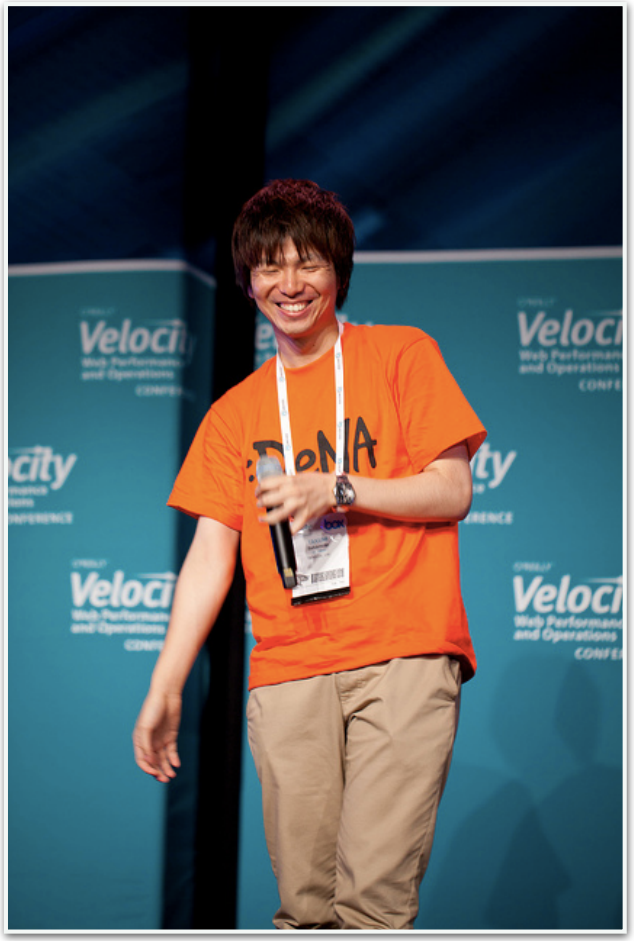
ぜんぶ AWS でやらないワケ

March 15, 2014

Takumi Sakamoto / @takus
IT Platform Dept.
System Infrastructure Management Gr.
DeNA Co., Ltd.

About Me

- 坂本 卓巳 Takumi Sakamoto
- @takus
- Operation Engineer
- DeNA Co., LTD (2012/04~)



私と AWS

- 2009 SOSP の Dynamo の論文読んで感動
- 2011 My365という写真共有アプリを運用

最近は色々やってます

ソーシャルゲーム事業



エンターテイメント事業




ぜんぶ AWS でやらないワケ

メインはオンプレミス環境

- DeNA のサービスの多くはオンプレで運用
- 東日本・西日本の拠点で DR 構成 (Act/Act)
- サーバの台数は数千台規模

コスト/運用で優位なら AWS も

- 海外市場向けのサーバを現地に用意する
- ピーク時の使用率が特に高い一部のサーバ
- 新規サービスで需要が読みにくい場合

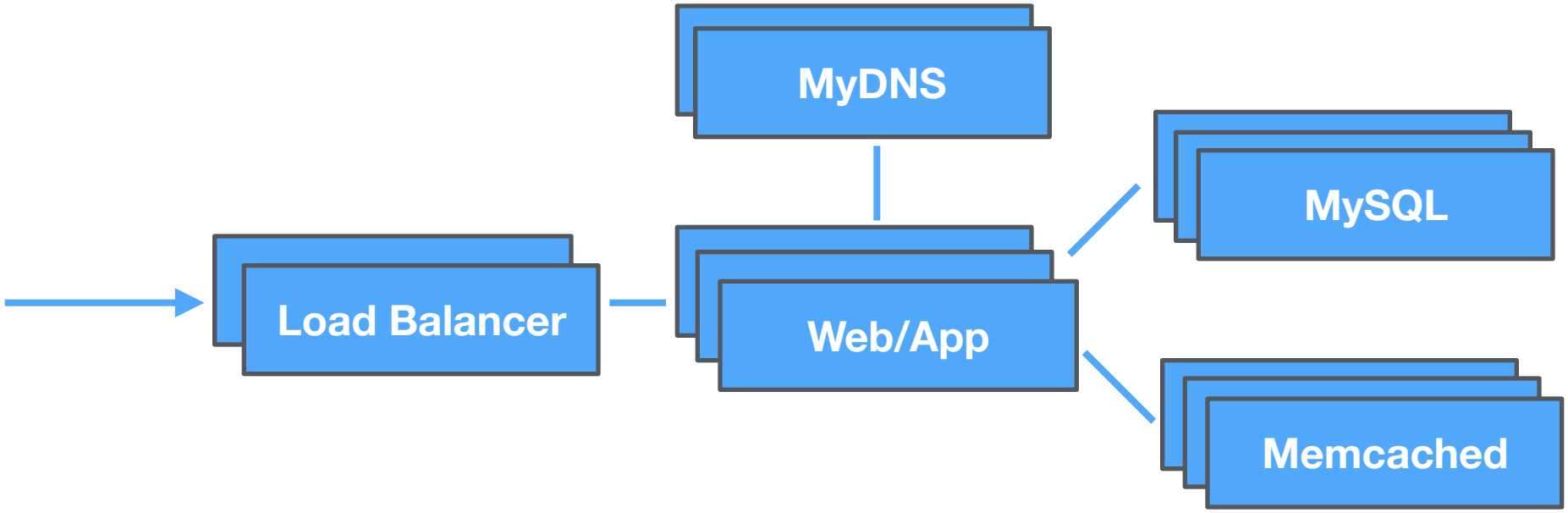
本日お話する内容

- オンプレ/AWS 両方のユーザとして
  - システムアーキテクチャ・オペレーションの紹介
- これで最強の AWS に
  - AWS 使う上でやってることなど
  - AWS で実現できたらいいこと

