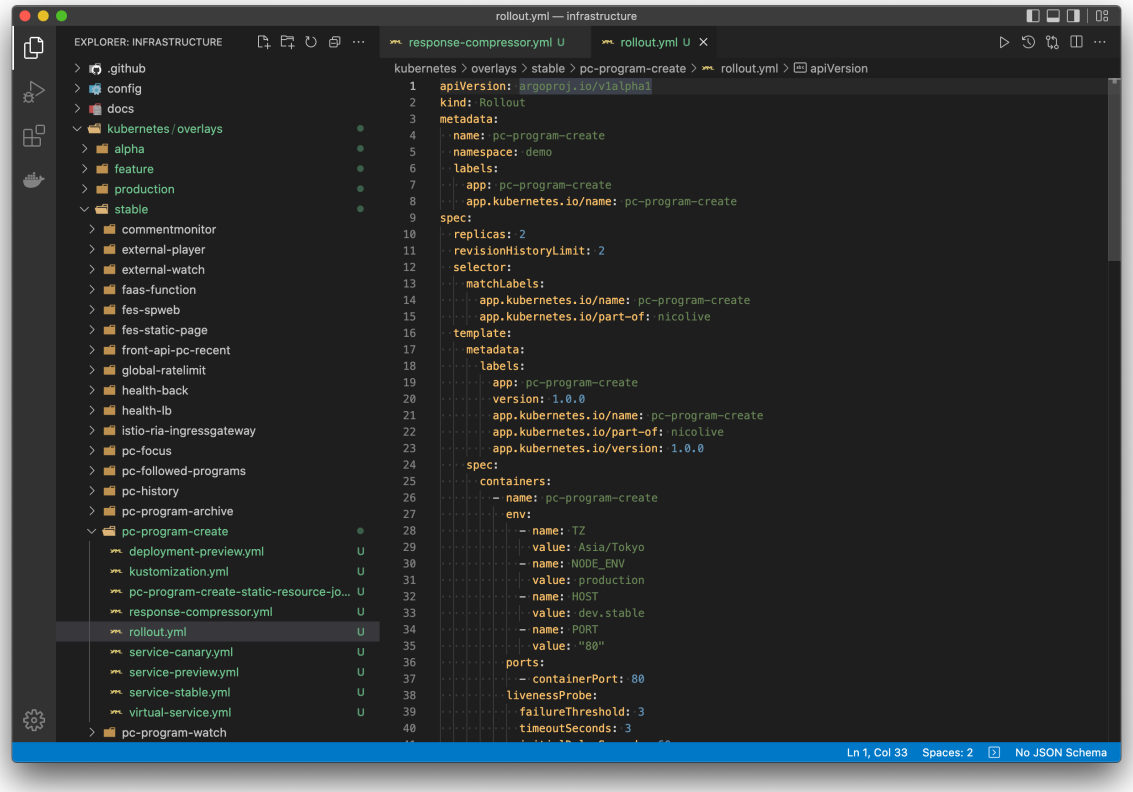


図 7: エディタの画面

- 後述しますが Global RateLimit などがその例

3. 1 つのマイクロサービスに複数のルーティング先が存在するが、デプロイ単位として分割したい場合の管理

これらは Web フロントエンドの開発時に npm パッケージをモノレポで管理しているところからの発想もあり、モノレポのほうが開発や保守効率が圧倒的に早いことが経験則としてわかっていることも決め手の背景としてあります。

Slack Bot による自動化で改めて紹介しますが、結果的にモノレポで管理した事によって Bot による更新が容易になり、マニフェストの変更から本番デプロイまでが 5 分以内で終わるスピーディなリポジトリなっています。もはや Bot ユーザーしかリポジトリの更新をしていない状態です。

6.4 `infrastructure` リポジトリのブランチ設計

マイクロサービスを Kubernetes クラスターにデプロイするためのブランチは以下の 2 つしか用意していません。

`master` .... **開発環境にリリースされるブランチ（`Default` ブランチ）**
`release/production` .... **本番リリース用のブランチ**

`kustomize` が提案するブランチ運用の場合、各環境ごとにデプロイするブランチが存在しますが、開発環境においては Default ブランチにマージした場合はすべての開発環境に問答無用でデプロイされるようにしています。

理由はいたって単純で、デプロイする環境数が多く、わざわざ各環境用にデプロイするための Pull Request 作成からマージまでのリードタイムは非常に遅いと判断したためです。

6.4.1 master ブランチは Squash Merge のみ許容する

モノレポにしていることもありたくさんの Pull Request が `infrastructure` リポジトリに飛んできます。すると commit 履歴も比例して多くなります。 通常 Merge Commit で GitHub の Pull Request を処理した場合、Pull Request 中で変更した内容と `Merge pull request` のコミットが一気に追加されるため、GitOps を運用しているリポジトリでこれを実施すると commit 履歴が単純に荒れます。これを防ぐため、master ブランチに対するマージは Squash Merge のみを許可し、必ず Pull Request 一つに対して commit が 1 つになるように実施しています。

また、`release/production` ブランチに関してはリリース用のブランチであるため、こちらは `Merge Commit` のみを許容しています。Squash Merge を実施した場合は `master` に対してバックポート処理が必要になるためです。さらに、2 つのブランチで異なる Merge 方法を GitHu のルールで強制することができないため、Slack Bot によるマージ処理の自動化も実施しています。

6.4.2 master ブランチに入ったものは必ずいつでもリリースして良いものとして扱う

manifest をモノレポで運用しているため、共生している他のマイクロサービスが別のマイクロサービスと同時にリリースされる可能性があります。すなわち、本番環境にリリースしたくない変更は `master` ブランチに入れなければ良いだけになります。また、特定の環境だけバージョンを上げたいときの手続きが長いといった要望は明らかに予想可能な問題なため、同時に Slack Bot によってデプロイの簡略化と自動化を実現しています。

6.4.3 `release/production` ブランチに対して Pull Request を投げたとき同時に tag とリリースノートを作成する

`release/production` に対する Pull Request は不可逆の処理として扱います。リリースすべきでない変更が入っている場合は `master` にコミットした後、改めて `release/production` に対して Pull Request を投げます。 このときも tag とリリースノートを同時に作成します。すでに切られた tag やリリースノートは削除せずどんどん新しいものを使っていく運用になっています。tag やリリースノートを削除して新しく新規のバージョンで切る方法もありますが、これはたくさんの変更を受け付ける際にコンフリクトしやすく、運用が難しいため、運用が簡単で速度が出やすい欠番方式を採用しています。

6.5 デプロイの速度が重要な理由

ここまで紹介してきた方法はすべて「デプロイの速度を落とさないため」に実施しています。速度に拘る理由は、「遅くする理由がないから」です。Kubernetes 上で障害が発生したときは一時的に CLI や管理用の Dashboard から Rollback の処理などを実施することができます。しかしながらそれだけでは対応できない構成変更やアプリケーションの再投入が必要な場合に、デプロイの部分が遅ければそれだけ影響の受けるユーザーが多く、損失も大きくなります。逆コンウェイの法則然り、最速でデプロイできるフローに運用の手続きをあわせていくことがこの場合、あらゆる面で有効であるため、最速のデプロイフローを作ることに拘っています。こう見ると安全にデプロイできるかという話がありますが、それは自動化によって対処すべき話であるため別途紹介します。

7 Argo Rollouts の導入

7.1 Argo Rollouts とは

Argo Rollouts は Kubernetes 上に Pod をデプロイする方法の選択肢を増やしてくれます。Blue/Green デプロイ、Canary デプロイ、Progressive Delivery など。

とくに Traffic Management を利用したデプロイ方法は非常に魅力的で、利用しない理由は見当たりませんでした。ちょうど Argo Rollouts が v1 系が利用可能な状態で、移行時の検証と同時に必要な機能が使えることを確認できたため導入しました。

- 2021/05 v1.0.0
- 2021/10 v1.1.0
- 2022/03 v1.2.0

7.2 Istio + Argo Rollouts

Istio 自体はすでに利用可能な状態にあったため、Traffic Management を実施する LoadBalancer は Istio を利用しています。

- Istio - Traffic Management

他と比較はできていませんが、Istio で Traffic Management をすると、Istio の Service Mesh の恩恵をそのまま得られることができ Canary デプロイ時に Traffic Weight が変化していることが観測できるようになります。なお、Istio Ingress Gateway の設定でその他の機能についても紹介しています。

7.3 Canary Deploy を実施する

Rollout の Manifest は`.spec.strategy`以外の部分は Deployment と同じです。

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: my-app
spec:
  strategy:
    canary:
      trafficRouting:
        istio:
          virtualService:
            name: my-app
            routes:
              - http-my-app-primary
      maxSurge: 33%
      canaryService: my-app-canary
      stableService: my-app-stable
```

```
      dynamicStableScale: true
      steps:
        - setCanaryScale:
            matchTrafficWeight: true
        - pause:
            duration: 60
        - setWeight: 34
        - setCanaryScale:
            matchTrafficWeight: true
        - pause:
            duration: 60
        - setWeight: 67
        - setCanaryScale:
            matchTrafficWeight: true
        - pause:
            duration: 60
        - setWeight: 100
```

特徴的なのは `canaryService` と `stableService` 用に 2 つの `Service` 定義が必要になるところです。Rollouts に定義された Service は

```
.spec.selector.rollouts-pod-template-hash: "HashValue"
```

が Argo Rollouts によって付与されます。また Pod の更新時に ReplicaSet にも

```
.spec.selector.matchLabels.rollouts-pod-template-hash: "HashValue"
```

が指定され、Canary 用の Service と Stable 用の Service のエンドポイントが区別されています。あとは Istio の Virtual Service の Traffic Weight に対する変更が Step 単位で記述することが可能です。

7.3.1 注意点

Argo Rollouts は Dashboard では `Promote Full` というボタンがあります。これを利用した場合、step を無視して Promotion の最終状態まで到達します。つまり、トラフィックを受け付ける準備ができていない Pod に対してもトラフィックが流れるため、使う場面を見極める必要があります。

- Argo Rollouts v1.2.0 で確認

7.3.2 Traffic Weight に応じた Replicas を指定する

```
.spec.strategy.canary.steps[].setCanaryScale.matchTrafficWeight = true
```

を指定することで、`.spec.replicas` を 100% ととした Replicas が Traffic Weight に比例して Pod が配置されます。

また、`setCanaryScale.replicas` と併用して指定している場合は、Canary の replicas は Rollouts の manifest で指定した値に必ずしもならず、Traffic Weight で算出された Canary の Replicas と手動で指定した Replicas の Traffic に耐えられるうち安全な方が優先されて指定されます。

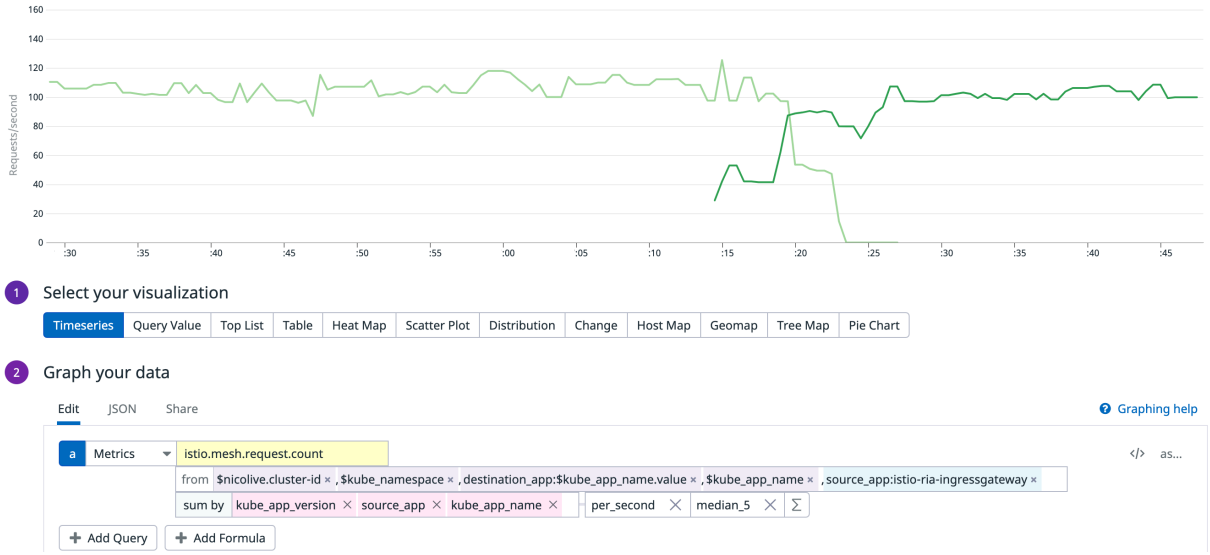


図 8: 更新時の Request Per Seconds の変化

- https://github.com/argoproj/argo-rollouts/blob/v1.2.0/utils/replicaset/canary.go#L331-L353

7.3.3 Canary Deploy 時トラフィックが流れていない Pod を縮退させる

以下のフラグを有効にすることで実現できます。

```
.spec.strategy.canary.dynamicStableScale = true
```

これにより、replicas と Traffic Weight を同時にコントロール可能な状態になります。

7.4 DataDog でのモニタリング例

あるマイクロサービスの Pod に対する Request Per Sec をバージョンで分類して、Rollouts による更新をモニタリングすると次のように Traffic が変更されていることがわかります。

同様に CPU の使用率も確認すると、たしかに Rollouts で定義した Step 通りに Pod は増加し、Traffic が流れなくなった Pod は徐々に終了していることが確認できます。

7.5 これから

Argo Rollouts は **Progressive Delivery** を実現する方法を提供しており、DataDog との連携も容易にできることがわかっています。

- https://argoproj.github.io/argo-rollouts/analysis/datadog/

これを実現するために、今現在は各種 Metrics の集計とその信頼性の検証を進めています。BFF のマイクロサービスの安定性を表すための定量的な指標を集計値として表せてはじめてこの機能を有効にできるため、運用の実績値を蓄積し、集計し、不足している Metrics を追加する作業を繰り返し行っています。

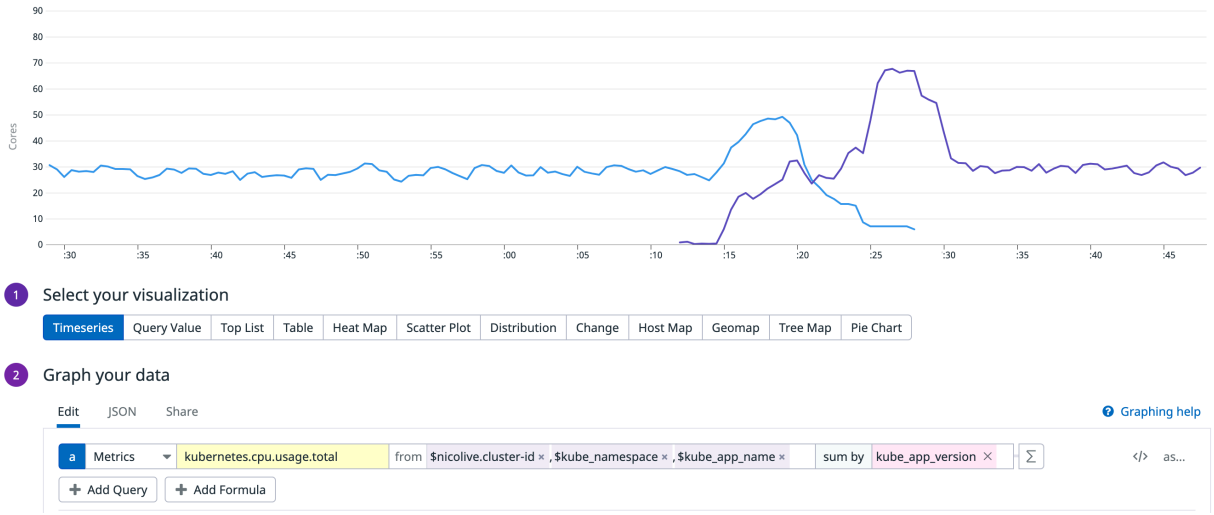


図 9: 更新時の CPU 使用率の変化

8 Slack Bot による自動化

Argo CD による GitOps の実現は同時に GitOps を開発者に強制します。すなわち、バージョンアップのための commit を実施し、Pull Request を投げ、マージする必要があります。更新頻度の多いアプリケーションを抱えた場合、この作業が非常に長く開発者の体験を悪くします。

そこで、Slack Bot サーバーを作成し Slack にメッセージを入力することで手続き的なタスクをサーバー側に実施するようにしました。

8.1 バージョンアップのシーケンス図

シーケンス図を使って紹介します。バージョンアップの手順は Slack 上で Bot に対して次のようなコマンドを投げることから始まります。

