ネットワーク運用自動化のための サービス・運用設計

S4 ようこそ、ネットワーク運用自動化の世界へ！

インターネットマルチフィード株式会社 技術部
川上 雄也

テーマ

運用自動化を可能にするサービスと運用の設計

運用自動化が可能な組織を実現するための取り組み

自動化に耐えうるサービス設計概論

自動化と機械処理

自動化とは、人手でやっている作業を
人が介さずに完了できるようにすること

コンピュータによる機械処理が必要

- 機械処理
  - ルールとアルゴリズムに基いて処理が実行される
  - 例外処理に弱い
    - 例外処理は全て処理方法を定義する必要がある

コンピュータは
「よしな」には
やってくれない

これまで人間が臨機応変にやっていた
部分までしっかり決める必要がある

自動化とサービス・運用設計

超重要

ネットワーク運用を自動化するためには
機械処理に適するような、サービスと運用の設計が必要

決まったルールが一貫して適用可能であること

- サービス設計
  - サービスの技術的な仕様
- 運用設計
  - サービスに対する運用ポリシー
  - サービスや設備の運用の方法

サービスの設計

お客様に対して提供するサービスの技術的な仕様を
サービスの要求仕様に基いて決定する

- 提供するサービスを分類する
  - 動的IPのサービス、固定IP1つのサービス、固定IP8つのサービス、等
  - どれがサービスの区分で、どれがオプションなのかはっきりさせる
- サービスに付属する要素を明確に定義する
  - IPv4アドレスは固定で１つとIPv6アドレスは/64を１つ
  - DNSの逆引き移譲はオプションで…
- サービスの中でルールを統一する
  - 動的IPのサービスのIPの払い出し方は…
  - このサービスの接続方法はPPPoEで…
- 例外要素を明確にする
  - このお客様は動的IPのサービスだけど特定のIPを割り当てる約束になっている

運用の設計

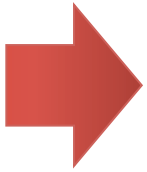
サービスやサービスに必要な設備の運用の方法を
運用ポリシーに基いて決定する

- 運用対象の単位や階層構造を決定する
  - どういう場合に何を１単位として扱うのか？
- 運用対象の名称を決定する
  - descriptionをどうするのか？
- 運用のためのリソースのアサイン規則や連携規則を決める
  - IPアドレスの割当は規則的にできるのか？
  - どこにどのサーバを置いて、何の機器と連携するのか？
- どうやって機器を制御するのかを決める
  - 手順書書いて手動？承認プロセスは？自動コンフィグ？どこまで自動にしていい？
- アラートの引き金、発出方法、対応方法を決める
  - 何をきっかけにどういうアラートを出してどう対応するのか？

運用自動化にあたってのポリシー設計

自分たちの運用スタイルやポリシーを鑑みて
何を目的として、何をどこまで自動化するのか考える

- 機械が生成するもの、実行する箇所
- 人間が目で見て確認する場所、手で実行する箇所
- 例えば…
  - アラートに対して即時対応が必要なので自動化する
  - 作業が定型化されていて、かつ安全なので作業を自動化する
  - 人間の目でチェックすることで品質を担保するので自動化しない
  - コンフィグは自動で生成するけど、確認とデプロイは手動でやる



コンフィグ自動生成＆目視確認＆手動デプロイを
基本にするのがおすすめ

「運用」自動化の大きな特徴

超重要

コンピュータが処理するために Machine Friendly に
オペレータが運用するために Human Readable に

- コンピュータが処理するためには何かの識別子があればいい
  - RDBのIDに使われる整数型とか
  - gitのcommit IDに使われるハッシュとか
- しかし人間であるオペレータも運用をしなくてはいけない
  - 無意味な値や文字列では運用できない！
  - 人間にとって意味のある識別子を付けなくてはいけない

設計に必要なスキル

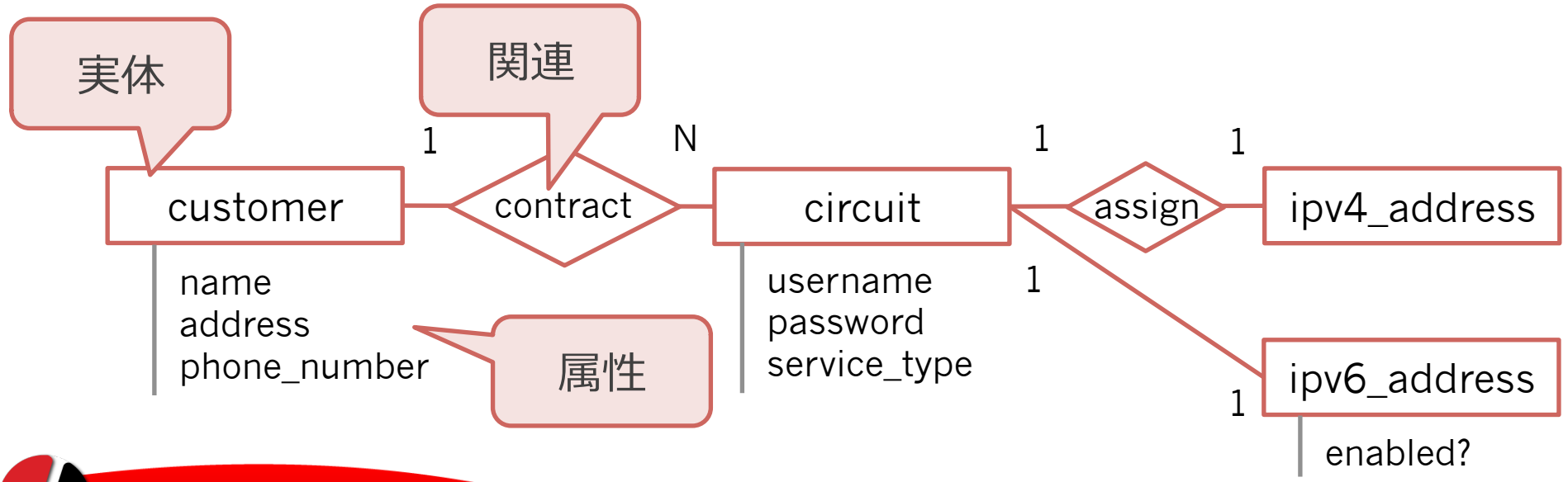
- 自社ネットワークのあらゆるオペレーション経験の網羅性
  - 自分のネットワークの運用がわかってないのに設計はできません
- 将来のサービスや運用の変化を見通す想像力
  - 将来の変化を見越して設計しないと後の人が苦労します
- オブジェクト指向の理解
  - コードに落とす際には、クラスやインスタンスの概念の理解が必要です
- ERMやリレーショナルデータベースに対する理解
  - モデル化やそのあとのデータベース化には必須です
- プログラミングの経験
  - 簡潔な機械処理を実現するための設計に必要です
- 例外処理を作りたくない怠惰さ
  - 楽するためなら苦労を厭わない！そんな人が向いてます

サービス設計とモデル化の実践

サービスのモデル化

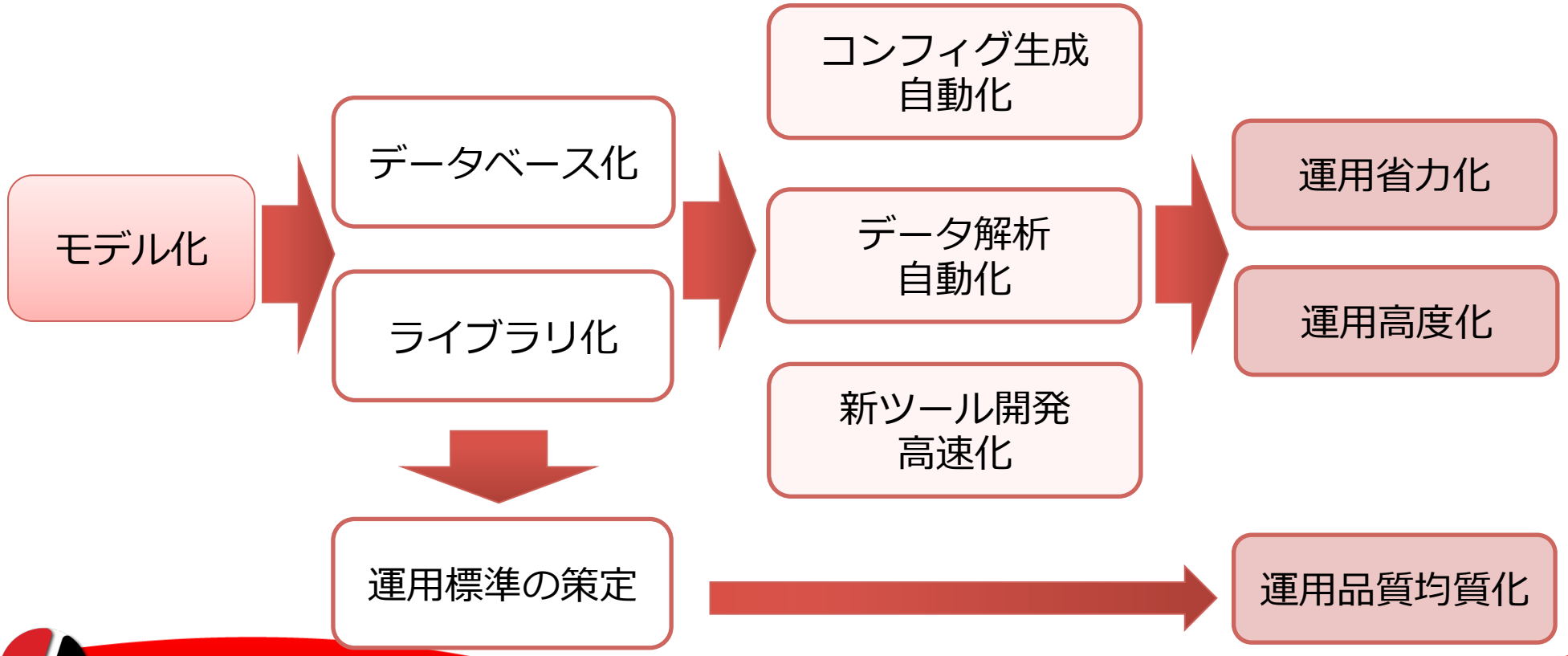
サービスを設計するためにはモデル化が有効
手法として単純な実体関連モデル(ERM)が適している

- ERM（実体関連モデル）
  - データベースを抽象的に表現する手法の１つ
  - 実体(entity)、関連(relationship)、属性(attribute) を考える



サービスをモデル化できると...