システムアーキテクチャ

システムの基本構成

- オンプレでも AWS でも基本的な構成は同じ
- Web/Database/Cache の 3 層構造 + α
- db/cache は DNS ラウンドロビンで負荷分散



ロードバランサ

- オンプレでは Hardware LB を利用
- ELB と同様に API でメンバの操作が可能
- 高価だが大量のトラフィックを安定して捌けている
- シェルスクリプトでヘルスチェックとか

ロードバランサ

- AWS ではもちろん ELB を利用
- 安価に利用できてスケールもするが懸念もある
- 突発的なアクセス増加/リアルタイム通信 (後述)

Web/App

- オンプレ / AWS で同構成
  - Web: Apache / Nginx
  - App: Plack / FastCGI (Perl)
- App は内製の WAF で開発することが多い(※)

※ DeNAが開発した新たなフレームワークGunyaSiFとは？

MySQL

- オンプレ / AWS で同構成
  - MySQL 5.1 + InnoDB
  - MHA for MySQL によるマスタの冗長化
  - Q4M や HandlerSocket も利用
- RDS との比較は後半で

Memcached

- オンプレ / AWS で同構成
  - 特殊なことはあまりしてない
  - ElasticCache は使っていない
    - ログが取れないなどのデメリットが多い
  - Redis は Redis Sentinel で冗長化できる

CDN

- Akamai CDN / CloudFront
  - Akamai が多いが CloudFront 使うこともある
  - 価格とエッジロケーションの数などに違いあり

External DNS

- Akamai GTM / Route53
  - オンプレで Akamai、AWS で Route53 を利用
- API の仕様が大きく違うのが特徴的
  - Akamai はゾーン単位で全上書きしかできない
  - Route53 はレコード単位で操作できる

サーバ・オペレーション

基本的な方針

- オンプレと AWS で単一の運用を実現している
  - 人的リソースの流動性が高まる
  - 監視スクリプトなども流用可能
  - ノウハウの共有が容易

Global-Infra Framework

- サーバ・オペレーションのためのツールセット
  - サーバ管理ツール (AdminTool)
  - 各種コマンド (サーバリスト取得、cron ラッパー)
  - 監視ツール用スクリプト (設定ファイル自動生成等)
  - 便利モジュール (logger、アラートメール送信)

オンプレ・AWS で同じ操作が可能

```
# => get active server list
get-hosts -u1 web -u2 ff
ww0001 ww0002 ... ww2046

# => check diff
diff_infra /etc/passwd
/etc/passwd
LOCAL       a030159cf068b727b2a14398dcec6f63
wd0005      a030159cf068b727b2a14398dcec6f63
wd0007      641b11cf9a916ebf4ece86e55a3376c1

# => generate monitoring configurations
make-nagios-conf -t $(get-hosts -u web:ff --delimtier ,)

# => deploy global-infra resources
dist-infra -x
```

プロビジョニング

- コピー元のサーバを複製してサーバを作成
  - コピー元サーバは必要なミドルウェアが全入り
- Touryo (構成管理ツール) でサーバ毎の設定を変更
  - あまり複雑なことはしていない
  - 設定ファイル編集、デーモンの起動など

プロビジョニングの例

```
# => create instance from golden AMI
ec2-create-instances -p ww -t c3.xlarge -z us-west-1b

# => provisioning by touryo(edit config files, start daemons)
touryo run ww2000
ww2000: Some run scripts executed. Test again...
…
    ok 1 - httpd.conf
    ok 2 - daemontools apache
    1..2
ok 10 - apache
…
All tests successful.

# => deploy application
dist-project -x

# => register to load balancer
update-elb -u1 web -u2 ff
```

これで最強の AWS に

AWS を使う上でやってること

- Perl から AWS を操る
- RDS は使わず MySQL on EC2
- 監視は CloudWatch を使っていない
- Auto-Scaling は自前で構築している
- ELB の状態変化を監視

Perl から AWS を操る

- AWS::CLIWrapper を利用
  - 文字通り AWS CLI のラッパー
  - AWS CLI と同じような使い勝手で使える
  - $res = $aws->ec2('describe-instances');
- 詳細は堀内さんの Programming AWS with Perl で

MySQL on EC2

- RDS に比べてメリットが多い
  - MHA で RDS より高速にフェイルオーバーできる
  - ストレージエンジンが自由に選べる

MHA vs RDS Multi-AZ

- 障害時のフェイルオーバー時間の違い
  - MHA は 30 秒程度、RDS は 3〜6 分程度
- メンテナンス性 (スケールアップやスキーマ変更)
  - どちらも無停止で可能
- マスタ切り替え時のレプリカ繋ぎ直し
  - MHA は新マスタに繋ぎ直してくれる