```
# server-a をバージョン 2.0.0 に変更する
@bot update version:2.0.0 app:servive-a
```

これを受け取った bot サーバーは、メッセージの入力者を判別したり、コマンド (`update version`) をパースしたりします。コマンドに応じて GitHub API を Call し、JSON で記述されたファイル (User Config) を書き換え、commit します。 その後 Pull Request を作成して、結果をユーザーに返します。

作成された Pull Request をさらに Slack からマージします。

```
@bot merge pr:123
```

これをシーケンス図で書き起こすと次のようになります。

基本的な操作はすべて Slack 上から実施が可能で、開発者がバージョンアップのためにリポジトリを Clone して環境構築する必要はありません。

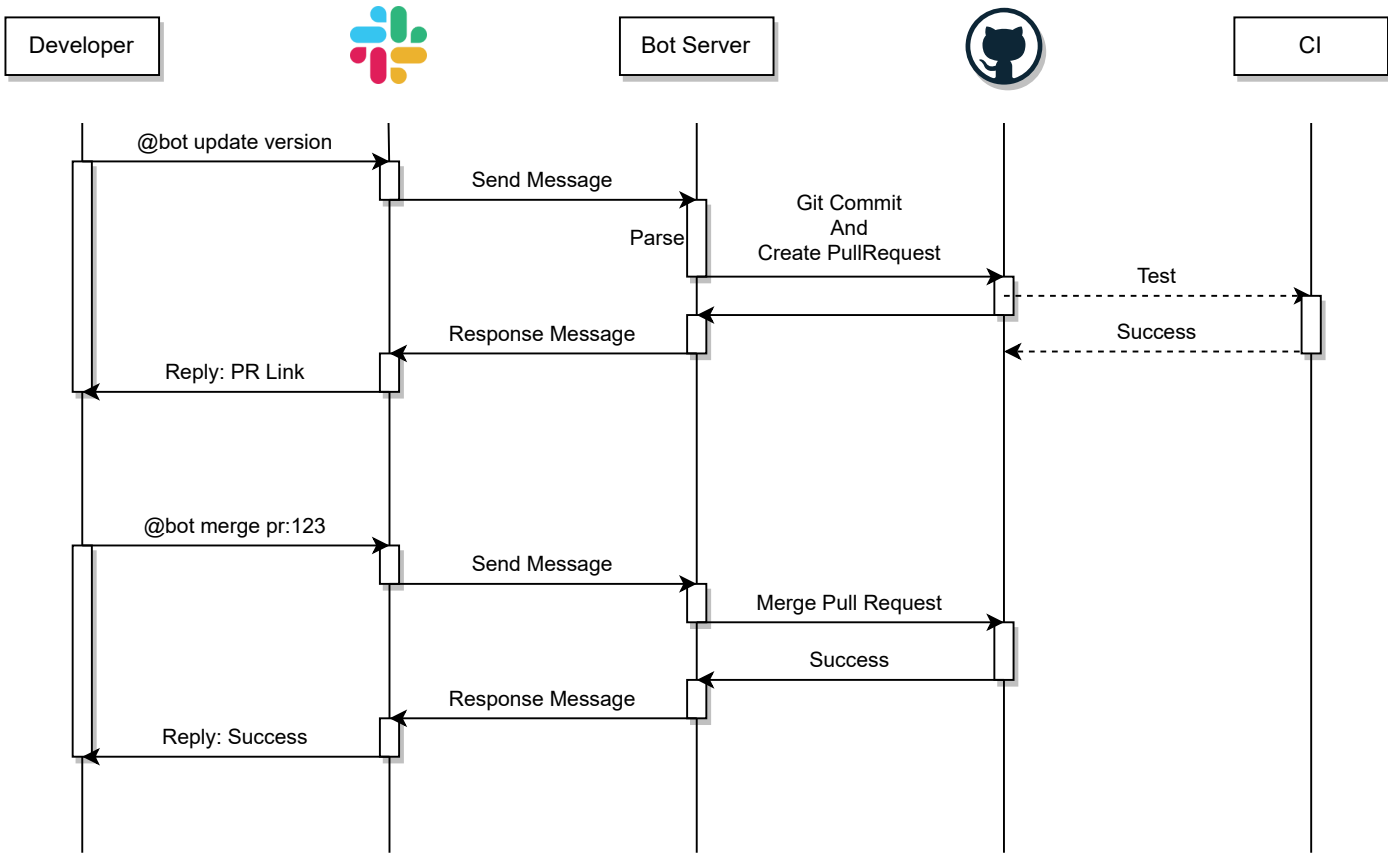


図 10: ユーザーが Slack でメッセージを入力した際のシーケンス

8.2 既存のアプリケーションの CI と Kubernetes Manifest のリポジトリの連携

Slack Bot によって自動化された Kubernetes の Manifest リポジトリは既存のリリースフローとも結合が容易になります。

例えば、アプリケーションにバージョンアップの CI タスクがあった場合、次のバージョン情報を Slack の Webhook を利用して先程と同じようにメッセージを Bot に対して送るだけで結合できます。

```
# 擬似コード
message="{\"text\":\"@bot update version:${nextVersion} app:service-a\"}"
curl -X POST -H 'Content-type: application/json' --data $message
↪  https://hooks.slack.com/services/{your_id}
```

大抵のサーバーは `curl` かそれに類する HTTP Client を用意できるため、たった 2 行挿入するだけでデプロイの簡略化ができます。

8.3 Slack Bot によってデプロイ作業を最小工数で終わらせる

バージョンアップのコマンドを紹介しましたが、他にも 10 個程度のコマンドがあります。

- リリース準備用のコマンド
- 最新のリリース情報の取得

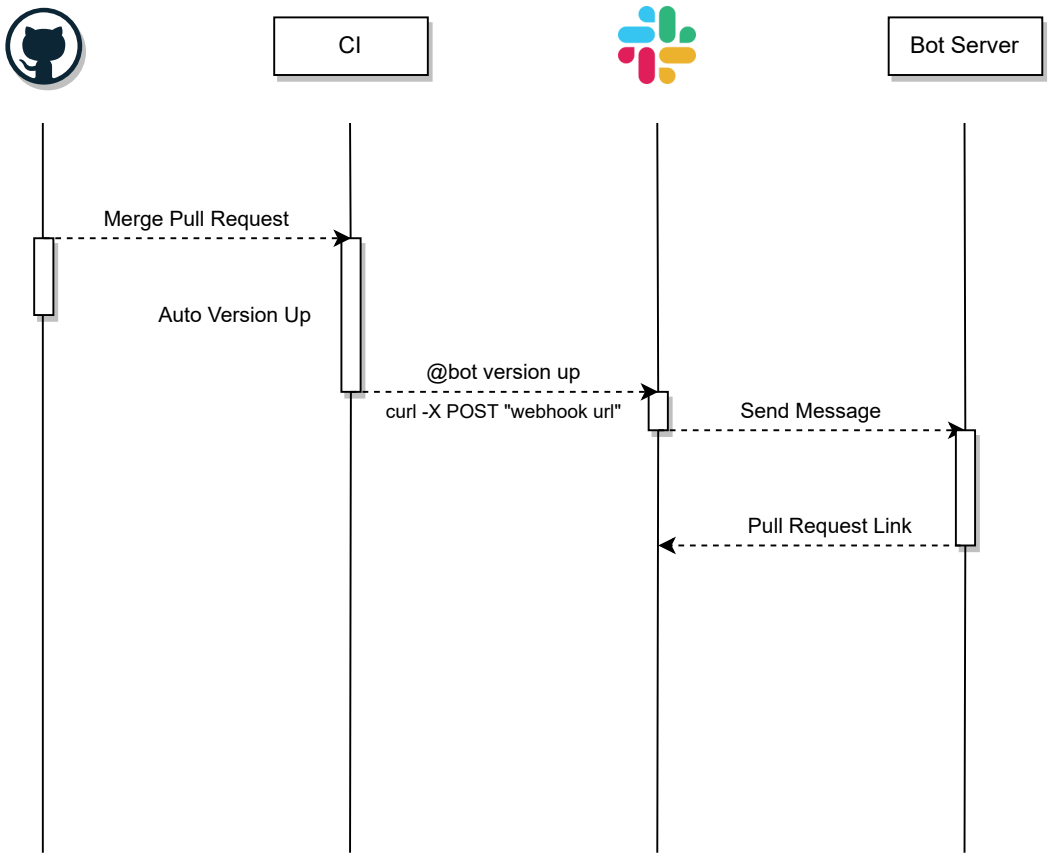


図 11: アプリケーションの CI と Kubernetes の Manifest リポジトリの連携

- 次に投入される予定のバージョン情報の取得（差分）
- リリース用のチケット作成
- リリースノート更新

など、リリースに関する一連の情報や作業が細かくできるようになっています。特に、差分情報やリリースノートの作成などを自動で実施しているためリリースの影響範囲が単純明快になるため確認コストが最小限になっています。

また、Slack のメッセージ経由で実施しているため Bot サーバーが失敗した場合でも何がやりたかったのか証跡が Slack 上に残ります。再実行を実施するのももう一度メッセージをコピー&ペーストするだけの作業になります。

9 BFF と Istio

9.1 Istio の利用

Istio は既存のマイクロサービスに対して後付で導入することができ、通信を可観測にしたり、負荷分散を実施したり、Proxy としての機能を持っています。Kubernetes 上で稼働するマイクロサービスの通信をよりプログラマブルに扱える機能を提供しています。

実際に触ってみると istio が謳っているこれらの機能は有用で、サービスメッシュは Kubernetes を運用する上で必要不可欠であることを実感させられます。

さて、詳細な部分はドキュメントを読むのが望ましいですが、とっつきにくい部分もあるのでフロントエン

ドのエンジニアが使うと便利な機能を紹介しつつ Istio のコンポーネント紹介します。

9.2 Istio と Envoy の関係

まずは Istio と Envoy の関係について知っておく必要があります。

Envoy はそれ自体が Proxy であり、nginx や Apache などの L7 LB と似たような機能を提供しています。大きな違いとして、Envoy はテレメトリが標準で豊富だったり、API による構成変更が可能だったりプログラマブルにコントロールできる機能を豊富に持っています。すなわち再起動をせずに構成変更が容易であり、Argo Rollouts の Canary Deploy で紹介したように Traffic Weight を柔軟に変更することが可能になります。

Istio はこの Envoy を利用して、Kubernetes 上で稼働するマイクロサービス間の通信を観測するために Control Plane から各 Pod に Sidecar として注入します。Istio から提供されている Envoy の Docker Image は `istio-proxy` という名前で提供されており、`kubectl get pod [podname]` などで構成を確認すると `istio-proxy` という名前を確認することができます。

Envoy 単体では通常以下のような YAML を記述して起動時に読み込ませることで Envoy の設定変更を実施します。

- https://www.envoyproxy.io/docs/envoy/latest/configuration/overview/examples

Istio の場合、Envoy はすでに起動された状態で存在しているため、既存の設定が存在しています。そのため、この構成変更をしたい場合は `EnvoyFilter` を利用します。

- https://istio.io/latest/docs/reference/config/networking/envoy-filter/

ただ普段書くような Traffic Management 用の設定は別のコンポーネントを利用して簡易に記述することができます。

Component 名	役割
Gateway	受け入れ可能なホスト名、ポート番号、プロトコルなどを記述する
Virtual Service	`PATH` 単位のルーティングの設定が可能。Traffic Weight の指定、Header や Query Parameter による個別のルーティング先もここで指定する。
Service Entry	Kubernetes クラスタから外部へのアクセス制限など。

その他（`https://istio.io/latest/docs/reference/config/networking`）にも Component はありますが最初に指定するものはおおよそ上記の 3 つでしょう。

9.3 Ingress Gateway をセットアップする

Istio は Sidecar に istio-proxy を注入するだけではなく、Ingress Gateway を作成することでその機能をより活かすことができます。

IngressGateway は `namespace` や Kubernetes の境界に位置する Gateway として機能させることで管理下にあるマイクロサービスに対するアクセスの制御ができます。Web フロントエンドが管理するような、Internet からアクセスされ、他マイクロサービスから直接 CALL が必要ないマイクロサービスは Ingress Gateway を通して管理すると負荷対策やデプロイの運用が楽になります。

Istio における Ingress Gateway のセットアップは `IstioOperator` 利用して実施します。

- https://istio.io/latest/docs/setup/install/operator/

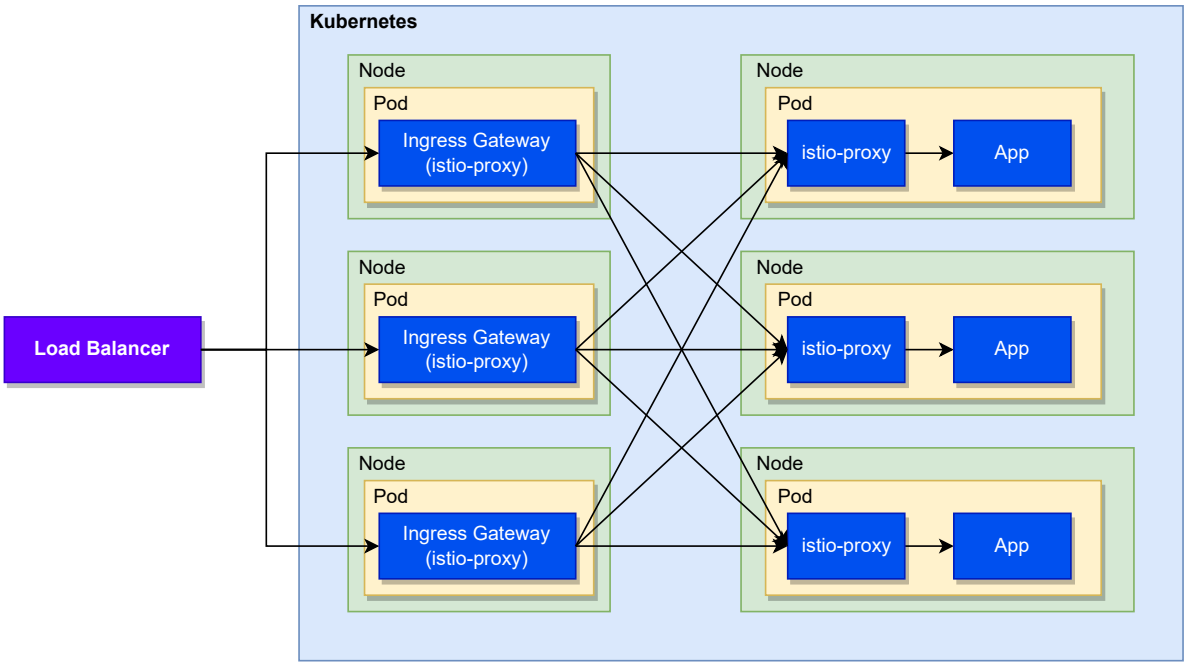


図 12: istio ingress gateway の概略図

9.3.1 istio-system 以外の namespace で IstioOperator が利用できるようにする

IstioOperator の管理を namespace を分けて管理したい場合、デフォルトの設定のままではインストールすることができません。例えば以下のように IstioOperator(`Deployment`) をセットアップすると `watchedNamespaces` で指定された `istio-system` でのみ `IstioOperator` のコンポーネントが利用できません。