データベースの作成やライブラリの作成が楽になり
運用の省力化や高度化につながる



モデル化に必要なこと

超重要

サービスをモデル化して設計するためには
サービスの運用スキームを全て調べあげる必要がある

- 過去に行ってきたオペレーションのパターンを全て列挙する
- 将来起こりうるであろうオペレーションを想像で網羅する

サービスのモデル化（ERM）1/2

- サービスを構成要素(entity)に分解する
  - サービスがどういった要素で構成されているのかを考える
  - Tips: 論理要素と物理要素は分けておいた方がいい
- 構成要素毎の関係性(relation)を定義する
  - サービス上の制約を規定することになる
  - 1:1なのか、1:Nなのか、N:Mなのか、個数の関係や依存関係を気にする
  - Tips: 最初のうちはERM的な関係性の名前は気にしなくて良い（全部hasでOK）
- 構成要素の名前(entity名)を決める
  - 構成要素の名前は全てのツールで同じにできるのが望ましい
  - Tips: おおよその言語で利用可能な、小文字英数字＋アンダースコアが望ましい
    - interfaceやmoduleは一部の言語では予約語なので注意
    - ハイフンでつなぐとプログラミング言語の中では減算の式として扱われることが大半

Tips: ERMの作成

✓ ERMの作成には MySQL Workbench が便利。ドキュメントとして MWBファイルを残すといいかも

サービスのモデル化（ERM）2/2

超重要

- 構成要素の識別子(code)のフォーマット(命名規則)を決める
  - 全ての運用・ツールにおいてこの識別子で対象を識別する
  - 識別子はユニークな文字列である必要がある
  - 識別子は運用者が読んで理解しやすいものにしなければならない
  - フォーマットは機械処理可能でなければならない
  - Tips: idではなくcodeとしたのはRDBで使う数値のidと区別するため
- 構成要素の属性(attribute)を決める
  - 属性もフォーマットをしっかり決める

Tips: 機械処理容易な文字列の例

✓ デリミタで容易に分割できる

```
service_code, customer_code, op_name = “IX1_A001_MF-JPNAP”.split(‘_’)
```

✓ 正規表現で容易にマッチングできる

```
customer_type = $1 if customer_code =~ /([A-Z]{1})([0-9]{3})/
```

✓ 大文字小文字の変換処理が容易

```
service_code = service_code.downcase ※他 upcase,capitalize など
```

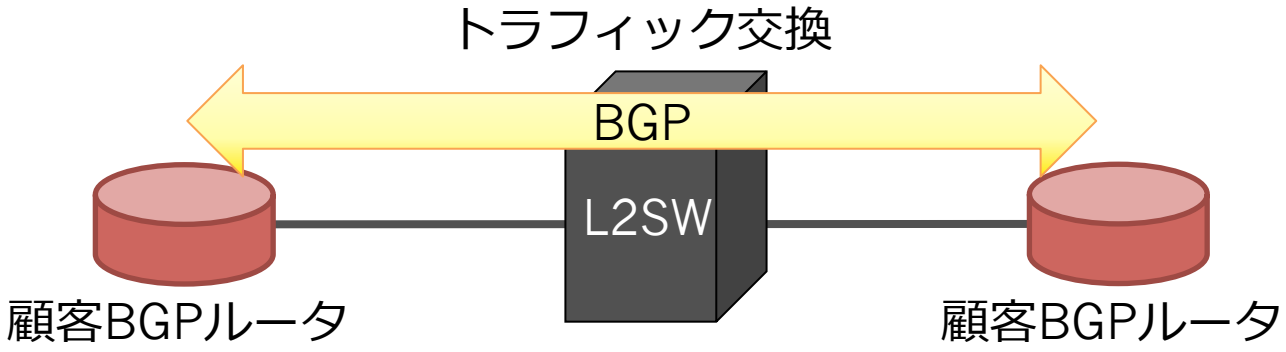
あるIXを例に
実際に
やってみよう

あるIXサービス

- L2 Internet Exchange(IX)サービス
- 東京Ⅰ，東京Ⅱ，大阪の３つのIX
- 100以上の事業者(AS)が接続
  - IPアドレス単位では数百規模の接続

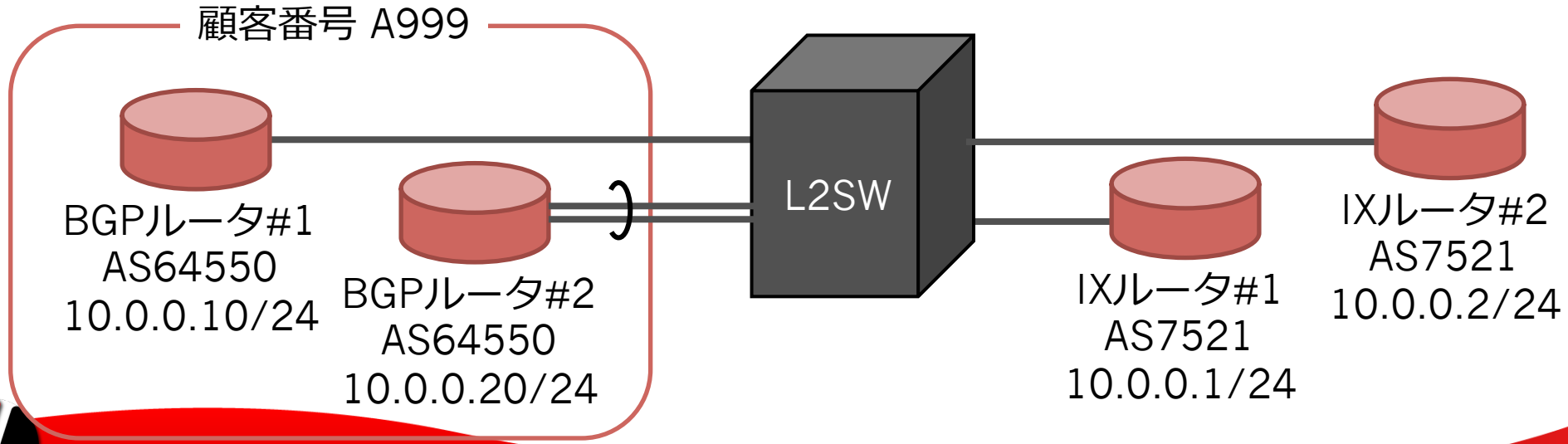
L2IX

- L2スイッチにBGPルータを接続して他の加入者とトラフィックを交換する方式
- １本の回線で複数の加入者とトラフィック交換ができるメリットがある



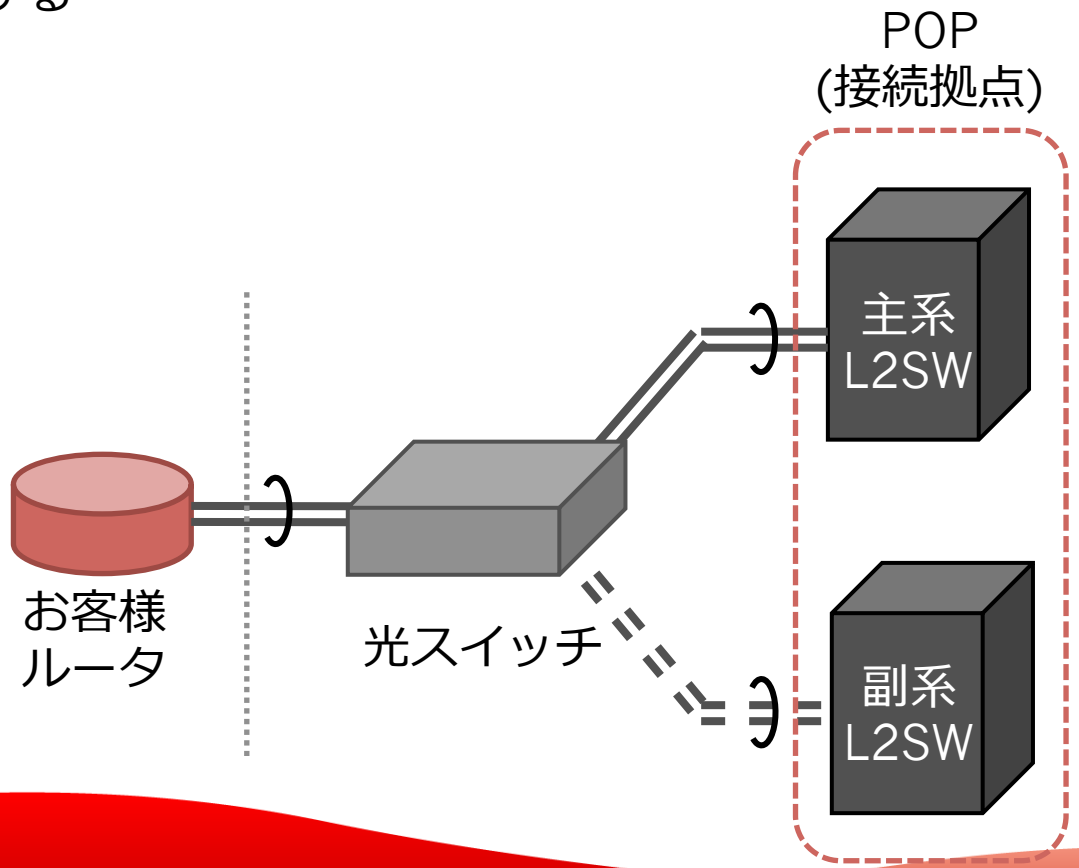
あるIXのサービス仕様要求例

- 顧客はIXサービスのL2スイッチのポートへの接続の買う
  - ２本以上の場合はLAGにする（１本LAGもあり）
- 顧客ルータにはIPv4/IPv6アドレスが１つずつ割り当てられる
- 顧客ルータはAS番号を１つ持っている
- 割り当てられたIPアドレスを使って他のIXメンバーとBGPピアを張る
- IXルータまたはルートサーバ２台とBGPピアを張ってもらう
- 同じ顧客が複数のルータを接続して良い
- 同じ顧客でもルータごとにAS番号は異なるかもしれない