ストレージエンジンが選べる

- Q4M や HandlerSocket が使える
  - 社内に作者がいて問題があればすぐ対応できる
  - AWS なら SQS や DynamoDB もあるけど
- 容量効率のいい TokuDB なども選択できる
  - innodb に比べて 1/29 のサイズになる (※)
  - ユーザアクティビティの保持などに有用

※ TokuDB v6.0: Even Better Compression

CloudWatch のアラーム

- 単純な機能しか提供されてないので使ってない
- サーバが多いとアラート爆発してしまう
  - 全 Web から一斉に CPU の通知が来る等
- 一時的に通知止めるみたいな機能が見当たらない
  - Nagios の Scheduled Downtime 相当

死活監視は Nagios

- 設定ファイルは構築時に自動生成している
- 特定クラスタのアラートをまとめる仕組みを実装
  - CPU アラートはクラスタ毎に 1 通しかこない
- Nagira (NAGlos Restful Api)
  - Nagios のステータスを REST API で取れて便利

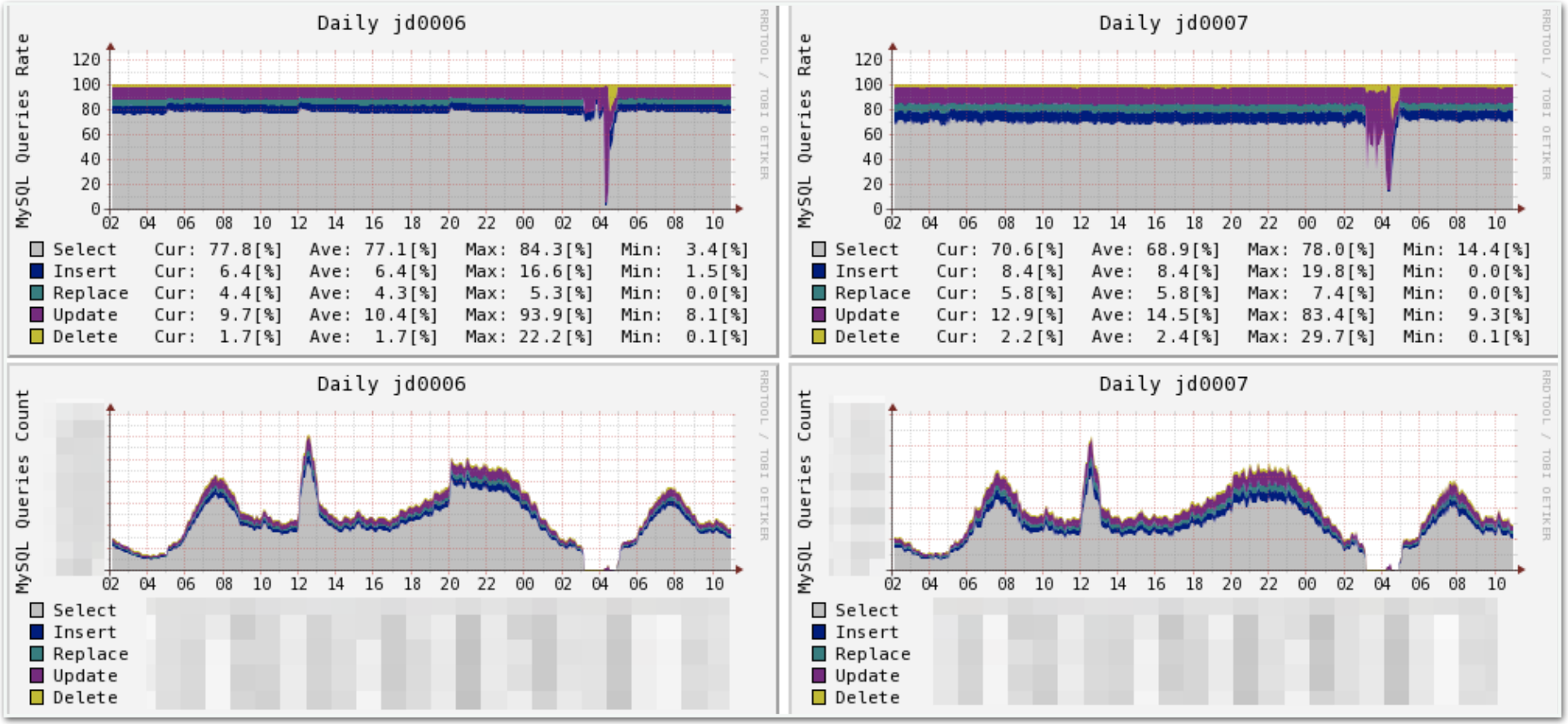
CloudWatch のグラフ

- 使い勝手がよくないので使っていない
  - ２ 週間分のデータしかみれない
  - グラフが一覧できない
  - Name や Tags でメトリクス探せない
  - アカウントが複数あると切り替えが面倒