```
$ helm install istio-operator manifests/charts/istio-operator \
  --set watchedNamespaces=istio-system \
  -n istio-operator \
  --set revision=1-9-0
```

hedNamespaces=istio-system
-n istio-operator
–set revision=1-9-0 \end{minted}

すでにインストールされた環境下の場合、次のようなコマンドで IstioOperator が利用可能な namespace を確認することができます。

```
kubectl get deployment -n istio-operator -l operator.istio.io/component=IstioOperator
↪  -o yaml | grep -A1 "name: WATCH_NAMESPACE"
```

環境変数 `WATCH_NAMESPACE` を更新することで `IstioOperator` のコンポーネントが利用することができるようになります。例えば `myteam` という `namespace` を追加したい場合は次のように実施します。

```
kubectl set env deployment/istio-operator-1-11-4
↪  WATCH_NAMESPACE="istio-system,myteam" -n istio-operator
```

Ingress Gateway の具体的な設定は次の節で紹介しています。

9.4 istio-proxy のサイドカーが不要なケース

9.4.1 Job

Istio を有効にした場合 Pod に対して istio-proxy が sidecar として挿入されます。しかしながら、不要なケースも存在します。1 回だけ実行される Job として実行される Pod は Job が終了したすると Pod の Status が Completed になりますが、istio-proxy は常駐するサーバーであるため Job が Completed になりません。

そのため、Job は Pod Template に対して `sidecar.istio.io/inject: "false"`を指定することで Sidecar を注入させないようにしています。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: myjob
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "false"
  # 省略
```

9.5 BFF でサービスメッシュが有効だと何が良いか

最も嬉しいのは可観測性にあります。BFF はその特性上、各マイクロサービスから情報をかき集め、場合によっては Server Side Rendering(SSR) を実施します。最終的な結果はユーザーに届くため、一連の処理がユーザー体験に直接影響します。ゆえに、明らかにレスポンスタイムが悪いマイクロサービスがある場合いくつかの行動を取ることができます。特定のバージョンから悪化しているのであればロールバックを実行したり、Client Side Rendering 可能な情報であれば最初の HTML を構成するためのクリティカルパスから除外したりすることが可能です。少なくとも、継続的な監視は問題を明確にし、物事の優先度を合理的に決定することができます。負荷試験やモニタリングの節で具体的な Metrics の可視化を紹介しています。

10 アクセスログ

Web フロントエンドが管理するサーバーにおける最重要なシステムの一つは**アクセスログ**です。不正アクセスなどのセキュリティ的な側面や、会社の収益のツリー構造に関わる部分など多くの重要な情報をここから得られます。ゆえにこの部分のシステムは信頼度が最も高い方法で実現する必要があります。

したがって移行前のアーキテクチャをなるべく踏襲しつつ、Ingress Gateway に近いところに配置する必要がありました。また、ログは既存の fluentd の収集と連携する必要がありました。

最終的に本番で稼働しているアーキテクチャは次のようになります。

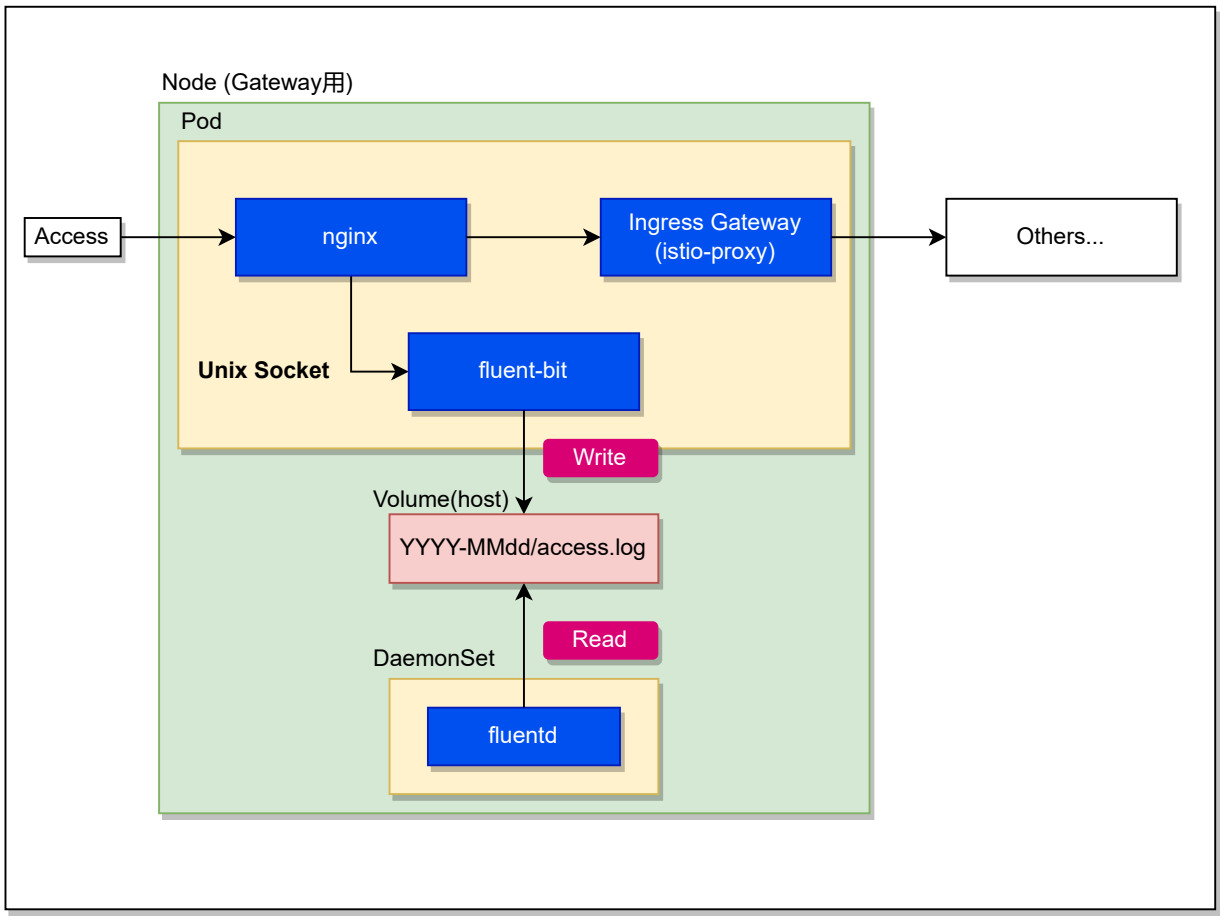


図 13: アクセスログの収集アーキテクチャ

10.1 Ingress Gateway とアクセスログ周りのアーキテクチャ

戦略としては Ingress Gateway の前段に nginx を配置し、クラスター外からのアクセスを最初に nginx が受けるようにしました。nginx から出力されるアクセスログは `syslog` で Unix Socket を経由して fluent-bit に転送しています。fluent-bit は syslog を INPUT として既存の fluentd と結合するために出力先のディレクトリとログの書き出しをコントロールしています。

このアーキテクチャに至った経緯を紹介します。

10.1.1 アクセスログの出力に nginx を利用している理由

今回は移行が伴っているため、なるべく低コストで移行を安全に実施したい狙いがありました。もともと nginx からログを出力していることもあり、その実績からそのまま流用する形を取りました。

また、envoy によるアクセスログの出力も考慮に入れましたが、Cookie などに含まれる情報を出力するために lua を書く必要があったり、そのパース用にスクリプト自体が保守するのが大変であるため断念しました。

10.1.2 fluent-bit で収集して fluentd に渡している理由

fluentd は移行前からあるログ収集の手段です。fluent-bit は fluentd の C 言語実装で、fluent-bit も出力先を fluentd と同じ場所に向けることは可能です。しかしながらこれも移行をスムーズに進めるために既存の fluentd の設定を頑張って fluent-bit に移すことはしませんでした。

10.1.3 nginx から fluent-bit に Unix Socket 経由でログを送信している理由

最初、fluent-bit を DaemonSet として配置して Ingress Gateway 用の Node に配置するようにしていました。nginx のログを `stdout` で出力し、`/var/log/containers/[containerId].log` に出力される nginx のログを fluent-bit の tail INPUT を利用して収集していました。

しかしながら、高 rps 環境下で tail を利用すると fluent-bit の tail が突然止まる不具合に遭遇しました。これは issue に起票されていますが、活発でないとして Bot によって 2022/04/09 クローズされました。

- https://github.com/fluent/fluent-bit/issues/3947

挙動を見ているとどうやら`/var/log/containers` に出力されるログファイルのシンボリックリンク先である、`/var/log/pods/[pod-id]/0.log` が`.gz` ファイルにアーカイブされるときにファイルディスクリプタあたりが変更されそこでうまく fluent-bit が処理で基底なさそうだということがなんとなくわかっています。とはいえこれを修正するために fluent-bit に Pull Request を送って、リリースされるまでの間ログが収集できないとなると移行スケジュールに問題が発生するため別の方法を考えました。

幸い、AWS の fluent-bit のトラブルシューティングがあったのでここを参考にしました。

- https://github.com/aws/aws-for-fluent-bit/blob/mainline/troubleshooting/debugging.md

Scaling の章に高スループットで fluent-bit を運用するための方法が紹介されており、そこに「DaemonSet モデルから Sidecar モデルへ」と「ログファイルの Tail から Log Streaming モデルへ」の変更が有効であることが記述されていました。

すぐにこれを採用し、最初に紹介したアーキテクチャへと変貌を遂げました

10.2 具体的な設定

これら理由を踏まえた上で設定は次のようになります。

10.2.1 nginx の出力先の設定

ログは取り扱いしやすいように一度 JSON で出力しています。syslog は`/tmp/sidecar-nginx/sys-log.sock` に対して出力しています。

```
log_format json_access_format escape=json '{ 中略 }'
server {
  access_log syslog:server=unix:/tmp/sidecar-nginx/sys-log.sock json_access_format;
}
```

10.2.2 fluent-bit

`INPUT` は`/tmp/sidecar-nginx/sys-log.sock` から nginx のログを JSON 形式で読み込み、syslog → JSON → 日付抽出 →タグの書き換え (出力先の調整)`FILTER` を通った後、ファイルに書き出しています。

```
[SERVICE]
    Flush               1
    Grace               120
    Daemon              off
    Parsers_File        parsers.conf
```

```
    HTTP_Server         On
    HTTP_Listen         0.0.0.0
    HTTP_PORT           2020
    Log_Level           info

[INPUT]
    Name                syslog
    Tag                 kube.*
    Path                /tmp/sidecar-nginx/sys-log.sock
    Parser              syslog-rfc3164-local
    Mode                unix_udp

[FILTER]
    Name                parser
    Match               kube.*
    Key_Name            message
    Preserve_Key        true
    Reserve_Data        true
    Parser              json

[FILTER]
    Name                lua
    Match               kube.*
    script              create-log-file-path.lua
    call                create_log_file_path

[FILTER]
    Name                rewrite_tag
    Match               kube.*
    Rule                vhost ^(.*)$ /log/output/path/$log_file_path true

[PARSER]
    Name                nginx_access_log
    Format              regex
    Regex               ^(?<container_log_time>[^ ]+) (?<stream>stdout|stderr)
↪   (?<logtag>[^ ]*) (?<message>.*)$
    Time_Key            time
    Time_Format         %Y-%m-%dT%H:%M:%S%z
    Time_Keep           On

[PARSER]
    Name                syslog-rfc3164-local
    Format              regex
```

```
    Regex               ^<(?<pri>[0-9]+)>(?<time>[^ ]* {1,2}[^ ]* [^ ]*)
↪   (?<ident>[a-zA-Z0-9_/.-]*)(?:[(?<pid>[0-9]+)])?(?:[^:]*:)? *(?<message>.*)$
    Time_Key            time
    Time_Format         %b %d %H:%M:%S
    Time_Keep           On

[PARSER]
    Name                json
    Format              json

[OUTPUT]
    Name                file
    Match               *
    Format              template
    Template            method:{method}  uri:{uri} 中略
```

日付順で出力するために、以下の lua script を利用して Tag を書き換えています。

```
-- create-log-file-path.lua
function create_log_file_path(tag, timestamp, record)
  new_record = record
  new_record["log_file_path"] =
  ↪  os.date("%Y-%m%d",timestamp).."/istio-ingressgateway-access.log"
  return 1, timestamp, new_record
end
```

10.2.3 IstioOperator

Kubernetes の Manifest ファイルは次のようになります

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio-my-ingressgateway
spec:
  profile: empty
  components:
    ingressGateways:
      - name: istio-my-ingressgateway
        enabled: true
        k8s:
          overlays:
            - apiVersion: apps/v1
              kind: Deployment
```