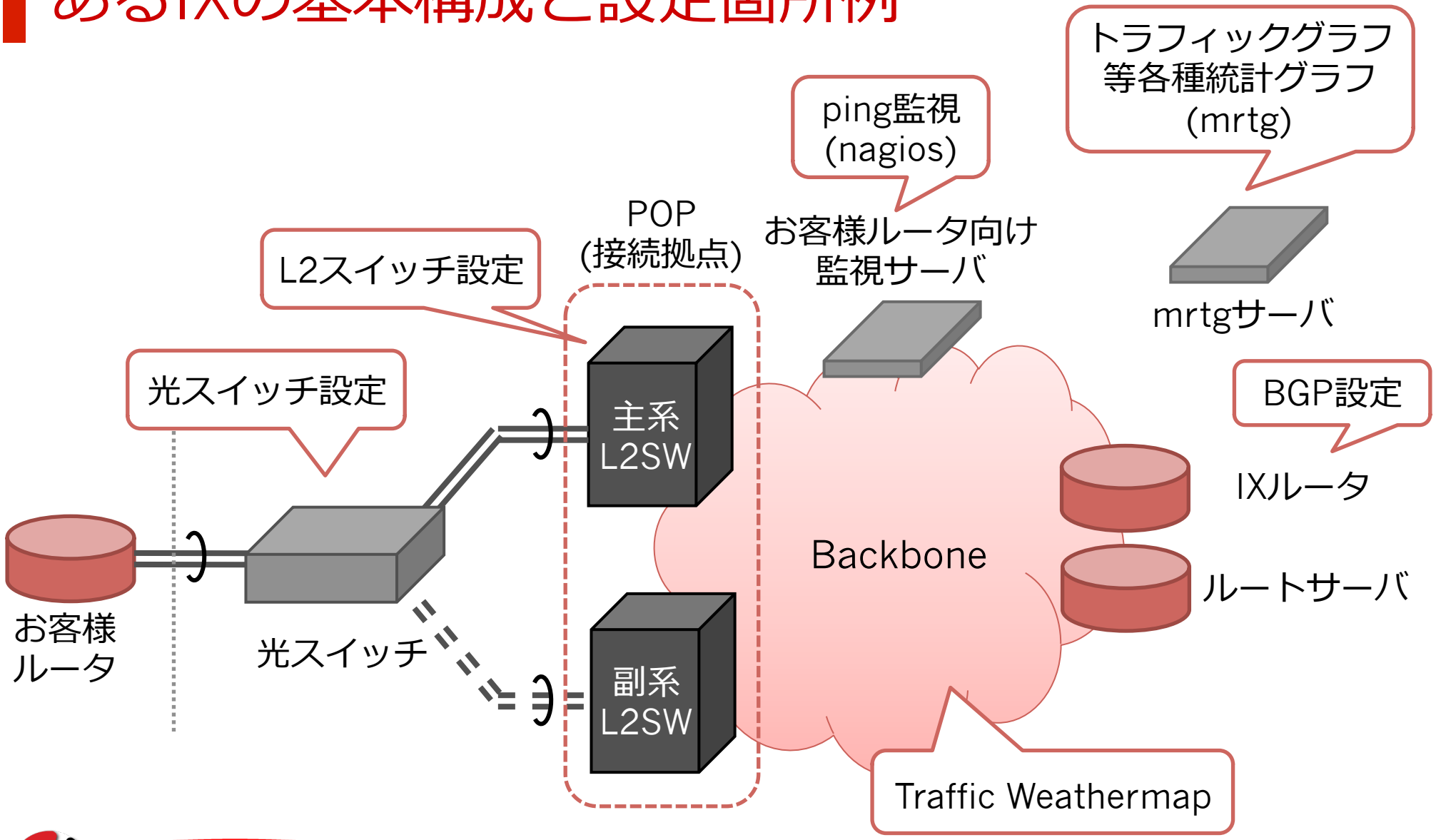


あるIXの運用設計要求例

- 東京が機能しなくなっても大阪だけで運用可能にする（Disaster Recovery）
- 各IXの各接続拠点(POP)には主副のL2スイッチの組がある
  - ただし１拠点に１組とは限らない
- 顧客の回線は光スイッチを介して主副のL2スイッチに収容されることで冗長性を確保する



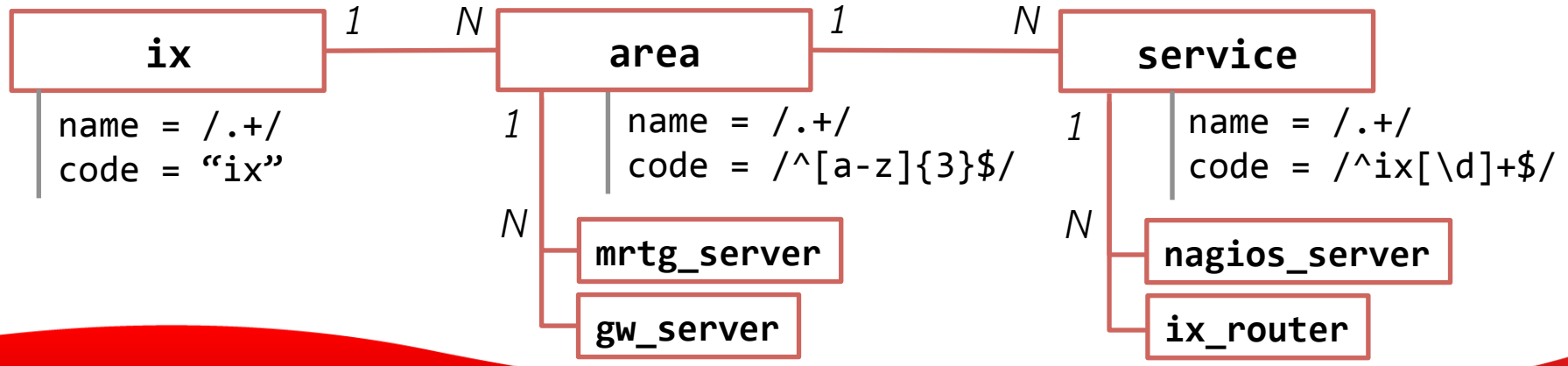
あるIXの基本構成と設定箇所例



実例：Step1 サービス構造のモデル化

要件 東京が機能しなくなっても大阪だけで運用可能にする（Disaster Recovery）

- 東京と大阪で地域を分けて、それぞれに運用に必要な設備を置く必要がある
- 地域を表現する **area** という実体を導入しよう
- **area** の code は /^[a-z]{3}$/ で表現しよう
  - 東京は tky 、大阪は osk にする
- 各IXサービスの実体を **service** とし、code は /^ix[\d]+$/ で表現しよう
- 一部の設備は東京Ⅰと東京Ⅱで共有してもよさそう
  - 踏み台サーバやMRTGサーバは共有可能
  - nagiosサーバやIXルータは各IXサービスごとに必要なので共有しない



実例：Step2 顧客部分のモデル化

要件

顧客はIXサービスのL2スイッチのポートへの接続の買う
同じ顧客が複数のルータを接続して良い

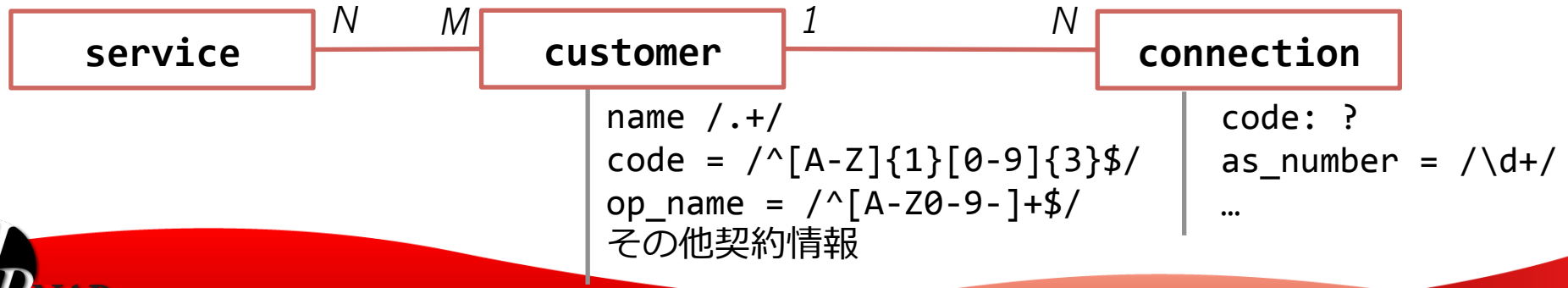
- 顧客を表現する **customer** という実体を導入し、code は /^[A-Z]{1}[0-9]{3}$/ としよう
- codeでは運用しづらいので運用用の名称 op_name = /^[A-Z0-9-]+$/ 属性を付与する
- 接続は **connection** という実体で表現しよう（あとで変わります）
- 複数の顧客が複数のIXサービスに接続できるので **service** と **customer** は N:N の関係
- **customer** と **connection** は 1:N の関係