サーバリソース監視

- CloudForecast
  - CPU / Mem / Disk / DB の QPS など
  - ワーカーを分散処理させてスケールアウトできる
  - 必要に応じて取得するメトリクスを増やすのも容易
    - Porting MySQL graphs from percona-monitoring-plugins #41
    - Add several graphs of redis #42

CloudForecast の例

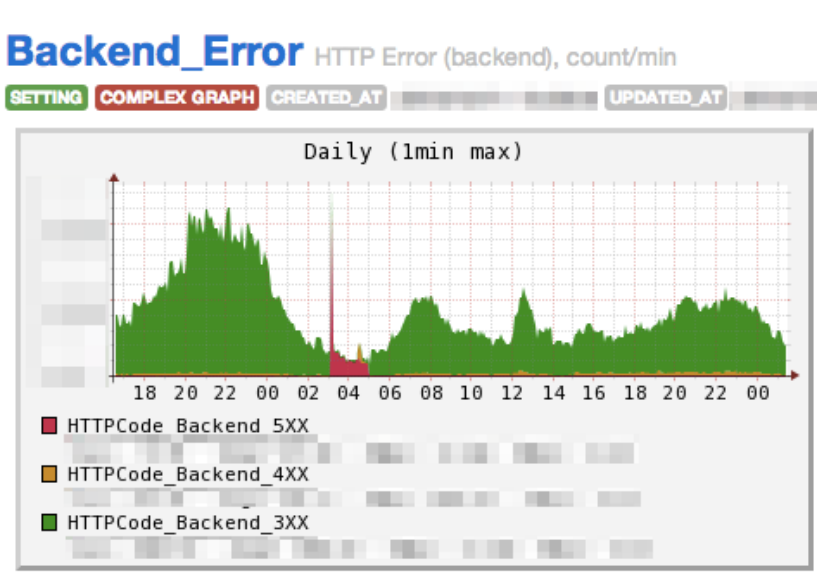


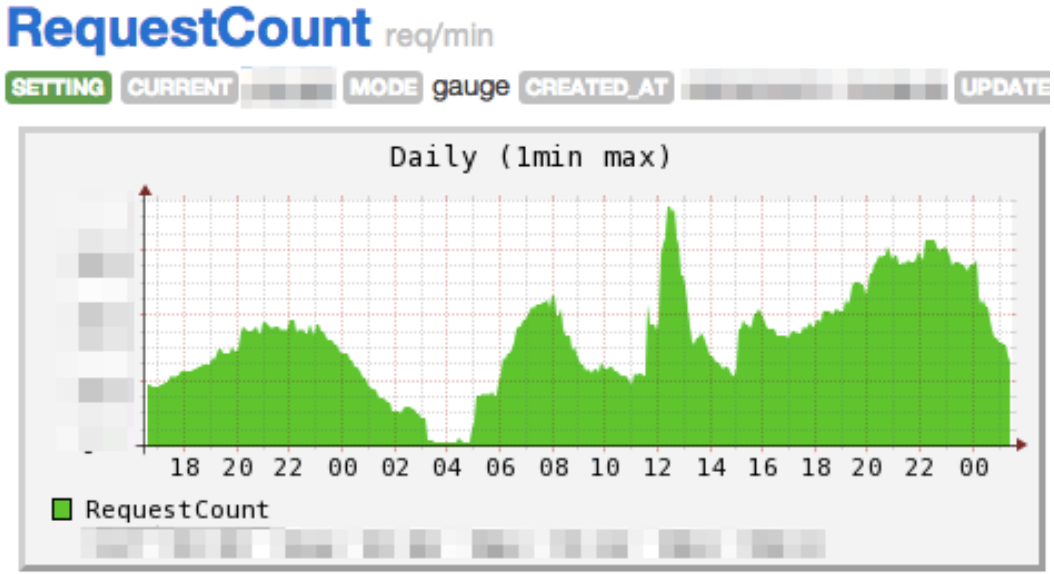
トレンド監視

- GrowthForecast
  - HTTP で値を POST するだけで可視化できる
  - ELB のリクエスト数・エラー数
  - DB の shard 毎のユーザ数の偏り
- Kibana + Fluentd + ElasticSearch
  - 直近のアクセスログ解析に使ってる