```
    name: istio-my-ingressgateway
    patches:
      - path: spec.template.spec.containers[1]
        value:
          name: sidecar-nginx
          env:
            - name: TZ
              value: Asia/Tokyo
          image: # nginx
          securityContext:
            privileged: true
            runAsUser: 0
            runAsGroup: 0
            runAsNonRoot: false
          volumeMounts:
            - name: cache-volume
              mountPath: /var/cache/nginx
            - name: pid-volume
              mountPath: /var/run
            - name: ingressgateway-sidecar-nginx-conf
              mountPath: /etc/nginx
              readOnly: true
            - name: nginx-unix-socket
              mountPath: /tmp/sidecar-nginx # nginxのsyslog出力場所
      - path: spec.template.spec.containers[2]
        value:
          name: sidecar-fluent-bit
          image: fluent/fluent-bit:1.8.13
          imagePullPolicy: Always
          ports:
            - containerPort: 2020
          securityContext:
            privileged: true
            runAsUser: 0
            runAsGroup: 0
            runAsNonRoot: false
          readinessProbe:
            httpGet:
              path: /api/v1/metrics/prometheus
              port: 2020
            failureThreshold: 3
            timeoutSeconds: 3
          livenessProbe:
```

```
        httpGet:
          path: /
          port: 2020
        failureThreshold: 3
        timeoutSeconds: 3
      resources:
        requests:
          cpu: 150m
          memory: 128Mi
        limits:
          cpu: 150m
          memory: 128Mi
      volumeMounts:
        - name: sidecar-fluent-bit
          mountPath: /fluent-bit/etc
        - name: log-output-volume
          mountPath: /log/output/path # fluent-bit のログの出力場所
        - name: nginx-unix-socket
          # fluent-bit が fluent-bit の UNIX Socket を読み込む場所
          mountPath: /tmp/sidecar-nginx
- path: spec.template.spec.volumes[8]
  value:
    name: ingressgateway-sidecar-nginx-conf
    configMap:
      name: ingressgateway-sidecar-nginx-conf
      items:
        - key: nginx.conf
          path: nginx.conf
- path: spec.template.spec.volumes[9]
  value:
    name: sidecar-nginx-error-page
    configMap:
      name: sidecar-nginx-error-page
- path: spec.template.spec.volumes[10]
  value:
    name: cache-volume
    emptyDir: {}
- path: spec.template.spec.volumes[11]
  value:
    name: pid-volume
    emptyDir: {}
- path: spec.template.spec.volumes[12]
  value:
```

```
      name: varlog
      hostPath:
        path: /var/log
  - path: spec.template.spec.volumes[13]
    value:
      name: sidecar-fluent-bit
      configMap:
        name: sidecar-fluent-bit
  - path: spec.template.spec.volumes[14]
    value:
      name: log-output-volume
      hostPath:
        path:  /log/output/path
  - path: spec.template.spec.volumes[15]
    value:
      name: nginx-unix-socket
      emptyDir: {}
```

`/tmp/sidecar-nginx` に対して Unix Socket 用の Empty Directory を作成し、Pod 内でシェアすることで Pod として見たときにポータビリティが確保できます。

IstioOperator で新しく Container や Volume を追加する場合は現状 `k8s.overlays` で頑張って追加するしかありませんが、Manifest を TypeScript で管理しているため、管理が難しいなどの問題は発生しませんでした。

ただしバージョンアップに伴って IstioOperator が作成する IngressGateway の Deployment を確認する必要があります。早々バージョン更新の頻度は高くないので、バージョン更新後の検証と同時にやっても問題ないでしょう。

10.3 これから

ここまでに説明したことを改めて整理すると、移行時に飲み込んだ冗長的な部分を最適化することがまずできます。

1. nginx のログ出力を Ingress Gateway の istio-proxy で実施する
2. fluentd によるログ転送処理を fluent-bit に移行する

これらを実施することで `IstioOperator` の Manifest はある程度見やすくなります。

11 Istio Ingress Gateway の設定

Ingress Gateway クラスター外部に対してクラスター内部の Service に対するルーティングを公開します (Ingress とは何か)。Istio も Ingress Gateway を提供しており、L7 のルーティング設定を記述することができます。

Istio Ingress Gateway の設定を変更するためにはいくつかの Component を定義する必要があり、代表的なのはドキュメント (Istio / Ingress Gateways) で紹介されている `Gateway` と `VirtualService` になります。nginx や Apache のように conf ファイルを起動時に読み込む形式と違い、istio が Envoy に対して API 経由

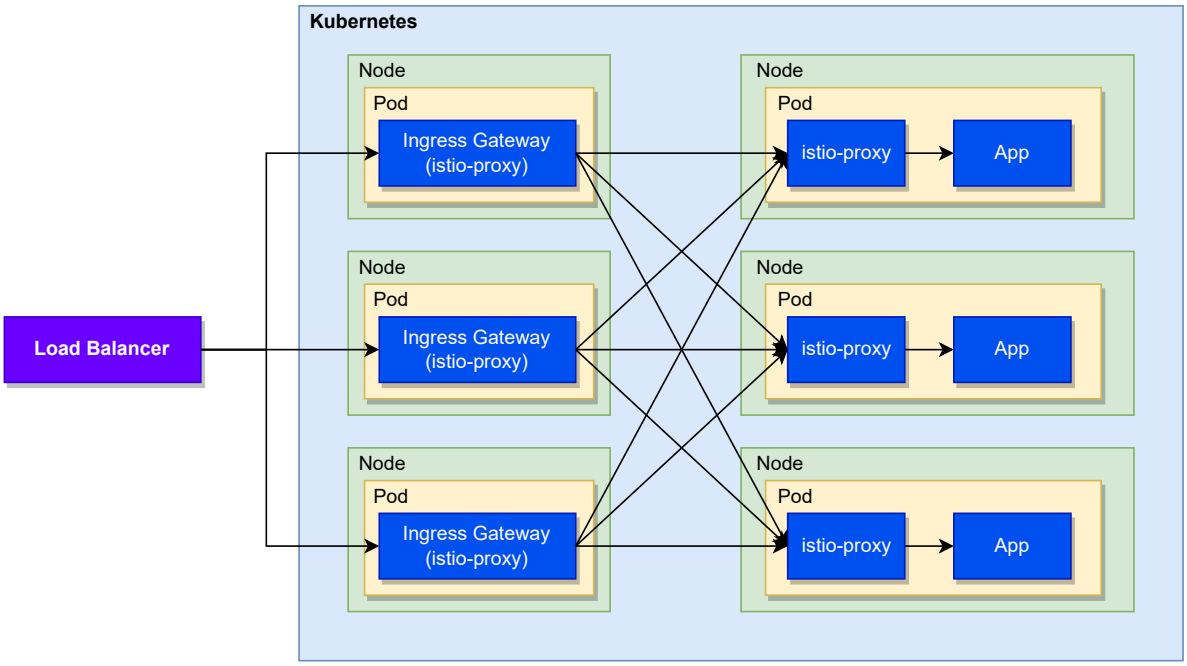


図 14: istio ingress gateway の概略図

で設定変更を動的に変更することになります。そのため、どの istio-proxy(Gateway もしくは Sidecar として機能している Envoy) に対して設定を適用させるか記述する必要があります。

ここでは、以下の図中の Istio Ingress Gateway に対して設定を変更します。

11.1 hosts でルーティングを分ける

例えば PC とスマートフォン (SP) でルーティング先を分けたい場合があります。これを実現するためにはまずは Gateway を宣言する必要があります。ここではわかりやすいように PC のルーティング先を pc.example.com、SP の行き先を sp.example.com として定義します。

PC 用 Gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: pc-example-com
  namespace: demo
spec:
  selector:
    app.kubernetes.io/name: istio-ingressgateway
    app.kubernetes.io/part-of: istio
  servers:
    - port:
        number: 33000
        name: http
        protocol: HTTP
```

```
    hosts:
      - pc.example.com
```

SP 用 Gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: sp-example-com
  namespace: demo
spec:
  selector:
    app.kubernetes.io/name: istio-ingressgateway
    app.kubernetes.io/part-of: istio
  servers:
    - port:
        number: 33000
        name: http
        protocol: HTTP
      hosts:
        - sp.example.com
```

このとき、.metadata.namespace と.spec.selector で設定を適用したい Istio IngressGateway を絞り込みます。 仮に Gateway を定義しなかった場合、Istio IngressGateway はリクエストを後方のマイクロサービスに疎通させません。次に、この Gateway を VirtualService に対してバインドします。

PC 版 Virtual Service

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: pc-route
  namespace: demo
spec:
  gateways:
    - pc-example-com
  hosts:
    - "*"
  http:
    - name: http-route
      match:
        - uri:
            prefix: /
      route:
        - destination:
```

```
            host: pc.demo.svc.cluster.local
            port:
              number: 80
          weight: 100
```

SP 版 Virtual Service

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: sp-route
  namespace: demo
spec:
  gateways:
    - sp-example-com
  hosts:
    - "*"
  http:
    - name: http-route
      match:
        - uri:
            prefix: /
      route:
        - destination:
            host: sp.demo.svc.cluster.local
            port:
              number: 80
          weight: 100
```

Virtual Service は.spec.gateways[] に Gateway の.metadata.name(namespace 内でユニーク) を指定することで、同一 namespace 内の Gateway を特定してバインドしています。

語弊を恐れずにこれらのコンポーネントの流れを書くと次のようになります。

```
[host]pc.example.com:33000          # アクセス
  → [Gateway]pc-example-com         # hosts と Port の宣言
    → [VirtualService]pc-route      # Gateway のバインド、PATH に対する Service へルーティング
      → [Service]pc.demo.svc.cluster.local # Pod に対するルーティング
```

以上で hosts に対するルーティングを実現することができます。

また、これらの操作により、同じ namespace 内で pc と sp で別々の Ingress Gateway に分離したい要求が発生した場合は Ingress Gateway が増えた場合は Gateway の.metadata.selector を調整することで対応することができます。

11.2 Header や QueryParameter でルーティングする

Virtual Service は URI や Header、Query Parameter などの情報をもとに、ルーティング先の Service を変更することができます。Argo Rollouts はこの機能を利用して BlueGreen デプロイや、Canary デプロイを実現しています。デプロイの機能として利用するだけでなく例えば「Preview 版の機能を特定の Header 情報を含む場合にのみ公開する」などを実現することが可能です。

例えば、Header に `version: v2` が含まれる場合は、`.metadata.name=pc-v2` の Service にルーティングする設定は次のように書くことができます。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: include-preview-route
  namespace: demo
spec:
  gateways:
    - pc-example-com
  hosts:
    - "*"
  http:
    - name: http-preview
      match:
        - uri:
            prefix: /
          headers:  # queryParams にすると ?preview=v2 でルーティングされる
            preview:
              version: v2
      route:
        - destination:
            host: pc-v2.demo.svc.cluster.local
            port:
              number: 80
    - name: http-common
      match:
        - uri:
            prefix: /
      route:
        - destination:
            host: pc.demo.svc.cluster.local
            port:
              number: 80
```

試験的に実現したい機能などを本番環境に投入したい場合などに役に立つことは間違いないでしょう。

VirtualService を記述する注意点として、`.spec.http[]` は条件の厳しいものが先にくるように記述する必

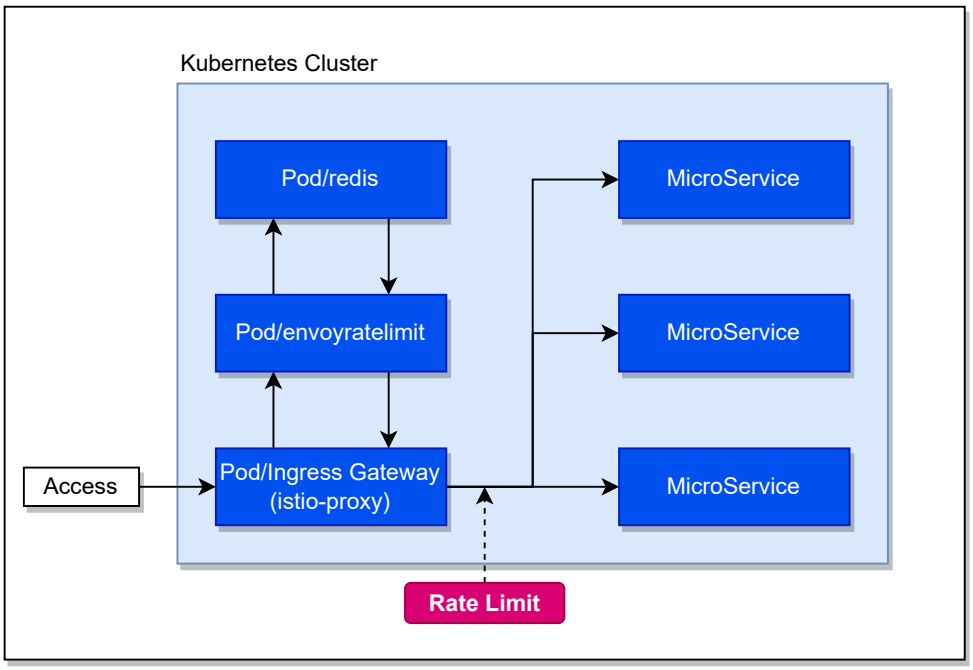


図 15: Global RateLimit の概略

要があります。また、ルーティング先の `host` となる Service が存在しない場合は Manifest が Apply できないことがあります。

12 Global RateLimit

Global Rate Limit は Ingress Gateway より後方側にいる Pod に対するリクエストの流量制限を実施します。すなわち、Kubernetes クラスターまではリクエストは到達します。envoyproxy/ratelimit はこれを実現するためのリファレンス実装で、外部サービスとして後付で導入することが可能です。

Traffic が ingress gateway に到達した後の大雑把な流れは次のとおりです。

1. `rate_limit_service` で指定されたマイクロサービスにたいして grpc で問い合わせをします。
2. `envoyratelimit` は `redis`（`memcache` も利用可）に格納した Descriptor に対するリクエスト数の計算を実施します。
   - ratelimit.go#L164
   - fixed_cache_impl.go#L39-L128
3. 結果を ingress gateway に対して RateLimitResponse に乗せて返却
4. ingress gateway はレスポンスを受けて 429 を返すかどうか決定する。

12.1 Global Ratelimit の設定

`envoyproxy/ratelimit` を利用するには 2 つの設定が必要です。

1. Descriptor の設置
2. Descriptor に対する Rate Limit の定義

Descriptor は Envoy（istio-proxy）に対して定義することが可能で、Gateway として機能している istio-

proxy だけでなく、Sidecar として搭載されている istio-proxy に対しても定義することが可能です。

Descriptor は Action によって条件が定義することができ、これをリファレンス実装された ratelimit のマイクロサービスで使用することにより、特定の PATH や header に対して ratelimit を適用することができます。

具体的な例を示しましょう。

12.1.1 :path 単位で Rate Limit をかける

例えば、/というパスに対して Rate Limit を作りたい場合、まず Descriptor を Gateway の istio-proxy に対して作る必要があります。

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ratelimit-actions
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  configPatches:
    - applyTo: VIRTUAL_HOST
      match:
        context: GATEWAY
        routeConfiguration:
          vhost:
            name: ""
            route:
              action: ANY
      patch:
        operation: MERGE
        value:
          rate_limits:
            - actions:
                - request_headers:
                    # HTTPの場合Request Headerの`:path`にURIが格納されている
                    header_name: ":path"
                    # "PATH"という名前でDescriptorを作成する
                    descriptor_key: PATH
```

※ これ以降、rate_limits より上層の定義は省略します。

この PATH に対して envoyproyx/ratelimit によって 10 rpm (request / minutes) の制限を加える定義は次のようになります。