重要

要件

顧客ルータはAS番号を１つ持っている
同じ顧客でもルータごとにAS番号は異なるかもしれない

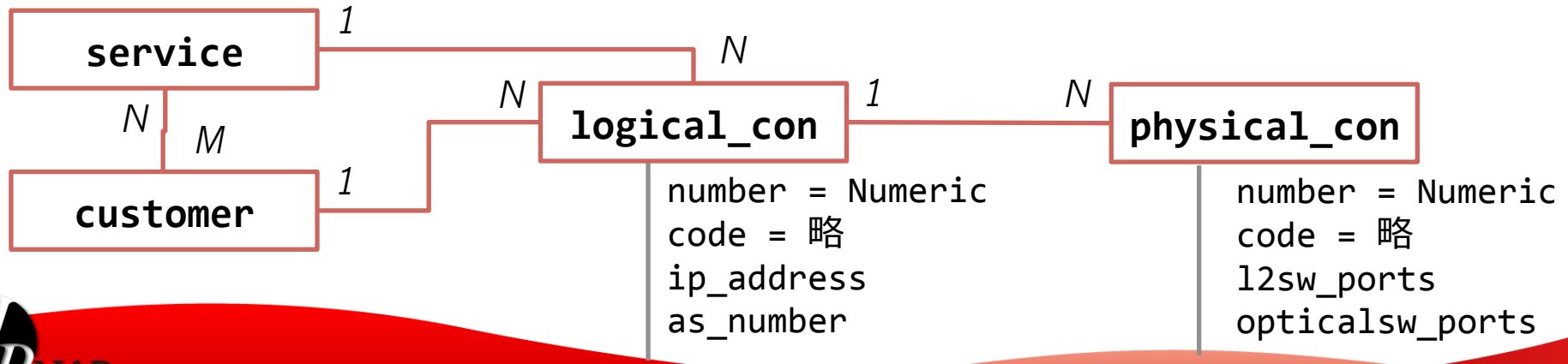
- AS番号 as_number は **customer** ではなく **connection** に関連付ける必要がある



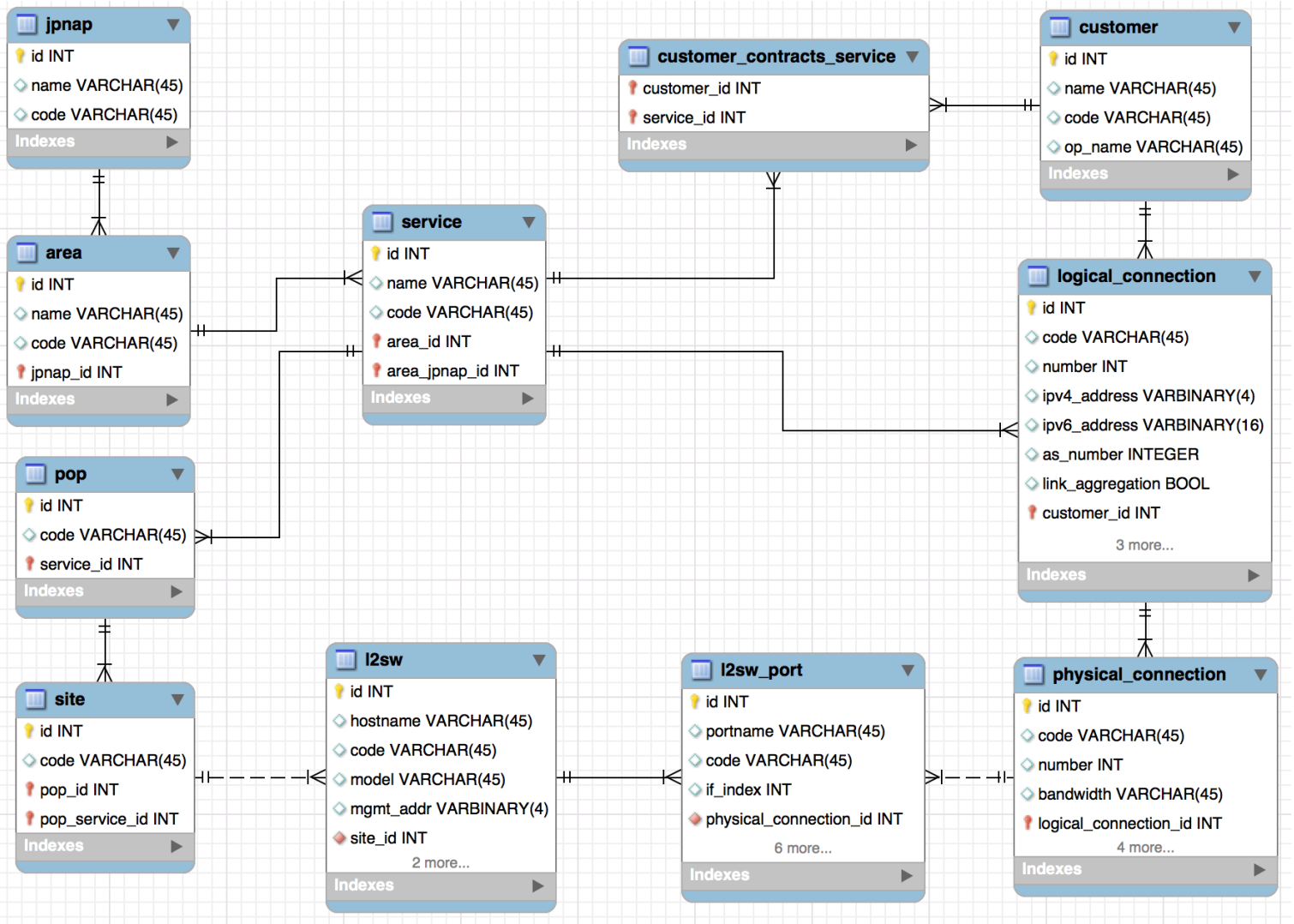
実例：Step3 接続部分のモデル化

要件	顧客ルータにはIPv4/IPv6アドレスが１つずつ割り当てられる 回線が２本以上の場合はLAGにする（１本LAGもあり）

- １つのIPアドレスに対して複数の回線が関連付くので **connection** は論理接続を表現する **logical_onnection** と物理接続を表現する **physical_connection**に分ける必要があった！
- IPアドレスや AS番号などの論理リソースは **logical_connection** の属性にする
- スイッチのポートのや接続速度などの物理リソースは **physical_connection** の属性にする
- 論理的な接続の番号を number で定義し、**logical_onnection** の code を <service.code>_<customer.code>_<customer.op_name>_<number> にする
  - op_name に – が含まれるのでデリミタは – にしてはいけない



MySQL Workbenchで描いてみる



モデルにデータを入れる (サービス構造部分)