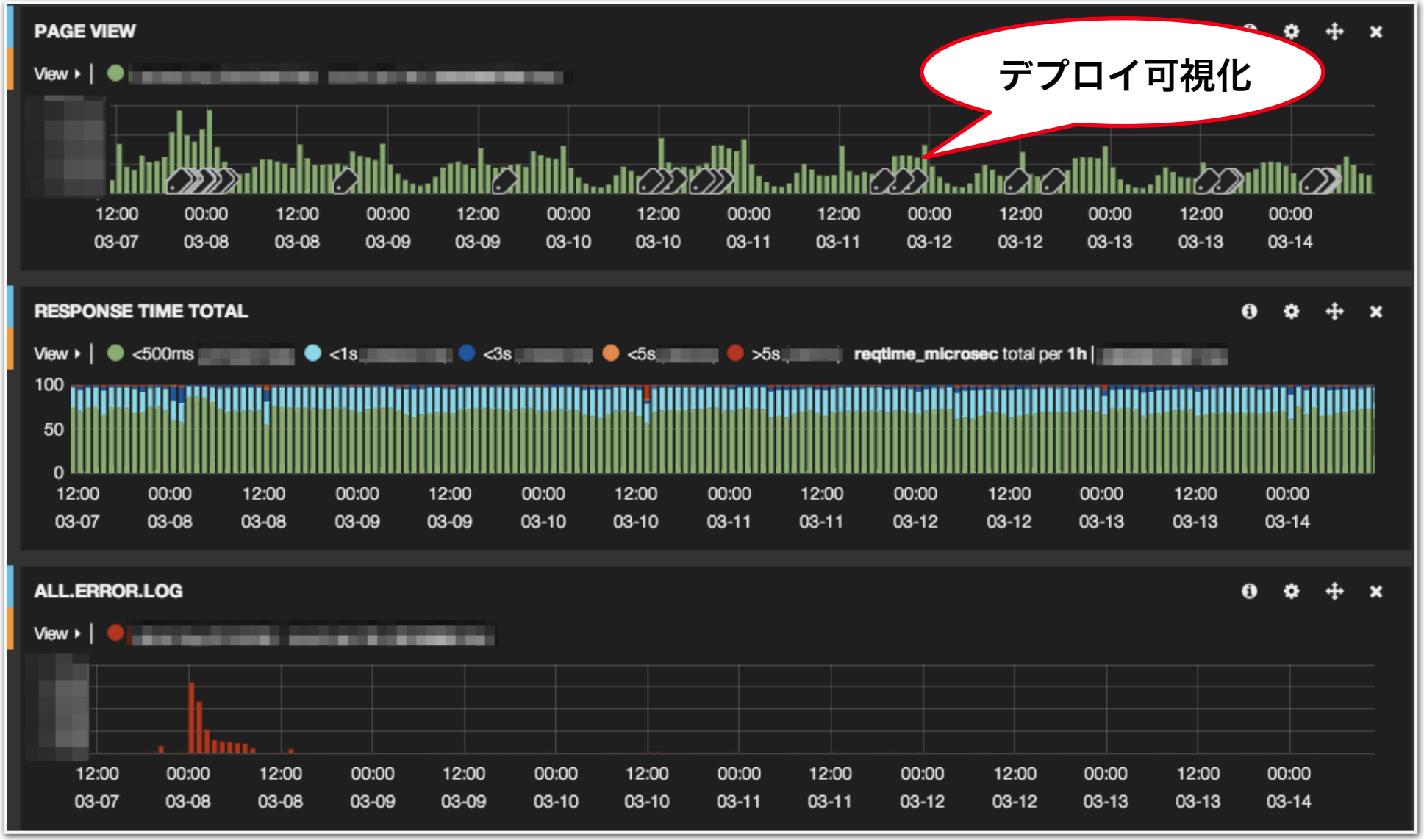
GrowthForecast

- 例: cloudwatch2gf で ELB を可視化
  - GF のいるサーバで daemon 起動するだけ
  - https://github.com/hirose31/aws-cloudwatch2gf





Kibana によるアクセスログ可視化



Kibana で条件の絞り込み

- 特定の URL で絞り込み

```
path: /user/*
```

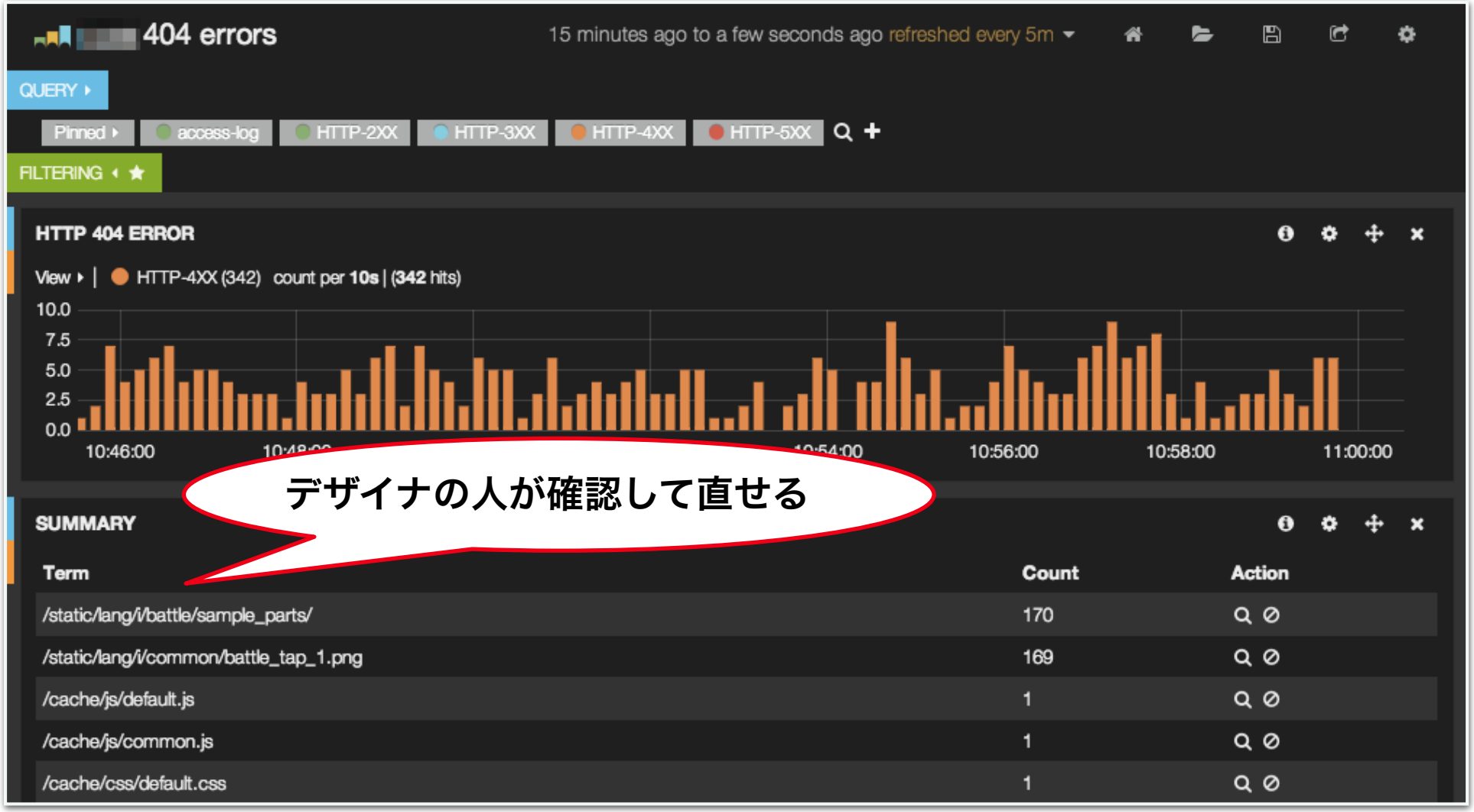
- EC2 の特定タグで絞り込み

```
tagset_project: ff
```