```
descriptors:
  - key: PATH # actionsに定義したdescriptor_key
    value: /  # Descriptorが取得するValue、つまり今回の場合はURI
```

```
    rate_limit:
      unit: minute
      requests_per_unit: 10
```

正規表現で Descriptor を絞り込む

実践的にはより複雑な URI に対して Rate Limit を書けることになるます。ニコニコ生放送では`/watch/lv12345...`といった具合の URI に対して制限を適用する必要があります。

この場合 Descriptor の定義は正規表現を以下のように記述することで表現することができます。

```
rate_limits:
  - actions:
      - header_value_match:
          descriptor_value: watch
          headers:
            - name: ":path"
              safe_regex_match:
                google_re2: {}
                regex: /watch/(lv\d+)
```

Rate Limit は前回と同様に Descriptor に対して記述するだけで定義できます。

```
descriptors:
  - key: header_match
    value: watch
    rate_limit:
      unit: second
      requests_per_unit: 9999999
```

12.2 istio-proxy と Rate Limit のマイクロサービスとの連携

次のような EnvoyFilter を Ingress Gateway に対して適用しています。

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ratelimit-gateway
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: GATEWAY
```

```
        listener:
          filterChain:
            filter:
              name: envoy.filters.network.http_connection_manager
              subFilter:
                name: envoy.filters.http.router
      patch:
        operation: INSERT_BEFORE
        value:
          name: envoy.filters.http.ratelimit
          typed_config:
            "@type":
↪   type.googleapis.com/envoy.extensions.filters.http.ratelimit.v3.RateLimit
            domain: nicolive
            # Envoy Ratelimit との疎通が失敗した場合など、
            # トラフィックを Upstream に流れるようにする
            failure_mode_deny: true
            timeout: 10s
            rate_limit_service:
              grpc_service:
                envoy_grpc:
                  # Istio のドキュメントとここが異なる
                  cluster_name:
↪   outbound|8081||ratelimit.mynamespace.svc.cluster.local
                  authority: ratelimit.mynamespace.svc.cluster.local
              transport_api_version: V3
```

Istio のドキュメントをそのまま流用した場合、Envoy の Cluster 定義を追加する記述がありますが、これは Kubernetes の Service を以下のように定義すると EDS（Endpoint Discovery Service）によって利用可能な `cluster_name: outbound|8081||ratelimit.mynamespace.svc.cluster.local` が自動的に定義されます。

これにより、cluster 名を `STATIC_DNS`（L4）から EDS（L7）に変更することができ、Rate Limit の Pod の更新時に瞬断が発生しなくなります。

```
apiVersion: v1
kind: Service
metadata:
  name: ratelimit
spec:
  type: ClusterIP
  selector:
    app: ratelimit
    app.kubernetes.io/name: ratelimit
```

```
ports:
  - name: http-ratelimit-svc
    port: 8080
    targetPort: 8080
    protocol: TCP
  - name: grpc-ratelimit-svc
    port: 8081
    targetPort: 8081
    protocol: TCP
  - name: http-debug-ratelimit-svc
    port: 6070
    targetPort: 6070
    protocol: TCP
```

また、RateLimit との疎通は `failure_mode_deny: false` を指定しています。Ingress Gateway に対するアクセスはすべてが一度 Rate Limit の Pod を経由します。デフォルトの場合（`failure_mode_deny: true`）、何らかの理由で Rate Limit の Pod との疎通が取れなくなった場合に Ingress Gateway からユーザーに対して 503 エラーが返るようになります。この影響はサービス全体に波及するためこのフラグは `false` にしています。

仮に Global RateLimit が機能しなくなった場合、リクエストはそのまま Upstream 側のマイクロサービスまで貫通しますが、異常なリクエストに対しては次に紹介する Local Ratelimit によって多重の防御が用意されています。

13 Local RateLimit

Global RateLimit との違い

Local RateLimit と Global RateLimit の違いは守るスコープの違いにあります。 Global RateLimit は Upstream 側のシステムを守るため、RateLimit のマイクロサービス間でリクエスト数を共有するためのストア (redis など) を外側に持っています。それに対し、Local RateLimit は RateLimit を提供する Proxy だけがリクエスト数を保持でいればよいためインメモリーで実装することができます。

Kubernetes 上における Local RateLimit の設置候補

Local RateLimit を実施する候補は 2 つあります。

1. istio-proxy（Envoy）の Local Ratelimit 機能を利用する
2. nginx を istio-proxy と App の間に立たせ、nginx の RateLimit を利用する

13.1 envoy と nginx の Rate Limit アルゴリズムの違い

envoy と nginx では Rate Limit のアルゴリズムが異なります。ゆえに、バースト性のあるトラフィックに対する制限が異なり、どちらからの乗り換えに対しても検証なしで乗り換えすることはできません。

Proxy Server	Rate Limit Algorithm
nginx	Leaky Bucket

Proxy Server	Rate Limit Algorithm
envoy	Token Bucket

13.2 envoy と nginx の設定例

例えば myapp というアプリケーションに対して 10 rps の Rate Limit の制限をかけ、バースト時のリクエストは 50 rps まで受け付けるようにした場合次のように記述できます。

13.2.1 Envoy のの Local Rate Limit 設定例

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: filter-local-ratelimit-myapp
spec:
  workloadSelector:
    labels:
      app: myapp
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: SIDECAR_INBOUND
        listener:
          filterChain:
            filter:
              name: envoy.filters.network.http_connection_manager
      patch:
        operation: INSERT_BEFORE
            value:
              stat_prefix: http_local_rate_limiter
              # Local Ratelimit のパラメーター
              token_bucket:
                max_tokens: 50
                tokens_per_fill: 10
                fill_interval: 1s
              filter_enabled:
                runtime_key: local_rate_limit_enabled
                default_value:
                  numerator: 100
                  denominator: HUNDRED
              filter_enforced:
                runtime_key: local_rate_limit_enforced
```

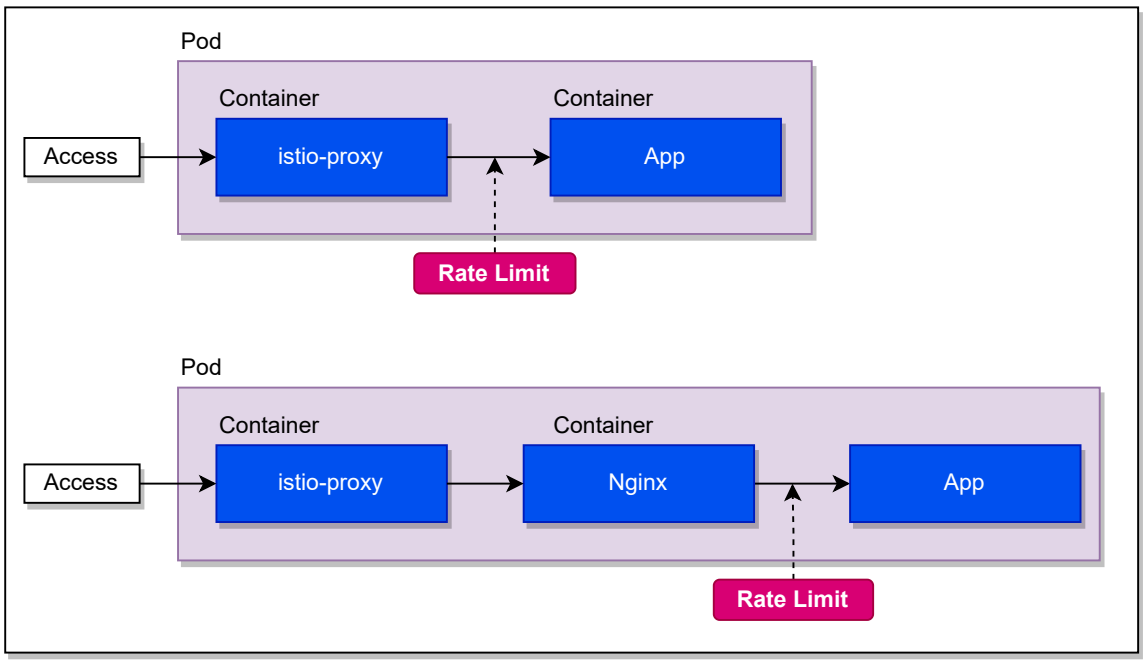


図 16: Local RateLimit の概略

```
                default_value:
                  numerator: 100
                  denominator: HUNDRED
```

13.2.2 nginx の Rate Limit 設定例

```
http {
  limit_req_zone myapp_zone zone=myapp_zone:1m rate=10r/s;
  server {
    location /myapp {
      limit_req zone=myapp_zone burst=50 delay=10;
      limit_req_status 429;
      # 省略 ...
    }
  }
}
```

13.3 どちらを選ぶか

これらを選択するにあたりバースト時の挙動を確認する必要があります。例えば、前述の設定で 1 つの Pod に対して 70rps 来た場合、envoy と nginx は次の挙動をします。

proxy	挙動
envoy	`max_tokens: 50` まで消費し、50rps が App まで到達する。`fill_interval` が 10 で指定されているためそれ以降のリクエストは token が回復するまで `10rps` を維持する。`max_tokens: 50` から溢れた 20rps は 429 を返す。
nginx	`burst=50` で指定したリクエスト数まで一度 nginx で受け付け、`delay=10` で指定した 10 req 分だけ App まで到達する。残りの 40req はキューイングされて FIFO で処理される。`burst=50` から溢れた 20rps は 429 を返す。

つまり、envoy で Local RateLimit を敷いた場合は `max_tokens` で指定したリクエストはたしかに受け付けますが、それをキューせずに Upstream の App にリクエストを流します。この流量をアプリケーション側が処理することが可能であれば `envoy` の Rate Limit を採用することができます。これが逆に処理できない場合は App のコンテナが処理しきれずに 503 を返します。

したがって、バースト耐性を獲得しつつ、移行というスケジュールが決まった範疇で選択できるのは `nginx` を利用した Local Rate Limit になります。istio-proxy に加えて nginx も proxy として挟まりスループットが若干悪くなりますが、BFF を構成するサーバーの応答速度と比較して十分に小さいため許容することにしました。

13.4 今後どうするか

Pod を構成する要素として Proxy が 2 段構えになっているのは多少格好は悪いですが、うまく機能しています。ただし、後述しますが、envoy や nginx の RateLimit では負荷の上昇を防げないパターン問題点もあります。アクセス傾向や Pod の Metrics などを総合的に鑑みて Rate Limit の構成と設定値を決めていく必要があります。

14 RateLimit で負荷の上昇を防げないパターン

RateLimit を導入したからといって必ずしも負荷状態をバーストさせない状態を作れるわけではありません。Global RateLimit として利用可能な envoyproxy/ratelimit や、Envoy 本体にある Local RateLimit、nginx の持つ RateLimit の実装を注意深く見ると、RateLimit の計算に利用するのは**リクエスト**のみです。すなわち、レスポンスが返ったことを確認してカウントアップしたリクエスト数を下げるわけではないのです。これはつまり、RateLimit よりも後方のサーバーがコネクションをキューイングした場合、RateLimit で指定したリクエスト数より多くのリクエストを処理することが可能になります。

14.1 発生メカニズム

簡略化したシーケンス図でまずは状況を説明します。図中には以下が登場します。

名称	役割
User	ブラウザなどのクライアント
Proxy	Request に対する Rate Limit を適用
Server(Frontend)	BFF Server と置き換えても問題ない。図中の `Heavy Task` は Server Side Rendering と解釈するとよい。
Server(Backend)	Server(Frontend) が少なくとも 1 つ以上はクリティカルに依存するサーバー

RateLimit が効いているにも関わらず Server(Frontend) の CPU 使用率が上昇する流れ

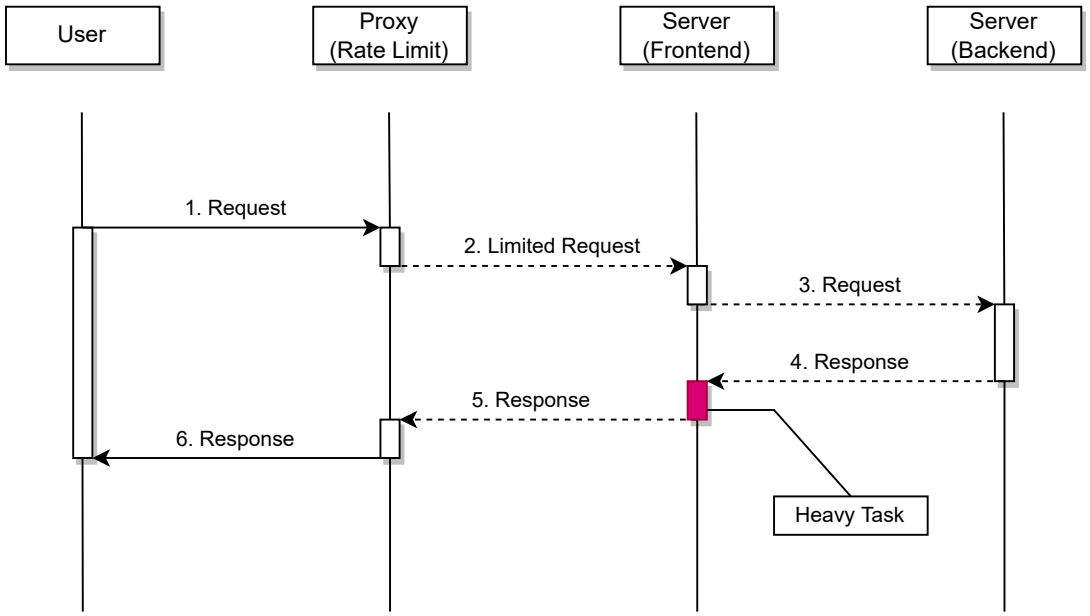


図 17: RateLimit が有効に使えないシーケンス図

1. 何らかの理由によって `Server(Backend)` のレスポンスが遅くなる場合が発生
2. このとき、`Server(Frontend)` からのリクエストは設定した timeout までコネクションを維持し続ける
3. RateLimit は Request に対してのみ有効なため、Limit が効かない間は `Server(Frontend)` にリクエストを送信する
4. `Server(Backend)` のレスポンスタイムが正常に戻ると図中の 4 の Response が発生する
5. すると、`Server(Frontend)` にキューイングされたリクエスト `Heavy Task` が定常よりも多く実行される
6. その結果、`Server(Frontend)` の CPU 使用率が上昇する

14.1.1 問題点と対策

基本的にどのマイクロサービスも、連携しているマイクロサービスに SLA(Service Level Agreements) を満たせない可能性がある前提で振る舞いを決める必要があります。

問題点

今回のケースだと、`Server(Backend)` のレスポンスが伸びた場合、`Server(Frontend)` がリクエストをキューイングするところに問題点があります。`Server(Frontend)` がレスポンスを返すために必要な情報を集めるために**長めに**タイムアウトを取っている場合、障害時にこのタイムアウト分だけコネクションが維持されることを忘れてはいけません。

対策

アプリケーションレベルの対応だと適切なタイムアウト設定や、そもそも高負荷になりうる処理がバーストしないように処理を組み替えるなどの対応が必要になってきます。とはいえ、そこまで工数をかけられない場合はコネクション数を絞ったり、istio(envoy) のサーキットブレーカーなどの機能を有効にして、問題が波及しないように布石を打つ必要があります。

- Circuit breakers — envoy documentation
- Istio / Circuit Breaking

15　水平スケール

Kubernetes は監視している CPU 使用率などの Metrics をもとに Pod の数を自動的にスケーリングさせる機構、Horizontal Pod Autoscaling を持っています。metrics server を利用すると CPU 使用率や Memory 使用量をベースに水平スケールを支援してくれます。

15.1　Horizontal Pod Autoscaler

Horizontal Pod Autoscaler（以下 HPA）は観測された Metrics を元に指定の Pod の replicas を増減させる仕組みです。Manifest の書き方はシンプルで、`autoscaling/v2beta2` で記述すると次のようになります。

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: myapp
spec:
  minReplicas: 10
  maxReplicas: 20
  scaleTargetRef:
    kind: Deployment # Argo Rollouts を使用している場合は Rollout を指定する
    apiVersion: argoproj.io/v1alpha1
    name: myapp
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80
```