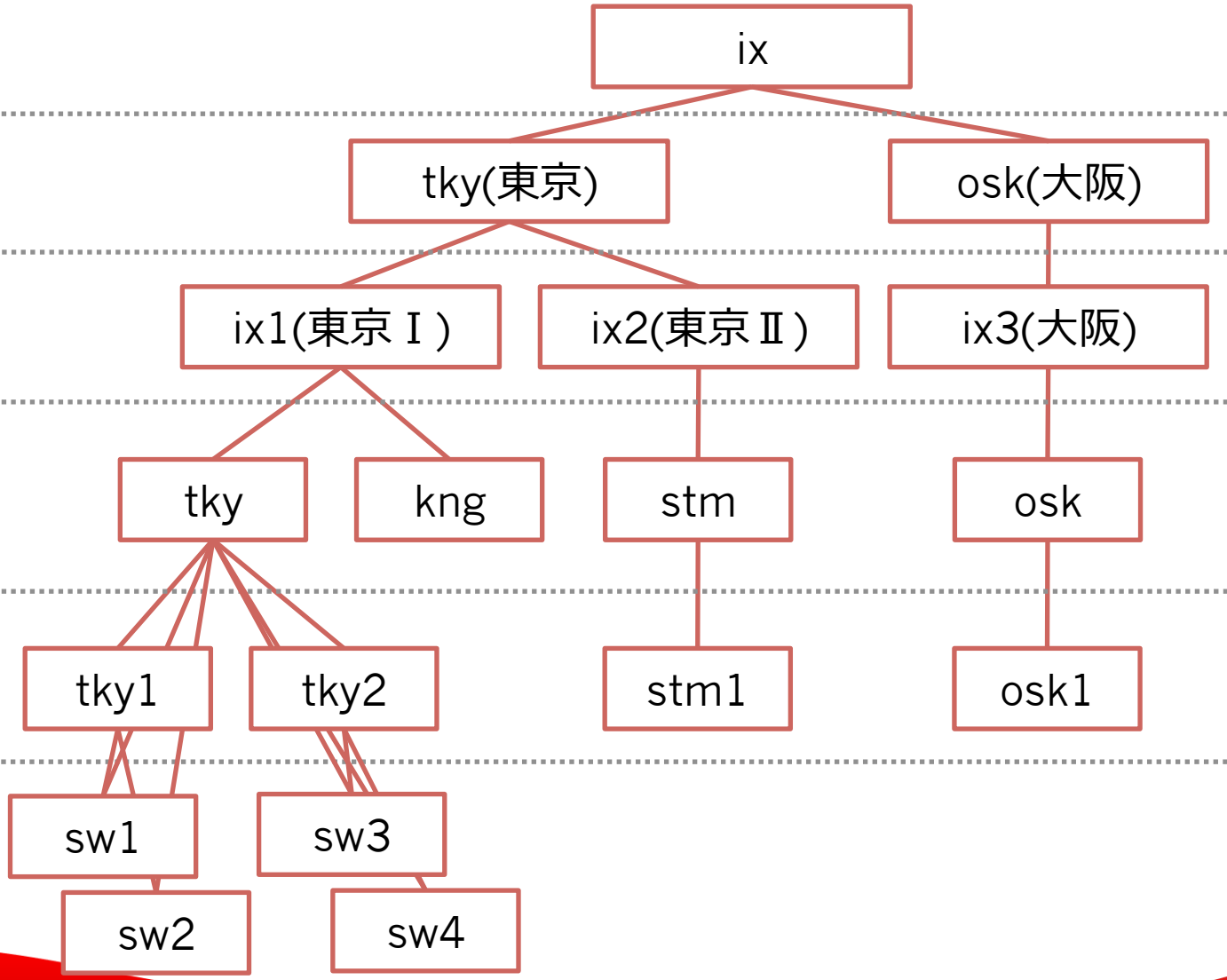
area
code: /^[a-z]{3}$/

service
code: /^ix[\d]+$/

pop
code: /^[a-z]{3}$/
物理拠点

site
code: /^[a-z]{3}[\d]+$/
論理拠点

node
code: /^[a-z0-9-]+$/
L2スイッチ



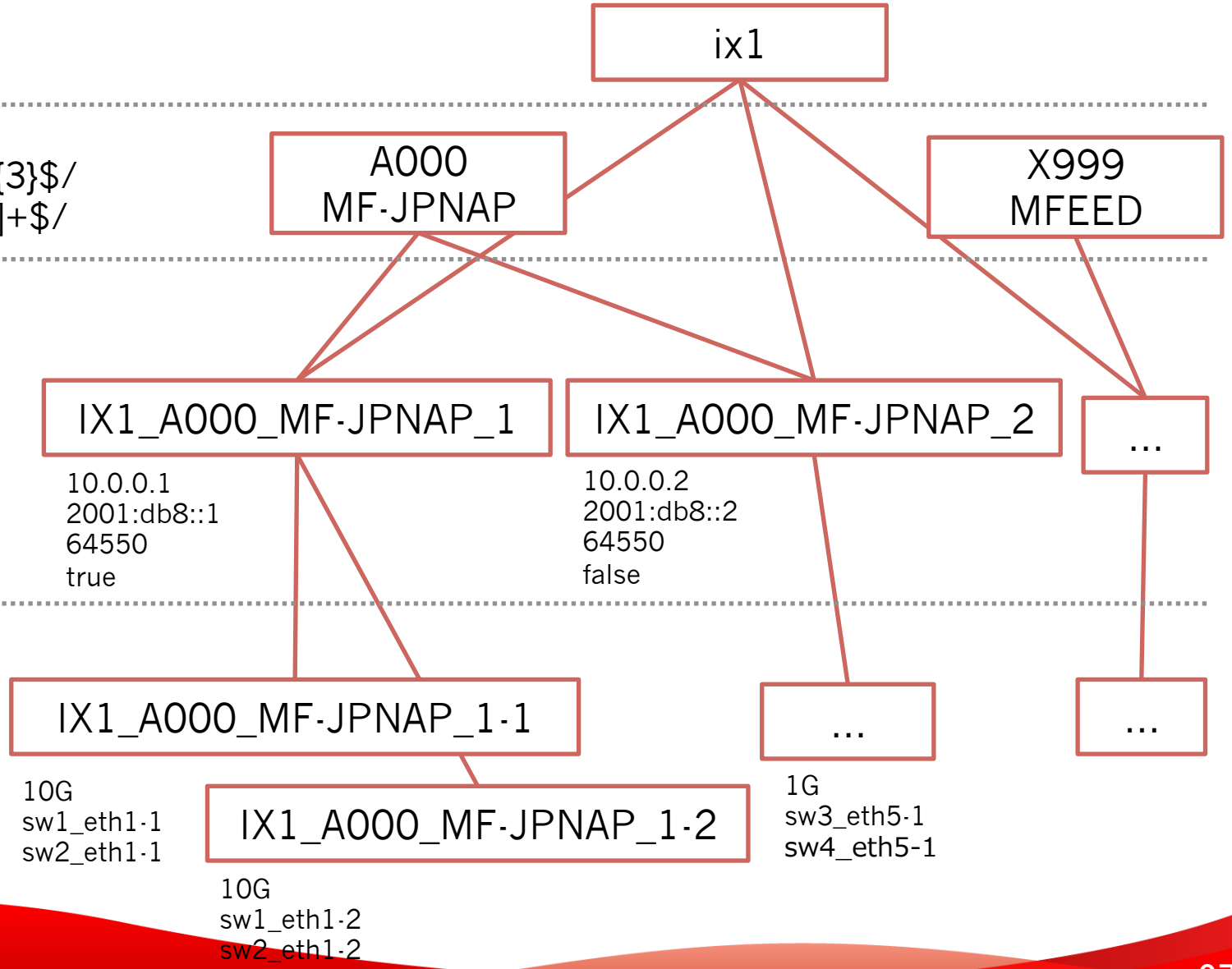
モデルにデータを入れる (顧客接続部分)

service

customer
code: /^[A-Z]{1}[0-9]{3}$/
op_name: /^[A-Z0-9-]+$/

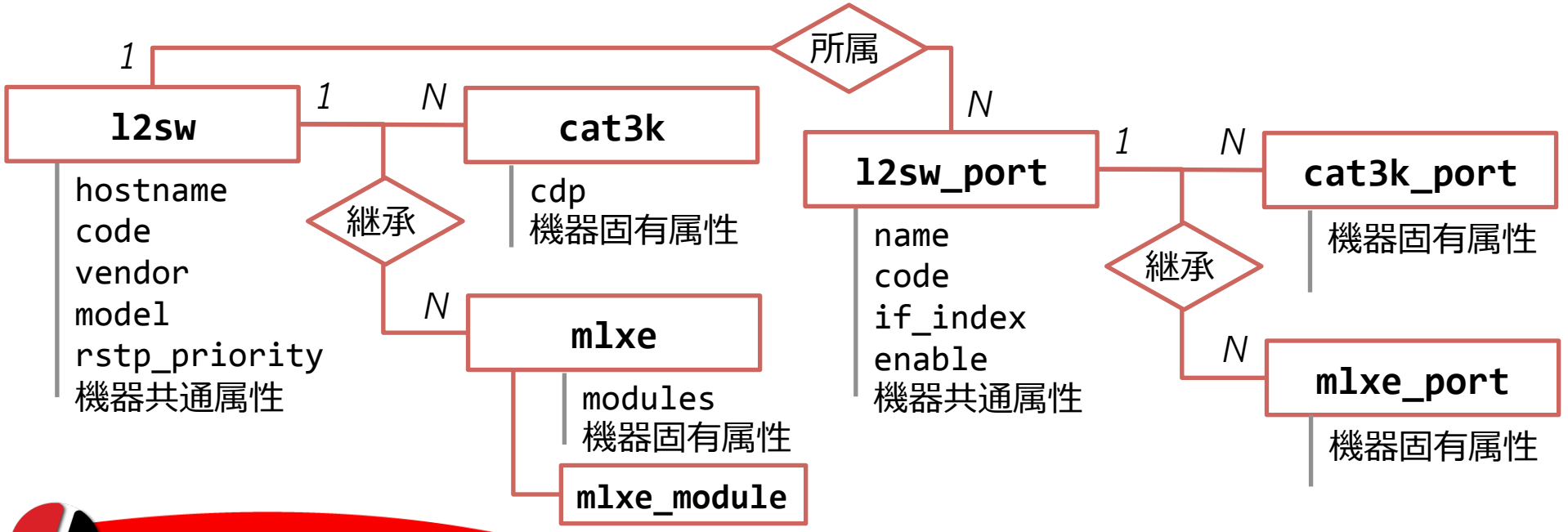
logical_connection
code: 略
論理情報を紐付け
- ipv4_address
- ipv6_address
- as_number
- link_aggregation

physical_connection
code: 略
物理情報を紐付け
- bandwidth
- l2sw_ports
- optical_sw_ports



ネットワーク機器のモデル化

- まず機器の役割ごとに実体を作る
  - L2スイッチ、BGPルータ、光スイッチ、伝送装置、など
- つぎに機種・モデルごとに実体を分けて、役割を継承する
  - メーカごとの違いなどを表現する
  - シャーシ型の機器にはモジュールを表現する実体を作ることなども考えられる
- ホスト名の命名規則をしっかり決める
- コンフィグ生成を見据えてYANG Modelをつかうのもアリ