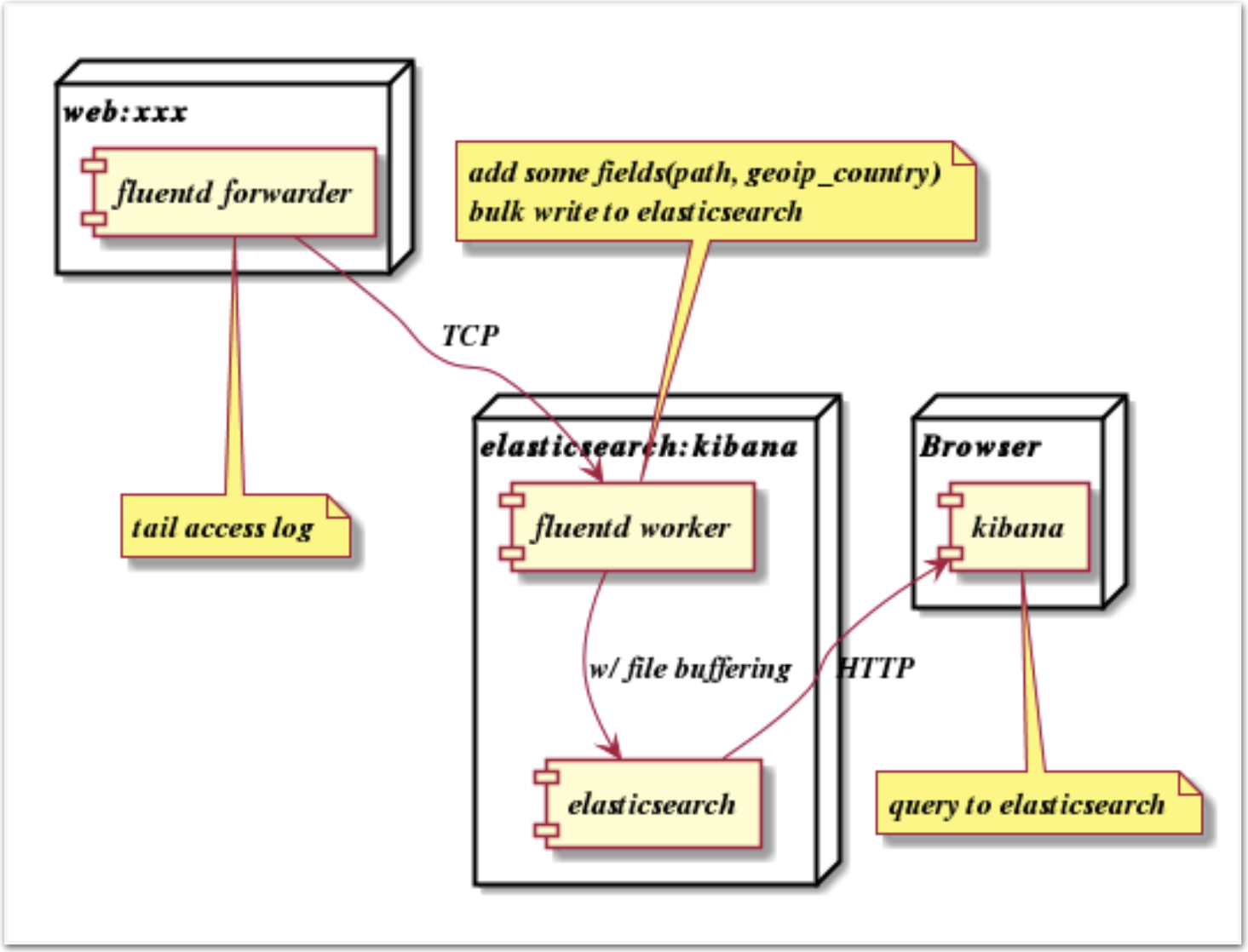
- EC2 のメタデータで絞り込み

```
az: us-west-1b AND instance_type: m1.large
```