これは探索された Pod の CPU 使用率が 80% を超えるようになった場合に、それを下回るように Pod 数を増加させます。逆に下回った場合は `minReplicas` まで戻るように働きます。

注意点として、

$$\text{CPU 使用率/1Pod} = \frac{\text{Pod を構成するコンテナ全体の使用中のコア数の合計}}{\text{各コンテナの limits.cpu の合計}}$$

で算出されます。つまり、Sidecar として挿入される istio-proxy の Resource も考慮した上で HPA の閾値を考慮する必要があります。実際に使用されている Resource は以下の CLI コマンドで確認できます。

```
kubectl top pod myapp-[podid]
kubectl top pod --containers myapp-[podid]
```

- Support for resource metrics

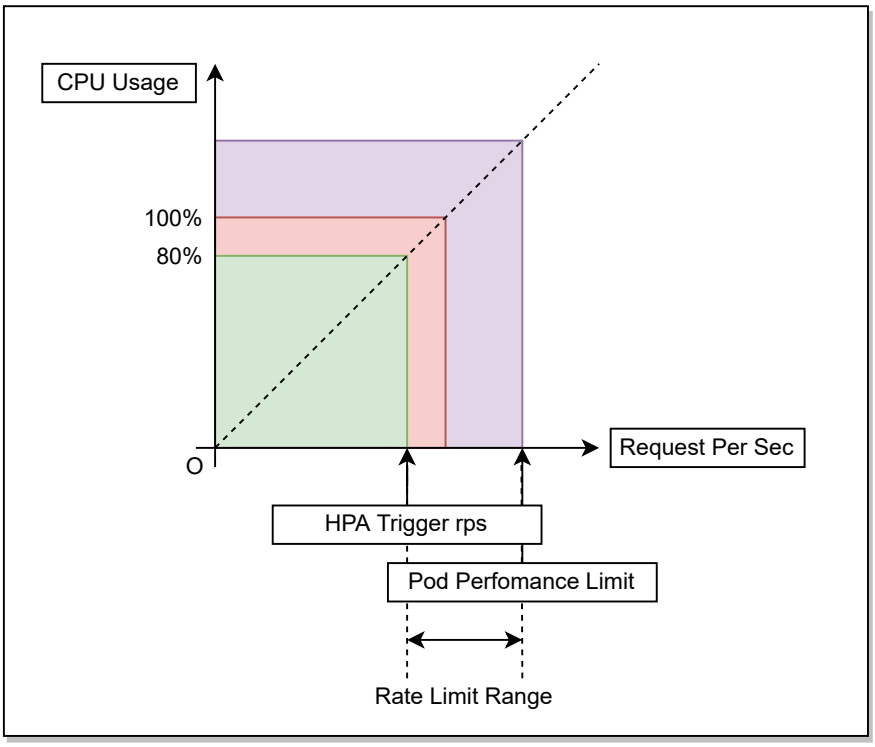


図 18: HPA と RateLimit の関係

これらを踏まえた上で HPA の設計をします。

15.2 リソースの値をどうやって決めるか

BFF を含む Pod の大きな特徴として、

1. Stateless
2. 他マイクロサービスからデータを収集する
3. Server Side Rendering を実施することもある

という点が上げられます。つまり、リクエストが BFF に滞在する時間が長くなることを基本的には想定はしていません。このことから単純に Request Per Sec(RPS) に比例して CPU 使用率が上昇することが予測できるため次のような比例グラフが書けます。

ただし、Pod の性能限界が存在するため無尽蔵に 1 つの Pod の RPS が伸びるわけではありません。途中で比例グラフが破綻するか、正常レスポンスを返せなくなるところがあります。図中の `Pod Performance Limit` はこれを示しています。

また、(Global / Local) Rate Limit で流量制御は Pod が `Pod Performance Limit` の rps よりも低く、HPA が発動する RPS よりも大きく指定する必要があります。

これらの値を決定するためには負荷試験を実施することで値を予測することが可能になります。

15.3 バースト性のあるリクエストをどうやって耐えるか

予測可能なバーストリクエスト数は事前にスケールアウトしておくことで必要なスループットを確保することができます。ニコニコ生放送のように人気番組が発生するようなケースは HPA External Metrics を利用して動的にスケールアウトをスケジューリングする方法が考えられます（参考文献 1, 2)。

しかしながら、実際には予測不能な負荷はどうするか考える必要があります。結論から言えば Pod 数を単純に多くして、定常時の Resource の CPU Request をその分小さくします（下限はあります）。そして、サービスが担保する最大 rps を Global Rate Limit によって制限することでそれを超えるリクエストはステータスコード 429 を返すようにします。

つまり次のような計算式で rps を最大にするための replicas と `resources.requests.cpu` を算出できます。

$$\text{replicas} \times \text{containers[].resources.requests.cpu} = \text{全体の requests.cpu}$$

$$\text{replicas} \times \text{HPA Trigger rps} = \text{スケールアウトしない場合の全体の rps}$$

$$\text{スケールアウトしない最大の rps} = \frac{\text{全体の requests.cpu (最大値)}}{\text{containers[].resources.requests.cpu (最小値)}} \times \text{HPA Trigger rps}$$

注意

- `containers[].resources.requests.cpu` は下げすぎると Pod が起動しないことがあるため、下限があります。
- **全体の** `requests.cpu` はクラスターのリソース内でしか割り当てられないため、上限があります。

これでスケールアウトせずに処理可能な rps の最大値を得ることができます。
また、Global RateLimit の値は次のどちらか小さい方の値を採用します。

1. BFF がスケールアウトせずに処理できる rps（前述）
2. BFF より後方のマイクロサービスが処理可能な rps の最大値

これを超えるような場合は増資増強が必要になってきます。

15.4 水平スケール設計の今後

バースト耐性を持たせるためには大量の Pod を常時配置しておく必要がありますが、やはりそれでは余剰リソースが出てしまうので、Kubernetes HPA External Metrics を利用したスケールアウトの柔軟なスケジューリングの構成も必要になってきます。ただし、リソースはやはり有限であるため優先度の高い Pod からスケールアウトするように Pod Priority and Preemption も決めていく必要があります。これらの精度を上げていくには、Pod を構成する個々のアプリケーションのパフォーマンスについてより詳しくなる必要があり、洗練された負荷試験が必要なことがわかります。

15.5 参考文献

1. Kubernetes HPA External Metrics を利用した Scheduled-Scaling - スタディサプリ Product Team Blog
2. Kubernetes HPA External Metrics の事例紹介｜メルカリエンジニアリング

16 負荷試験

16.1 負荷試験の目的

Docker Swarm から Kubernetes に移行するにあたり、移行前のスペックを移行後に同等以上のスペックを発揮することを試験する必要があります。したがって、アプリケーション自体の詳細な性能テストではなく、

クラスターとして同等の性能が発揮できていることが大雑把でも確認できればよい、というのがここでのゴールとなります。

評価するためには Prometheus や DataDog Agent などから収集したデータを Grafana や DataDog Dashboard で確認しつつ再現性を持った試験を実施する必要があります。

16.2 何を試験するか

実際に試験して確認した内容を次の表にまとめました。

試験内容	目的
経路上の Proxy を最小台数にしたときに想定する rps を処理ことができるか確かめる	Connection Pool や TCP ESTABLISHED に余裕があるか確認する
Runtime（nodejs）やライブラリの処理可能な rps を計測する	同じ Runtime やライブラリを使用しているアプリケーションの CPU/MEM リソースの基準値を見積もる
Pod をスケールアウトしたときの rps に対するリソースの変化を確認する	Pod 数に対するリソース使用量と rps の関係を把握する
連続的にリクエストを投げることでほかサービスに影響がないか確認する	負荷試験の対象は挙動に問題ないが、同じ Gateway を使っている場合や同じマシンに乗っている他のサービスに影響がないことを確かめる

基本的に「性能限界」と「リソースの見積」のための試験になります。これらの情報を元に、常設の replicas の見積りや、Horizontal Pod Autoscaler のための CPU の limit 設定、 共倒れを防ぐための Rate Limit 設定を割り出していきます。

合わせてリソースの値をどうやって決めるかを確認してみてください。

16.3 試験方法

16.3.1 他のマイクロサービスと連携していない HTTP サーバーを用意する

Kubernetes の一連の動作検証も含め以下のようなサーバーを用意しておくと検証が捗ります。

```
import * as express from "express";

const app = express();

app.get("/", (req: express.Request, res: express.Response) => {
  res.send("ok");
});

const httpServer = server.listen(3000, "0.0.0.0");

process.on("SIGTERM", () => {
  httpServer.close();
});
```

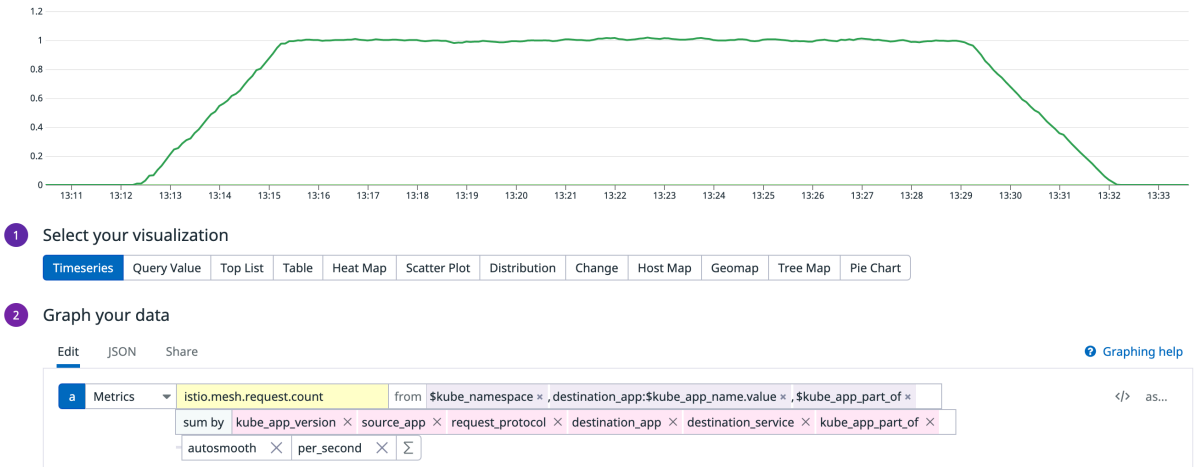


図 19: DataDog での 1rps

このサーバーをコンテナ化して Kubernetes 上で稼働させ、疎通確認や負荷試験の対象として扱います。

16.3.2 負荷試験ツール

使用したツールで簡易で便利だったのが tsenart/vegeta でした。Binary で手に入るためセットアップが容易で適当なサーバーから直接実行したり、docker-compose でローカルに環境を作り、その中で動作確認ができるのは非常に作業効率が高いです。

例えば 1rps のテストを実施したい場合は次のような Shell を実行するだけで済みます。

```
echo "GET http://test-server:3000" | vegeta attack -rate 1/1s > /dev/null
```

これを istio-proxy から収集された Metrics を Grafana で可視化することで、まずは集計クエリが正しいことを確認していきます。ある程度高い rps で負荷をかけ続けると、パラメーターで指定した rps よりも低い値で可視化されることがありますが、負荷試験用のクライアントの性能だったり、常に均質なネットワークを提供できるわけではなかったりするためです。

とはいえ負荷試験時の rps 目標値があるため、観測される rps が目標値になるまでクライアント側の rps を上げるか、クライアントを分散して負荷を高めるなど工夫をします。

16.3.3 Proxy からリクエストを Mirroring する

いきなり移行対象のクラスターやサーバーに対してトラフィックを 100% 向けるのはかなり危険です。見積もりが正しくともそれが確実かどうかはやってみなければわかりません。安全に取るなら同じ流量のリクエストを移行後の機材に向けて検証した後に徐々に Traffic Weight を調整していくのが無難でしょう。

一番手っ取り早いのはリクエストをミラーリングして実際のリクエストを受け付けることです。BFF のサーバーの場合、リクエストのほとんどが GET であるためアクセスログを出力するロードバランサーより後方でミラーリングを実施することが可能です。

実際に使用したミラーリングは nginx の機能によるものを利用しています。

- http://nginx.org/en/docs/http/ngx_http_mirror_module.html

実際には次のような経路でミラーリングを実施しました。

移行後のクラスターに対して GET リクエストをミラーリングし、CPU/MEM などのリソース使用量、エラー率をモニタリングツールで監視しました。この結果と単純な負荷試験による見積もりの誤差を把握した上