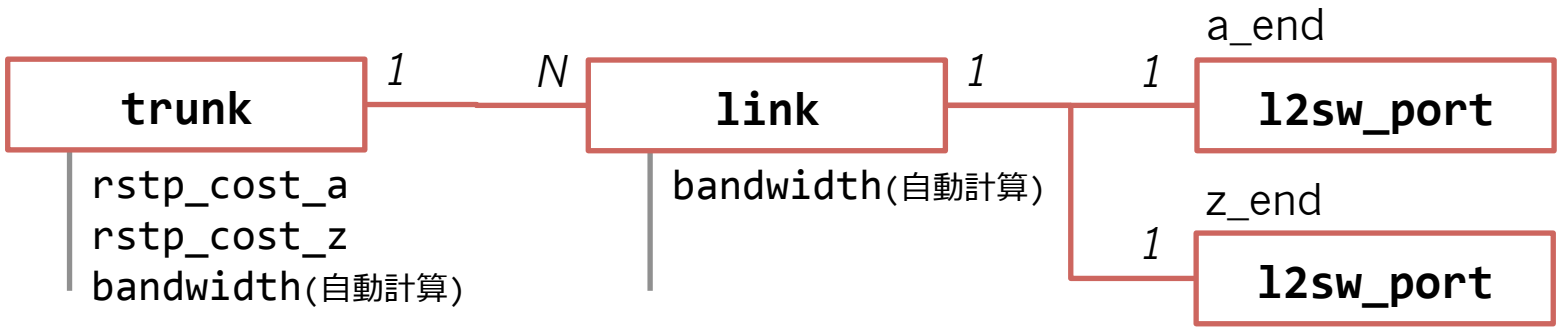


ネットワーク機器のモデルの実装 Tips

- 必要な部分だけ実装する
  - 機器の機能を全てを網羅するのは無理
- プログラミング言語での実装
  - オブジェクト指向言語のクラスの継承を利用する
  - データベースにRDBを使う場合はモデル化通りにテーブルを分けて作成して、プログラムからはORMを利用する
    - Ruby on RailsだとPolymorphic Associationを使うと良い
- バージョンの差異
  - バージョン情報を持たせて、処理が異なる部分だけメソッドを使い分けるのが良さそう
    - show 系コマンドのパースとか
  - メジャーバージョンアップなど、バージョン間の挙動の差異が大きい時は、実体（クラス）自体を分けてしまうのもアリ
- LAGのような論理インターフェイスを扱うために一段噛ませるのもアリ
  - physical_port, logical_port と分けるとか
- 仮想化が出てきてモデル化も難しくなってきた...
  - 複数の物理筐体を１台の機器として扱う: Virtual Chassis、Virtual Switchなど
  - １台の物理筐体の中に仮想機器が複数ある: Logical Router、など

ネットワークのモデル化

- たぶんいちばん知りたいところだと思いますが、事業者によりネットワークの機能は様々なので、一概にこうすべきと言うのは難しいです
- 一般化したモデルよりは、自分たちのネットワークに特化したモデルを組むほうが楽です
  - ネットワークトポロジーの構築規則を決めるのが大事
  - 必要なところだけ実装する
- リンクを主体にして、ポートをひもづけるのが一般的
  - 始点と終点があるので有向グラフになる（結構扱いが面倒）

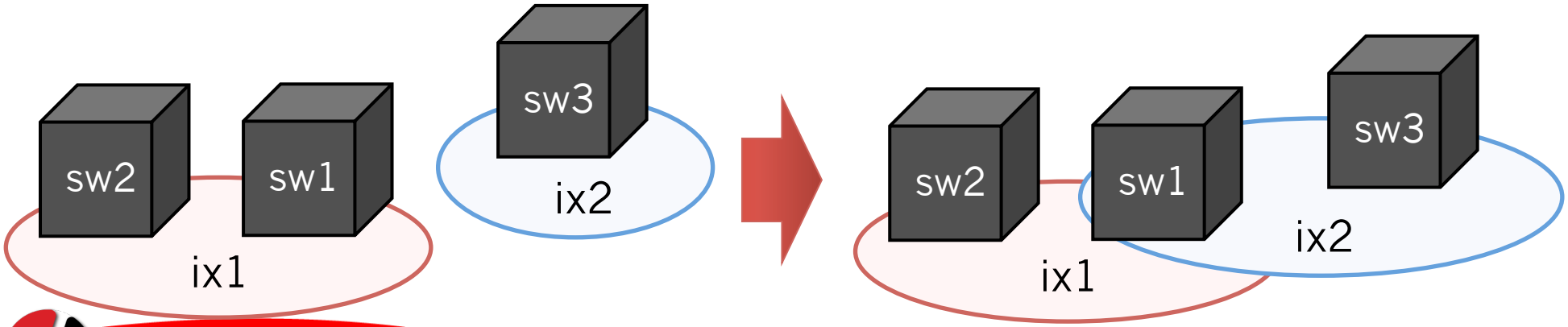


関連付けを失敗した設計の例

1. あるスイッチのMRTGのRRDファイルを設置するパスを決めたい
2. このスイッチはsw1という名前である
3. このスイッチはix1というIXサービスに使っている

```
/var/mrtg/traffic/ix1/sw1/sw1_eth1.rrd
/var/mrtg/traffic/ix1/sw2/sw2_eth1.rrd
/var/mrtg/traffic/ix2/sw3/sw3_eth1.rrd
```

- もしもix2のサービスのセグメントをsw1にも出すことになったら…
  - Tips: スイッチ（物理要素）をサービス（論理要素）に依存させてはいけない



機械処理が難しい設計の例

1. スイッチのMRTGのグラフのファイル名を決めたい
2. 通常の daily と、前週との比較の daily を in 方向について作りたい
3. 対象は ix-sw1 の eth20と ix-sw2 の eth3/1

- ダメな例

```
ix-sw1-eth20-daily.png
ix-sw1-eth20-daily-in.png
ix-sw2-eth3-1-daily.png
ix-sw2-eth3-1-daily-in.png
```

➡ “-” でsplitしたときに意味のある区切れにならない！(文脈依存)

- 良い例

```
ix-sw1_eth20_daily.png
ix-sw1_eth20_daily-in.png
ix-sw2_eth3-1_daily.png
ix-sw2_eth3-1_daily-in.png
```

➡ hostname, portname, graph_type = filename.split(‘-’) できる！

自動化に適した運用にするためのポイント

- 同じものを指すときには同じ言葉を使う（表記ゆれをなくす）
  - 悪い例：MFEED、MF、IMFが全て同じお客さんのことを指している
- 命名規則の遵守を徹底する
  - 区切り文字、大文字小文字
  - 場合分けや例外が発生しないように
- 機械処理可能か常に考える
  - 生成される文字列が常にパース可能か考える
    - 区切り文字で意味のある単位に区切ることができる
    - 正規表現で必ずマッチングできる
  - 文脈依存を起こさない

運用の標準と例外

自分たちのネットワークの運用において
どこまでが標準で、どこからが例外かを定義する

- 例えば…
  - このお客様のルータのping監視はしきい値を変えてある
  - このリンクだけIPアドレスのアサイン規則は適用せず静的に設定する
  - このお客様のこのリンクは監視対象から外す

Tips: 例外の扱い方

- ✓ 例外は例外の種類を定義し、例外専用のデータベースを作り、
  各例外に対してその処理を記述するするのがよい
- ✓ 例外一覧みたいな画面を作れると運用が楽

運用自動化と運用設計のバランス

運用を自動化するために、運用設計を変える発想を持つ

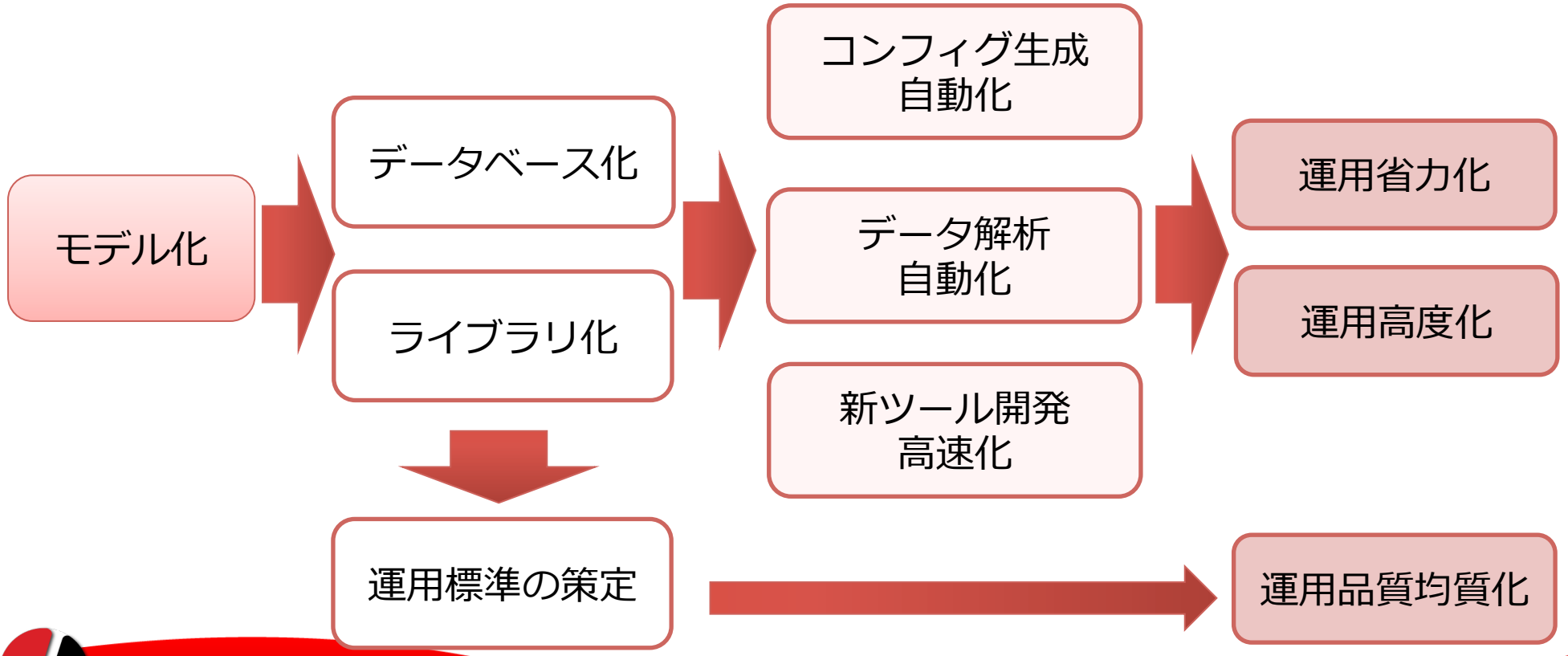
- 例えば…
  - 自動化に向かない機器は調達しない（極論）
    - 調達要件として、ファイルでコンフィグをアップロード→差分だけを commit & rollback ができる機器を指定する
  - モデル化できないネットワークはそもそも組まない
  - モデルに反する要求は全力で拒否する
    - といってもリーズナブルであればモデルとして組み込む
- その一方で自動化のためにサービス提供や運用に阻害が発生してもダメ