Kibana による 404 の可視化



参考: Elasticsearch + Fluentd



参考: Fluentd の送信側

```
<source>
  type    tail_ex
  tag     raw.acc.ff
  path    /var/log/apache/ff.acc.%Y%m%d
  format ltsv
</source>

<match raw.acc.ff>
  type         record_reformer
  remove_keys conn_status,num_ka,pid,user,ua,referer
  enable_ruby false
  output_tag  reformed.acc.ff
</match>

<match reformed.acc.ff>
  type         ec2_metadata
  output_tag  acc.ff
  <record>
    hostname       ${tagset_name}
    project        ${tagset_project}
    instance_type ${instance_type}
    az             ${availability_zone}
  </record>
</match>
```

ログ読み込み

不要フィールド削除

EC2 のメタデータ追加

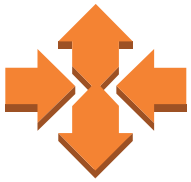
参考: Fluentd の受信側

```
<match acc.**>
  type        record_reformer
  output_tag reformed.${tag}
  path        ${req}
</match>

<match reformed.acc.*>
  type           rewrite
  remove_prefix reformed
  add_prefix     es
  <rule>
  …
  </rule>
</match>

<match es.acc.*>
  type           forest
  subtype        elasticsearch
  remove_prefix es
  <template>
    logstash_format true
  </template>
</match>
```

req フィールドを複製

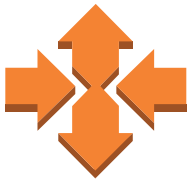
クエリストリング除去

Elasticsearch に保存



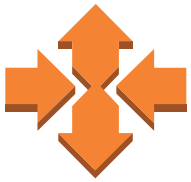
Auto-Scaling β

- 独自の実装でやっている
  - 必要に応じてカスタマイズするのが容易
- 対象は web サーバのみ
- CPU と ワーカーのビジー率が基準



Auto-Scaling のポリシー

- アクティビティが多い時間はスケールさせない
  - 急激な変化にサーバ追加が間に合わない
  - ゲーム毎にフィーバータイムがあり予測が難しい
  - かといって閾値低めにすると無駄が多い
- アクティビティが低い時間に台数を減らす（特に深夜）
  - 突発的にアクティビティが増えることはまずない