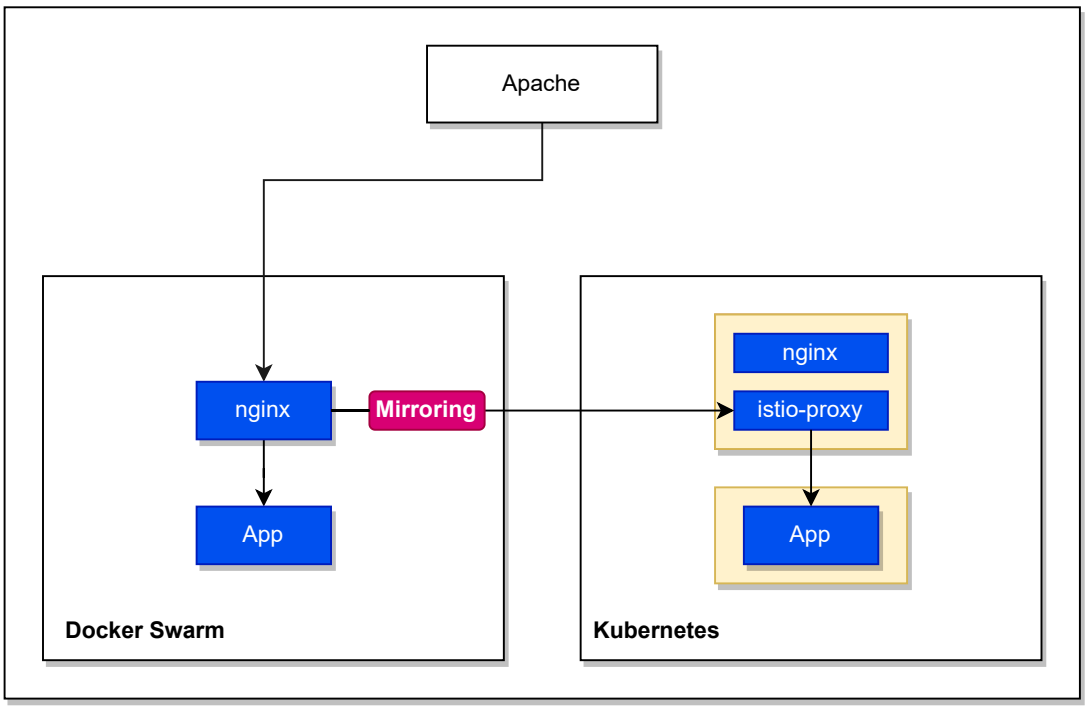


図 20: リクエストのミラーリング概略図

で移行に備えました。

17　モニタリング

言わずもがな、サーバーを持ちサービスを運用する上でモニタリングは必須です。収集したログや Metrics は不正アクセスや障害の検知、アラートの発報などサービスの運営をする上で必ず必要な情報です。

17.1　利用しているモニタリングツール

今回利用しているモニタリングツールは以下の 2 つがあります。

	Explorer	DashBoard
1	Prometheus	Grafana
2	DataDog Agent	DataDog

2 つ存在する理由は、費用的な面とツールの精度を確認するための理由があります。

1. DataDog は本番環境で使う。一部の開発環境でも検証可能な状態で利用する。
2. 上記の DataDog を使わない環境では Prometheus と Grafana で代用する
3. 2 つのツールでモニタリングの精度を比較する（本来は最低でも 3 つあったほうが良い）

どちらも同じ Metrics を収集できるため単体での Observability の差に大きく違いはありません。 ただ DataLake として DataDog がすでに基盤として存在しているためメインは DataDog に各種情報が集約されています。

17.2 BFF サーバーの何を観測するか

BFF サーバーにおいて観測する主要なものは以下の表にまとめました。どれも時系列データとして記録されるため時間変化が分かる状態になっています

BFF サーバーとして主に観測しているのは次のようなメトリクスになります。

※ 略語

- rps = Request Per Seconds (1 秒間あたりのリクエスト数)

メトリクス	目的
CPU 使用量の最大値、平均値、合計値	想定値との比較
Memory 使用量の最大値、平均値、合計値	メモリリークの観測、想定値との比較
HTTP Status Code の数	200, 300, 400, 500 系の観測
リクエスト総数に対する Status Code 500 の数	エラー率の観測
マイクロサービス間の Response Time（HTTP, GRPC)	ボトルネックの観測
マイクロサービス間の rps	実効 rps が想定内か観測する
Node ごとの replicas	スケールアウトやデプロイの変化を観測する

BFF 以外はアクセスログを出力する fluent-bit や、RateLimit を実施しているマイクロサービスも監視する対象となります。

17.3 DataDog の例

具体的な例を紹介します。 DataDog の Kubernetes タグ抽出では粒度の低いタグとして、 Kubernetes の Recommend Labels が利用できます。

DataDog のタグ	Kubernetes の Pod Label
`kube_app_name`	`app.kubernetes.io/name`
`kube_app_instance`	`app.kubernetes.io/instance`
`kube_app_version`	`app.kubernetes.io/version`
`kube_app_component`	`app.kubernetes.io/component`
`kube_app_part_of`	`app.kubernetes.io/part-of`
`kube_app_managed_by`	`app.kubernetes.io/managed-by`

TypeScript で Kubernetes の manifest を記述するで紹介したように、これらのラベルを機械的に付与していくと DataDog 上での分解能が飛躍的に向上します。最も利用頻度の高いタグは `kube_app_name` と `kube_app_version` で、これらの 2 つは非常に重要な役割を担います。

例えば、新しい Pod をデプロイした際に、`kube_app_version` をフィルタリングのクエリとして利用することで、どの時刻で新旧のバージョンが入れ替わったのかが可視化されます。

DashBoard では他の指標と見比べる事が可能ですので、例えば新しいバージョンにバグが有り、マイクロサービス間の通信のエラー率が高まった場合の観測が可能です。

上記のクエリは次のようになっています。

Template Parameter

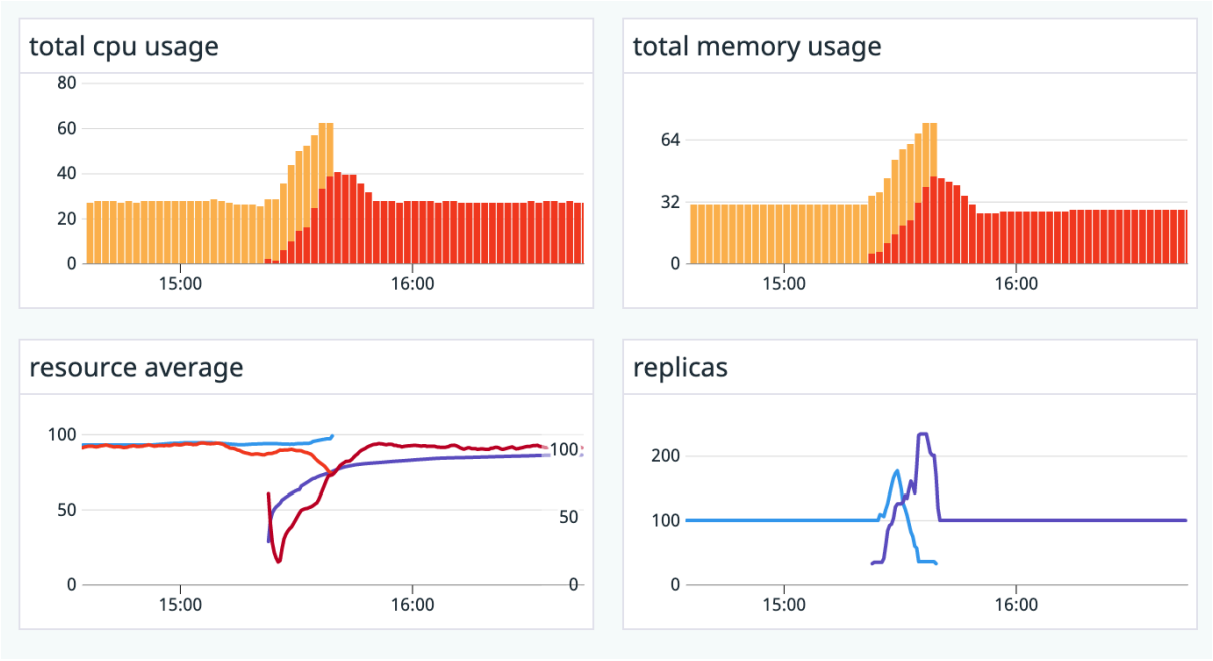


図 21: デプロイ時のダッシュボード

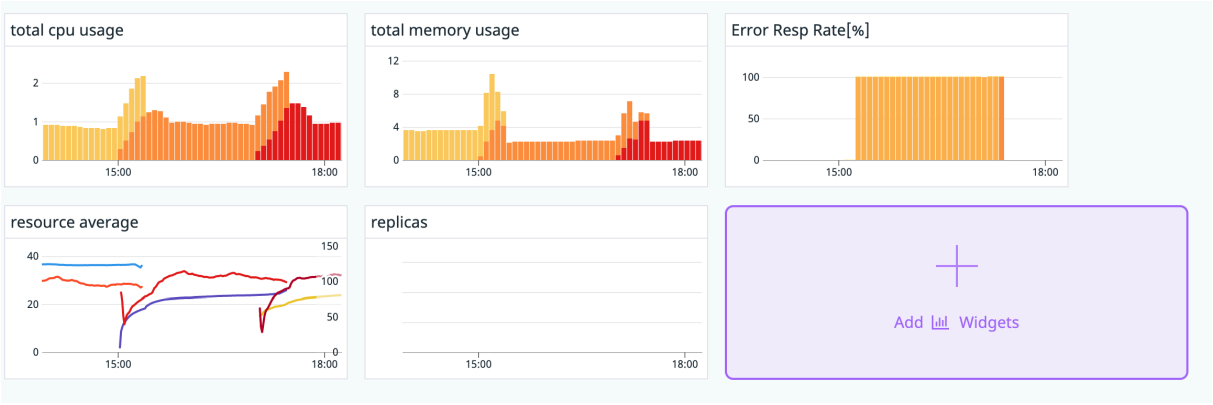


図 22: ロールバックの例

- $kube_app_name
- $kube_namespace

```
# Total CPU Usage
autosmooth(sum:kubernetes.cpu.usage.total{$kube_app_name,$kube_namespace} by
↪  {kube_app_version})
# Total
autosmooth(sum:kubernetes.memory.usage{$kube_app_name,$kube_namespace} by
↪  {kube_app_version})

# Error Resp Rate[%]
a / b * 100
## a
```

```
  sum:istio.mesh.request.count.total{response_code:5*,$kube_app_name,$kube_namespace}
  ↪ by
  ↪ {response_code,destination_service,request_protocol,kube_app_version}.as_count()
## b
  sum:istio.mesh.request.count.total{$kube_app_name,$kube_namespace} by
  ↪ {destination_service,request_protocol}.as_count()
```

より高度な演算が可能なため、`Error Resp Rate[%]` のように複数の Metrics を組み合わせることが可能です。

17.4 モニタリングの次にやること

更新のたびに Dashboard を見に行き、バグがないか確認するはいわゆる「トイル」な仕事になります。Argo Rollouts では Progressive Delivery を支援するための Analysis 機能があり、Prometheus や DataDog などの集計データをもとにデプロイを続行するかどうか自動的に判断する事が可能です。

- https://argoproj.github.io/argo-rollouts/analysis/datadog/

これを導入するためにはまずは信頼できる指標の作成が必要で、BFF サーバーは何を指標とするかはこれから吟味が必要です。

18 負荷対策

18.1 nodeSelector

Node は Pod が配置される VM や物理マシンですが、配置される Pod の処理内容によって使用されるリソースが大きく変わることがあります。具体的には Ingress Gateway はクラスター内外のトラフィックを集中的に処理することが事前にわかっています。また、Ingress Gateway がなければアプリケーションにアクセスできないため、必ずリソースが枯渇しない状態にする必要があります。そのため、Gateway としての役割以外を持つ Pod と別の Node に配置されるようにすることで、Pod が安定して稼働できるリソースの確保を実現します。

Kubernetes は Node に対してラベルを付与し、Pod に nodeSelector を付与することで指定の Node に対して Pod を配置することができます。Node に付与されている label は以下のコマンドで確認することができます。

```
$ kubectl get nodes --show-labels
# 簡単のため表示を省略
gateway01   Ready    <none>        1d    v1.21.12   node-role=gateway
gateway02   Ready    <none>        1d    v1.21.12   node-role=gateway
worker01    Ready    <none>        1d    v1.21.12   node-role=worker
worker02    Ready    <none>        1d    v1.21.12   node-role=worker
```

この場合、`worker` に対して Pod を配置したい場合は次のように記述することができます。

```
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: app
  namespace: demo
spec:
  template:
    # 中略
    spec:
      nodeSelector:
        node-role: worker
```

- Node 上への Pod のスケジューリング | Kubernetes

18.2 PodAntiAffinity

Pod のスケジューリングはデフォルトのまま使用すると、Node に対する配置は明示的にコントロールされません。つまり、あるアプリケーションを搭載した Pod が Node に偏らないようにしたいが、偏ってしまう（逆も然り）など発生します。特に BFF サーバーはステートレスなサーバーであるため、分散配置されている方が望ましいでしょう。

Kubernetes では `podAffinity`(pod を条件に応じて集約) または `podAntiAffinity`(pod を条件に応じて分散) を指定することで Pod のスケジューリングをコントロールすることができます。例えば、「`app.kubernetes.io/name=myapp` というラベルを持つ Pod が、**なるべく**同じ Node に配置されない」スケジューリング設定は次のように表現できます。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  namespace: demo
  labels:
    app: myapp
    app.kubernetes.io/name: myapp
spec:
  template:
    spec:
      affinity:
        # ポッド間のスケジューリングルール
        # 以下の条件に合致する場所に配置しないポリシー
        podAntiAffinity: # 逆は podAntiAffinity
          # 優先的に考慮されるスケジューリングポリシー
          preferredDuringSchedulingIgnoredDuringExecution:
            - # 優先度の重み付け（1 - 100 の間で定義）
              # Node ごとに weight を加算し Score を算出し、
              # 最も高いスコアを獲得した Node に対して Pod が配置される
```

```
            weight: 1
            # app.kubernetes.io/name = "myapp" のラベルを持つ Pod
            podAffinityTerm:
              # Node をフィルタリングするためのキー。
              # この空間内の Node に対する Pod 間の Selector で Affinity の Score が計算される
              # kubernetes.io/hostname は各 Node に付与される識別子として利用できる
              # (Node のフィルタリング条件として偏りがないキー)
              topologyKey: kubernetes.io/hostname
              labelSelector:
                # app.kubernetes.io/name = "myapp" にマッチする Pod を集計対象とする
                matchExpressions:
                  - key: app.kubernetes.io/name
                    operator: In
                    values:
                      - myapp
```

preferredDuringSchedulingIgnoredDuringExecution は podAffinityTerm で指定された labelSelector に該当する Pod を topologyKey ごとに weight を加算してスコアを算出します。podAntiAffinity で利用されるスコアになるため、スコアが高くなるほど Pod はスコアの高い Node に対してなるべく**配置されない**ようになります。(podAntiAffinity はスコアの符号がマイナスで、podAffinity はスコアの符号がプラスと思えばわかりやすい。)

また、ここでは labelSelector で使う key や topologyKey は Well-Known Labels を利用しています。

- Node 上への Pod のスケジューリング | Kubernetes
- Well-Known Labels, Annotations and Taints | Kubernetes

19 アプリケーションの移行

Kubernetes 移行にあたり BFF サーバーで調整した内容を紹介します。

19.1 Graceful Shutdown

プロセス終了命令 (SIGTERM) などのシグナルを受け取ったときに、サーバーに残っているリクエストがレスポンスを返し終わってからプロセスを終了する仕組みです。これは Kubernetes でなくても実装すべき内容で、安全に処理を終わらせるために必要です。

express での実装例は次のとおりです。

```
import * as express from "express";

const app = express();

const httpServer = app.listen(process.env.PORT || 80);