自動化のための規則遵守と柔軟性のバランスを考える

自動化に耐えうるサービス設計
モデル化のその先に

ひとたびサービスがモデル化できると

データベースの作成やライブラリの作成が楽になり
運用の省力化や高度化につながる



あるIXでの自動化関連の成果

運用省力化

- スイッチ・ルータの定形作業（主にSO）の手順書をDBから生成
- ルートサーバをDBに基いて自動事前設定、activationの手順書を生成
- サーバ関連のコンフィグをDBから生成、ツールでデプロイ
- 特定のアラートに対してスイッチを操作する作業を全自動化

運用高度化

- 顧客の回線状態や統計情報が一目でわかる Dashboad をDBを利用して作成
- アラートを定型化して、監視解除を自動で判別
- アラートにDashboardのリンクを埋め込むことで、障害対応を高速化（予定）

運用品質均質化

- ３つのIXサービスの運用水準を統一（作業方法や監視項目など）

データベース化とライブラリ化

作成したモデルを元にデータベースとライブラリを作成し、
いろいろなツールの開発に役立てる

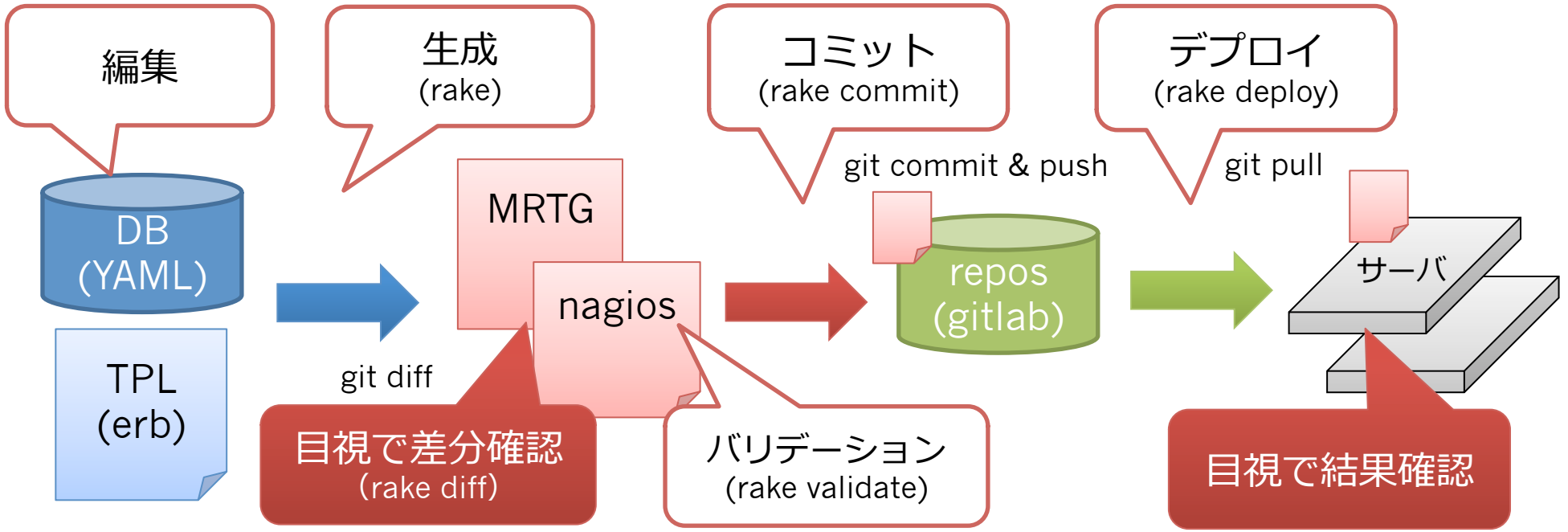
あるIXの事例

- データベース（YAML）
  - モデルに従って実際にデータを登録する
  - git管理したかったのでテキスト形式が良い
    - MySQLやPostgreSQLなどのRDBは却下
  - Machine Friendly かつ Human Readable なものが良い
    - CSV, json などは却下
- ライブラリ（Ruby）
  - データベースを読み込んで、Rubyのオブジェクトを作るライブラリ
  - Rubyを書ける人がチームに多かった
    - Pythonを書ける人がいなかった。Perl, PHP は却下

運用省力化 — サーバコンフィグの自動生成

8種類のサーバ関連コンフィグをDBから生成し、ツールでデプロイ

- コンフィグフォーマットミスや編集作業漏れなどのミスがなくなった！
- 手書きで数時間かかっていた作業が自動化で30分以内に終わるようになった！
- 誰がいつ何を編集したのか履歴が残るようになった！



運用高度化 — Dashboard

IXの統計情報や顧客回線の接続状況などをひと目で把握

IPアドレス
MACアドレス

IP疎通性

BGP状態

トラフィック

L2SW I/F
の状態

L2SWの
I/Fの統計

トラフィック

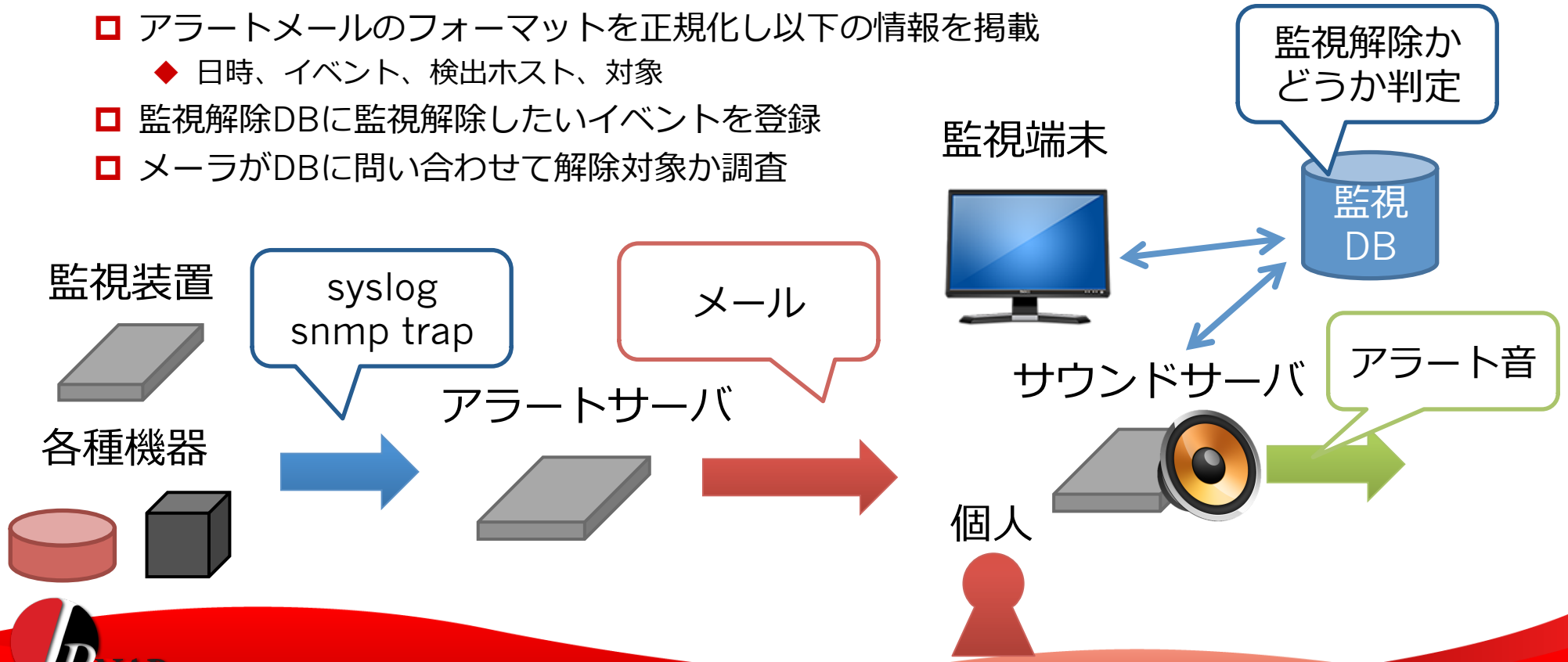
エラーカウンタ

受光レベル

運用高度化 — アラート監視解除自動化

無視するべきアラートを自動判定し、鳴らすアラート音も変える！

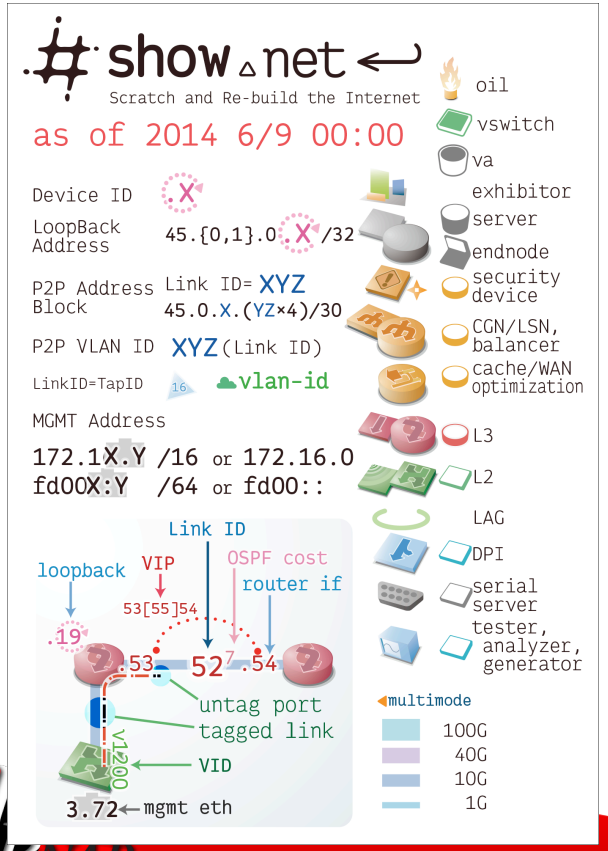
- 方法
  - 全てのアラートイベントの表記を正規化しメールタイトルに付与
    - Link_Up, Ping_Dwon, BGP_Up, 等
  - アラートメールのフォーマットを正規化し以下の情報を掲載
    - 日時、イベント、検出ホスト、対象
  - 監視解除DBに監視解除したいイベントを登録
  - メーラがDBに問い合わせて解除対象か調査

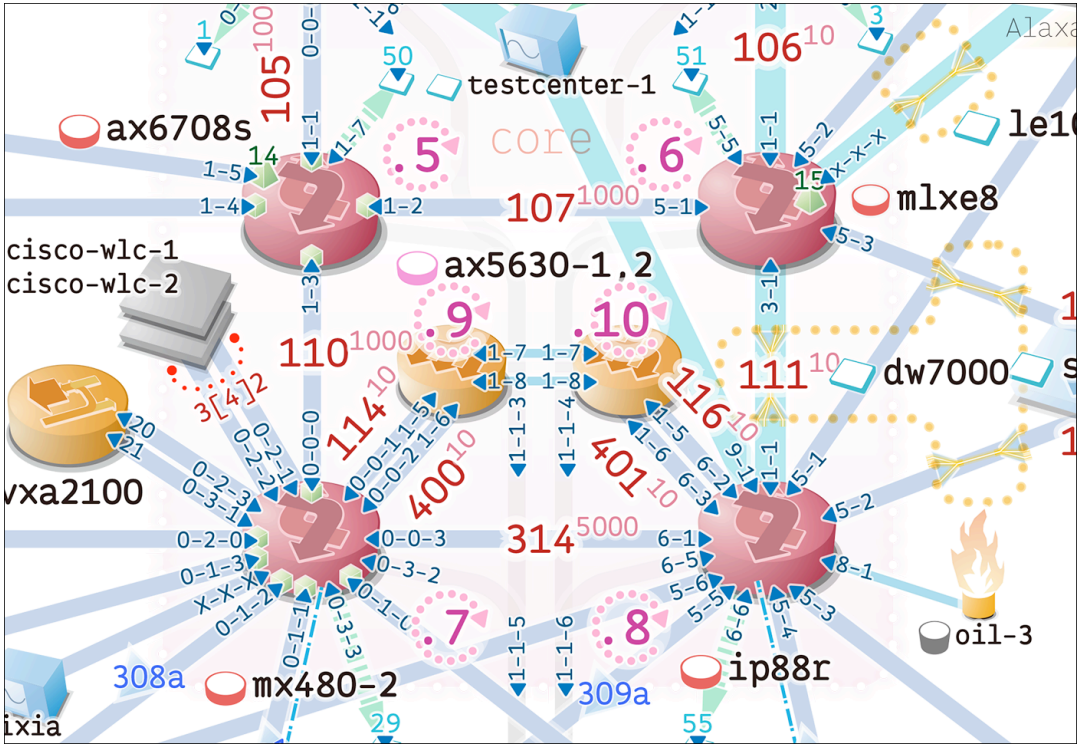


自動化に適したNW設計事例: INTEROP Tokyo

デバイスIDやリンクIDから規則に従ってVLAN番号やIPアドレスを生成可能！

- ただしIPv4のアドレッシングをルール化することは、それなりに大きいアドレス空間がないと実現できない…（IPv6だとかなり楽）





http://www.f2ff.jp/interop/2014/noc/shownet-topology-map_4.php

組織の中で自動化を進めるためのコツ

自動化を認めてもらうために

超重要

徹底的な検証で信頼を得るためにひたすら努力する
自動化のメリット・デメリットをアピールする

- 手でコマンドを打ち、目で確認するという文化が根強いです
- スクリプトで機器を制御するのは、予想外の事態の対応ができないと思われます
  - 例えばエラーが起きているのにそのまま次のコマンドを投入してしまうとか
- プログラムにバグはつきものです
  - バグが起きた時にサービスが停止するような影響がないか証明する
  - 自分のプログラムにバグがあるだけでなく、機器側にもバグがあるかもしれない
- メリット・デメリットをしっかりアピールする
  - メリットだけ言ってもだめです
  - デメリットやリスクをしっかりアピールしましょう

自動化に対する評価

評価されなければ自動化は認められない
ある程度定量的に評価できる方法が必要になる

- 即時性の実現　：作業が完了するまでの時間で評価できる
- スケールの実現：一定期間内に作業が完了した件数で評価できる
- ミスの低減　　：ミスの件数で評価できる
- 作業時間の短縮：作業開始から作業終了までの時間で評価できる

作業短縮時間×作業件数＜開発時間になることも…

- ✓ ミスの低減など他の指標も合わせて評価してもらう
- ✓ 将来的な転用の可能性や開発コストの低減などを評価されるように意識して開発する

自動化のプロジェクトを成功させるために

スキマ時間でプロトタイプを作りデモをする
承認を得ることでプロダクト版を作り込む時間を確保する

- さくっとプロトタイプを作る
  - ツールであれば、想定した簡単な動作だけするものをデモする
  - Web画面であれば、静的なHTMLでコンセプトを提示する
  - 簡単な検証も行う
  - 例外処理などの作業は後回し
- デモで有用だと判断されたらプロダクト版を作りこむ
  - ここで具体的な実装に取り掛かる
  - 例外処理や検証などを実施する

動くプロダクトの作成の時間に比べて、
デバッグやテストの作成には数倍の時間がかかる

自動化したツールを使ってもらうために

徹底した手順化とレクチャーとサポートが必要

- 手順化（マニュアル作成）は必須です
  - 内部動作を覚えてもらうのではなく、まずは手順を完璧につくりましょう
  - このコマンドを打てばorこのボタンを押せば、何が起きてどういう結果が得られるのか、どの結果が得られれば成功なのか、しっかり書きます
- ツールの開発者がサポートできる体制を作る
  - 最初のうちは開発者は、作業の際に作業者につきっきりでサポートする
    - サポートできないうちはツールで作業しない
  - 独り立ちした後も手順書で想定されていない結果が出たらすぐに対応できるようにする
  - この手間を惜しまない！
- 定期的に勉強会を開く
  - 自動化すると中身がブラックボックス化することがよくあります

開発における標準化

チームとして開発ができる体制を整えるために
標準になる開発環境を決める

- 標準のサーバOSを決める（CentOS）
- 標準のプログラミング言語を決める（Ruby）
- 標準のデータベースアプリケーションを決める（PostgreSQL）
- 標準のバージョン管理システムを決める（git）

ポイント

書ける人が多い、触れる人が多いなど、
チームのメンバーが最も親しんでいるものにする

持続可能な開発体制

開発体制が持続可能になるように気を配ることも必要

- メンテ不可能になっているツール、ありませんか？
- １つのツールに対して最低２人の開発者が関わるようにする
  - 本来はマネージャーが気をつける箇所ではある
- コードはすべてリポジトリにつっこむ
  - 社内にgitlabを立ち上げるなど
- コードレビューやペアプログラミング、ハッカソンを積極的に行う
- 新しい開発者の開発の敷居を下げる努力をする
  - 標準として決めた開発方法に沿って開発する
  - トリッキーな実装方法はなるべく避けて標準的な書き方をする（コーディング規約）
  - テストを残す
  - ドキュメントを残す
    - Tips: ドキュメントはREADME.mdなど、ツールと一緒にバージョン管理すべきです

まとめ

まとめ

- 自動化とは機械処理
  - 機械処理可能(Machine Friendly)かどうかを常に意識する
  - 一方で人間の運用のために可読性(Human Readable)を意識する
- 運用自動化のためにはサービスと運用を設計する必要がある
  - サービスの要求仕様に基いて、サービスの技術的な仕様を決める
  - サービスの仕様や、運用ポリシーに基いて、運用方針や設備構成を設計する
- 設計するためにはモデル化を行う
  - サービスをモデルに落としこむ
  - モデル化にはERMが簡単
  - 命名規則とモデルの遵守を徹底する
- モデルに従ってデータベースやライブラリを作成し、アプリケーションを作る
- 自動化を皆に認めてもらうための努力を惜しまない
- 持続可能な開発体制を構築する