// SIGNAL を受け取る
```

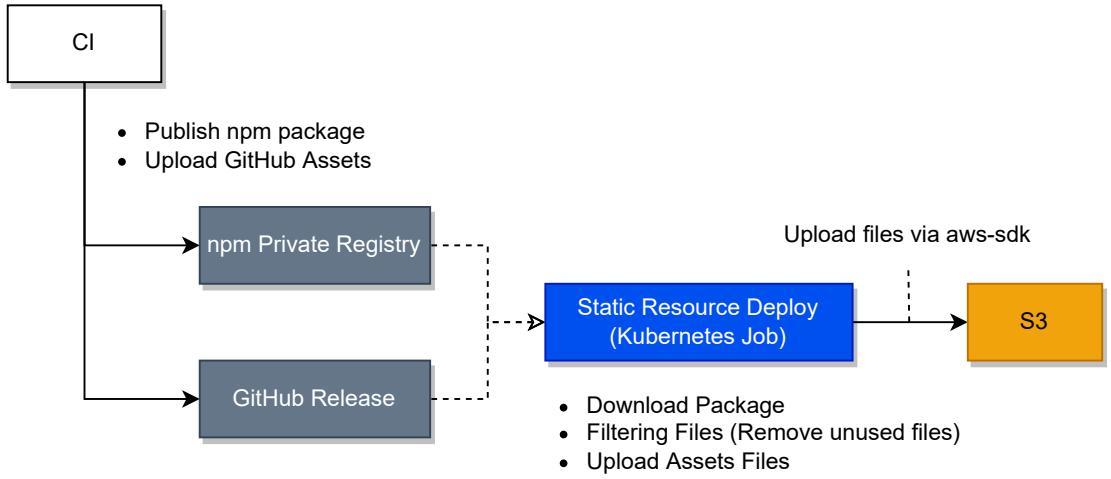


図 23: 静的リソースのデプロイフロー

```
process.on("SIGTERM", () => {
  httpServer.close();
});
```

process.on で SIGNAL を受け取ることができるため、そこで HTTP Server を Close するだけになります。 他にも WebSocket Server を起動している場合もここで close 処理を実施すると安全に終了できます。

19.2 静的リソースのアップロード

静的リソースは Amazon S3 にアップロードされたファイルを CDN(CloudFront) から配信する形式を取っています。そのため、S3 にアップロードする処理が必要で、移行前は Jenkins でこれを実行していました。

静的リソースはこれまで Jenkins のタスクによってアップロードしていましたが、Kubernetes の Job として移行しました。

具体的には、アプリケーションの CI によってリリース用の npm パッケージが作成され、Private Registry にアップロードされたり、 ものによっては GitHub の Release Assets にアップロードされたりしています。

Kubernetes 上の Job は

1. npm package にリリースする静的リソースをダウンロードし、
2. 静的リソースのホスティングに本当に必要なファイルだけを抽出し、
3. S3 にアップロード

という処理を実施しています。

内部の処理は環境変数で処理できるように実装されており、npm パッケージとアップロード先の S3 の保存先を選ぶことができます。また、Argo CD の Sync Wave と組み合わせて、アプリケーションの Deployment が Apply されるより前にこの Job を実行するように順序を決めることでクリティカルパスを形成することができます。以下は静的リソースのアップロード Job の例です。

```
apiVersion: batch/v1
kind: Job
metadata:
```

```
  name: static-resource-upload-job-v1.0.0
  labels:
    app: static-resource-upload-job-v1.0.0
    version: 1.0.0
    app.kubernetes.io/name: static-resource-upload-job
    app.kubernetes.io/version: 1.0.0
  annotations:
    argocd.argoproj.io/sync-wave: "-1"
spec:
  ttlSecondsAfterFinished: 86400 # 24 時間後に Job の Pod が消える
  template:
    metadata:
      labels:
        app: static-resource-upload-job-v1.0.0
        version: 1.0.0
        app.kubernetes.io/name: static-resource-upload-job
        app.kubernetes.io/version: 1.0.0
      annotations:
        sidecar.istio.io/inject: "false"
    spec:
      containers:
        - name: static-resource-upload-job
          env:
            - name: S3_REGION
              value: "upload-region"
            - name: S3_BUCKET
              value: "upload-bucket-name"
            - name: NPM_PACKAGE_NAME
              value: "npm package name"
            - name: NPM_PACKAGE_VERSION
              value: "version"
            - name: NPM_PACKAGE_DIST_DIR
              value: dist
            - name: UPLOAD_DIST_DIR
              value: "upload-dist-dir" # S3 の Key に該当
          image: # 静的リソースをアップロードするための処理が実装された Docker Image
      restartPolicy: Never
```

19.3 ルーティングの切り替え

Docker Swarm から Kubernetes に移行するにあたりマイクロサービス間の通信に必要な Host 名が変更されます。接続するマイクロサービスは必ずしも Kubernetes 上に存在しないため、各クラスターごとに別々の host になる可能性があります。Reverse Proxy でルーティング先をコントロールする場合でも、Reverse

Proxy に対してアプリケーションからのルーティング先を変更する必要があります。したがって、アプリケーションは少なくとも以下の 3 つの優先度で変更できると作業が容易になります。

環境変数 > リリース環境 > デフォルト値

Nodejs 環境における TypeScript の擬似的な実装は次のようになります。

```
const urlMap = {
  qa: {
    microServiceUrl: process.env.MICRO_SERVICE_URL || "https://qa.example.com",
  },
  development: {
    microServiceUrl: process.env.MICRO_SERVICE_URL ||
↪  "https://development.example.com",
  },
  production: {
    microServiceUrl: process.env.MICRO_SERVICE_URL ||
↪  "https://production.example.com",
  },
};

const getUrlMap = (env: "qa" | "development" | "production" =
↪  process.env.RELEASE_ENV) => {
  if (!["qa", "development", "production"].includes(env)) {
    throw new Error(`Invalid env ${env}`);
  }
  return urlMap[env];
}
```

環境変数が最優先なのは Twelve-Factor App で提言されている通りの理由になります。移行時のトラブル・シューティングや、開発環境、QA 環境における部分的な設定変更を実施するために問題を切り分けるために利用することがあります。

今回の Kubernetes の移行において、環境変数で一部のルーティング先を変更しながらアプリケーションに手を加えずに移行を完遂しています。

20　通信経路の切り替え

移行前の状態（Phase 1）から移行後の状態（Phase 2）までのステップは次のような経路で実施しました。

Phase	経路
Phase 1	Apache → nginx → Container
Phase 2	Apache → istio-ingressgateway → App(Docker Swarm)
Phase 3	Apache → istio-ingressgateway → App(Kubernetes)

Apache による経路変更はパス単位（URI 単位）で実施できるため、流量が明らかに少ないパスを管理する

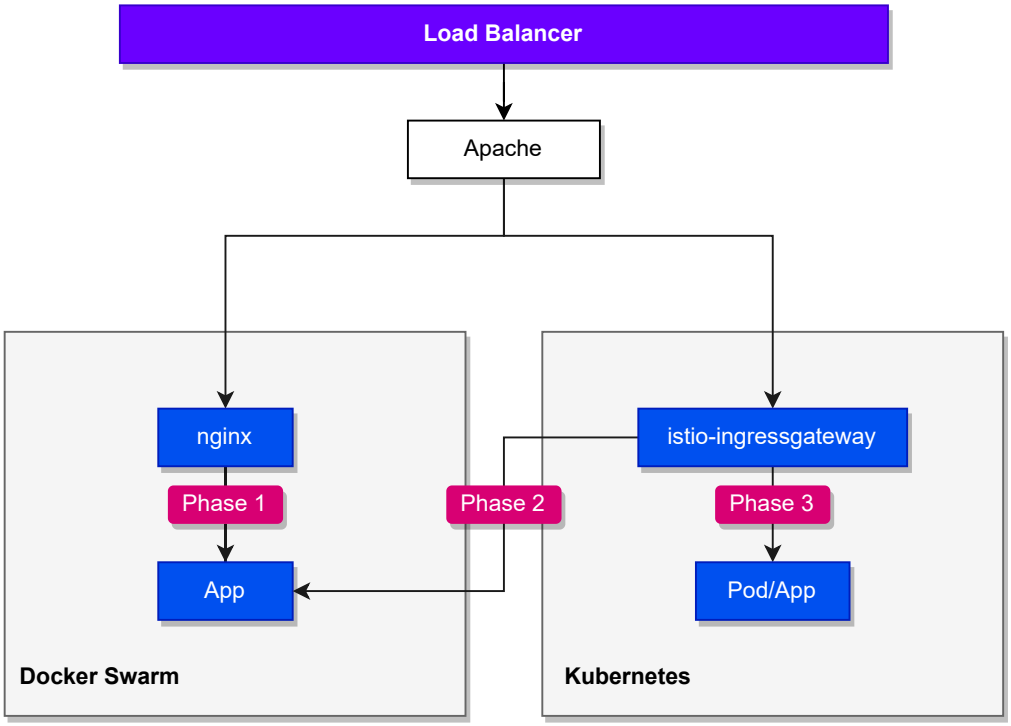


図 24: Kubernetes でのネットワーク

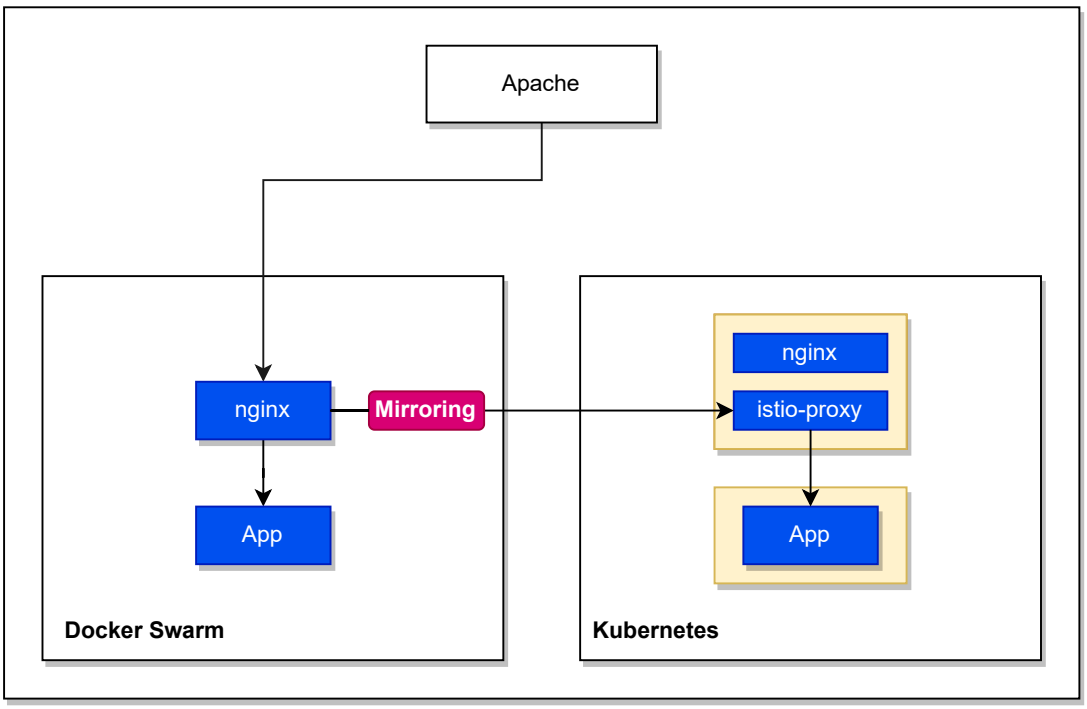
マイクロサービスから移行を実施しました。

20.1 移行の流れ

移行時の細かい手順は次のようになります。

1. Apache の BalancerMember を利用してから Phase 1 から Phase 2 に切り替え
2. istio-ingressgateway と Kubernetes 内のネットワーク系の状態を確認
   - 負荷試験の結果と照らし合わせて Gateway のリソース使用量などを見る
3. Phase 3 への切り替え前に、Virtual Service で特定の Header か Query Parameter を利用して移行後の Pod に対してアクセス。
4. Kubernetes 内への疎通も確認できた後、istio-ingressgateway の Traffic Weight を完全に切り替える

rps がそこまで高くない BFF はこの手順を繰り返すことで移行を淡々とすすめることができました。 高 rps の BFF サーバーはこの手順でやるにはリスクが高いので、トラフィックのミラーリングを実施して Gateway と Kubernetes クラスター全体の状態を確認していきます。アクセスログで紹介したように Pod にログ出力のための `nginx` が含まれるため、二重計上されないために `NodePort` を `istio-proxy` に向けたものをミラーリングのためのポートとして提供しています。nginx のミラーリングによって高 rps の時間変化が DataDog に蓄積され、そこから対応表を用いてリソースの逆算を実施し、移行フェーズへステップを進めることができました。



20.2 ロールバック設計

Phase 1, 2, 3 で移行ステップが区切られているのはロールバックのためです。Istio の Virtual Service や Gateway に指定するパラメーターは Allow List 形式であるため、明示的に指定しなければ疎通が取れません（全部通す設定も可能ではある）。ゆえにアプリケーション側で必要な URI が開放されていない場合などにエラーが発生するため、ロールバックする可能性が十分にありました。即時性を考えた結果、Phase1 と 2 の状態を用意することで影響範囲に応じて即ロールバックできる状態にすることで落ち着きました。

結果は何度か Phase 1、2 の状態に戻すことはありました。ただ、これによって得られたものは、Manifest にアプリケーションのルーティングの仕様が明示的に記述されるようになり、忘却されにくい状態になりました。

20.3 スケジュールの振り返り

時期	内容
2021/07	Kubernetes の移行作業着手
2021/12	Production 環境での Kubernetes 移行実施
2022/03	Production 環境での移行作業完了

この間、Kubernetes 自体や Argo 系のアプリケーションの更新も実施ししています。

リリース時期	リリースされたもの
2021/04/09	Kubernetes v1.21.0
2021/08/05	Kubernetes v1.22.0
2021/08/20	Argo CD v2.1.0

リリース時期	リリースされたもの
2021/10/13	Argo Rollouts v1.1.0
2021/12/08	Kubernetes v1.23.0
2021/12/15	Argo CD v2.2.0
2022/03/06	Argo CD v2.3.0
2022/03/22	Argo Rollouts v1.2.0

※ minor バージョンは省略

20.4 移行中・移行後の運用

移行期間中、ArgoCD の GitOps に則り、Manifest を管理する Pull Request を担当者が出す方式を取っていました。しかしながら、高頻度で更新されるアプリケーションはこれが手間であるため、Slack からリリースに必要な準備一式が整うように調整しました。

Docker Swarm からのデプロイ手順は Slack 上でコマンドを打つことで準備ができるようになり、Kubernetes どころかリポジトリそのものを意識することが減りました。より詳細は Slack Bot による自動化に書いています。

20.5 まとめ

仕事は段取り八分とよくいったもので、移行に要した時間のほとんどが検証作業に費やされています。Kubernetes に加え、サービスメッシュの導入によって Observability が向上し、リアルタイムで多くの情報が得られました。移行期間中もリソース消費の予測がかなり簡単にでき、定量的に決定できたことは今後もデプロイを確実かつスムーズにするのに大いに役に立つと考えられます。