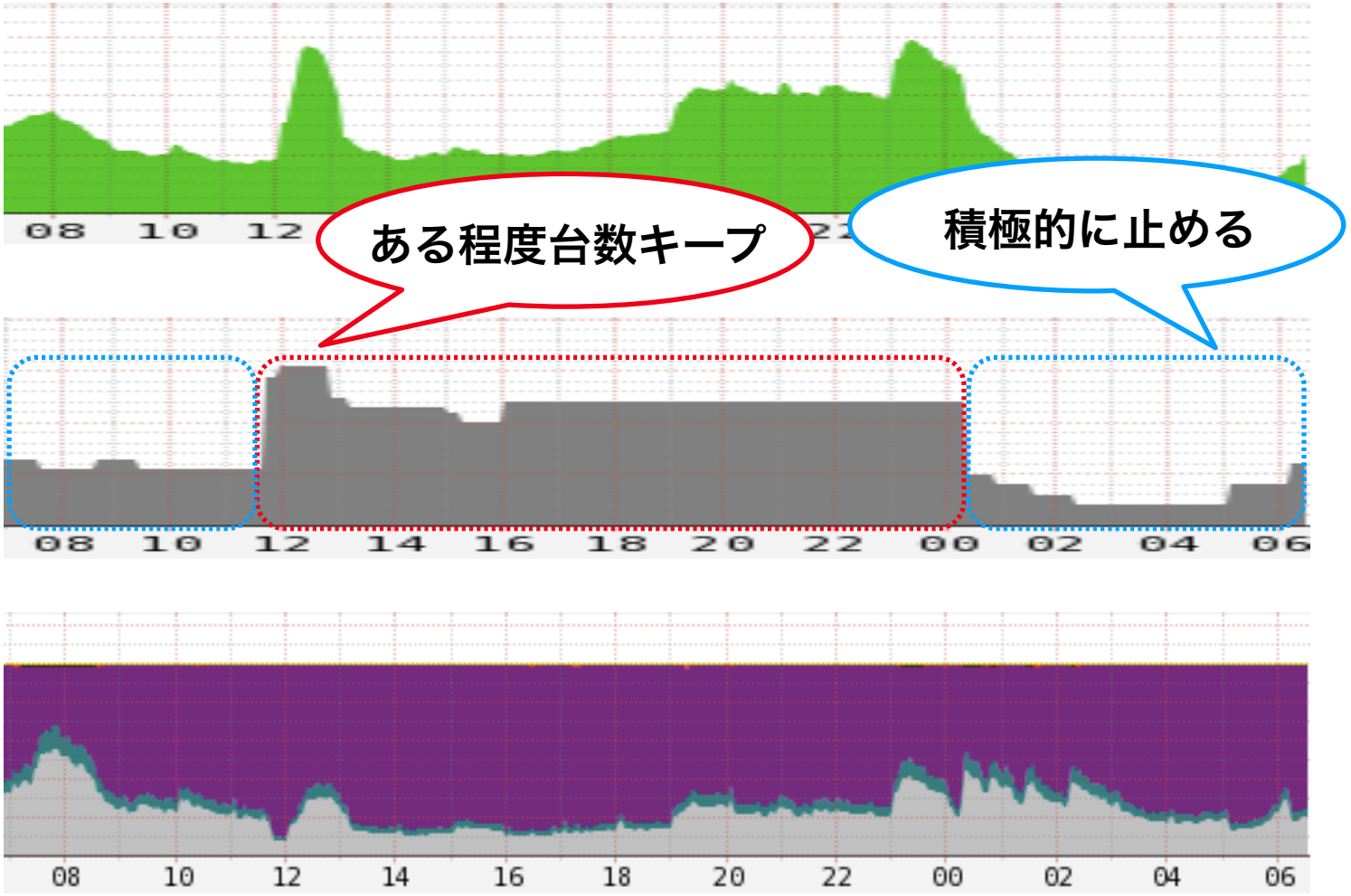


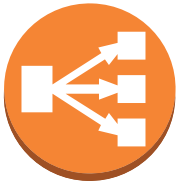
Auto-Scaling の例

リクエスト数

インスタンス数

CPU 使用率





ELB の状態監視

- ELB の IP アドレスの変化を IRC に通知
  - ELB はスケールアップしたりすると IP が変わる？
  - アラートの原因切り分けに利用している
  - https://github.com/hirose31/monitor-elb-address

```
22:44:39 (ikachan31g) IP Address of ELB has changed
22:44:40 (ikachan31g)  community-kr-806175539.ap-northeast-1.elb.amazonaws.com
22:44:42 (ikachan31g)    54.249.89.90
22:44:44 (ikachan31g) -  54.250.201.80
```

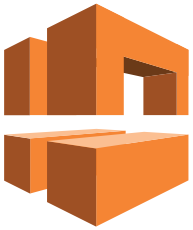
AWS で実現できたらいいこと

- 他の IaaS のいいとこどり
- リージョン間通信の向上
- ELB の見える化・無通信
- マネジメントコンソールのマルチアカウント
- サポートに関して

ほかの IaaS のいいとこどり

- たとえば Google Compute Engine から
  - 1 分単位での課金
  - 急激なアクセス増加に耐えうるロードバランサ
  - ライブマイグレーション
  - 複数 VM に接続可能な Persistent ディスク

※ AWSよりGoogle Compute Engineを選びたくなる10の理由



リージョン間通信の向上

- グローバルなサービスでリージョン間通信していきたい
  - 現状は JP/US で完全に独立した環境になっている
  - App はリージョン毎、DB は共通みたいにしたい
- IBM の softlayer がこのあたりは優位とのこと(※)
  - 各 DC と NW 拠点が 10Gbps の専用線
  - ネットワークの使用料は無料

※ AWSリージョン間通信向上と今後のエコシステム

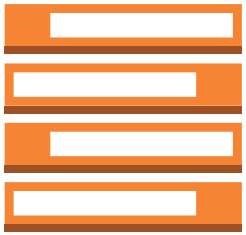
ユーザによる ELB の管理

- ソーシャルゲームは突発的なアクセス増が日常
  - 毎回プレウォーミングはちょっとツライ
- もうちょっとユーザが管理できるようになると嬉しい
  - キャパシティ確認、事前にキャパシティ追加

ELB の強制切断

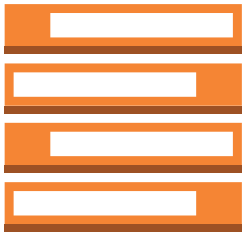
- HTTP のリアルタイム通信で不便
  - ゲームやチャットなどで利用するケースが増えてる
  - 無通信状態が続くと強制的に接続が切れてしまう
  - 最大で 17 分までは伸ばせるというウワサ(※)
  - オプションで設定できると嬉しい

※ https://forums.aws.amazon.com/thread.jspa?messageID=382182#



マネジメントコンソール

- マルチアカウント対応してほしい
  - サービスがたくさんあると行ったり来たりしがち
  - 王道なのは Chrome のマルチユーザ機能？
  - 別のブラウザを起動するという手も



AWS Support

- ちょうどいいサポートプランがない
  - 緊急な質問をまれに送る程度
  - ELB のプレウォーミングは使いたい時がある
  - 利用料金の数パーセントかかる
- サポートもオンデマンドで
  - 優先度付きのチケット買えるみたいな仕組みがよい？

まとめ

ぜんぶ AWS でやらないワケ

できないことは（たぶん）ない

ぜんぶ AWS でやるメリット

- システム全体を AWS 上での運用に最適化できる
  - 現状はオンプレのシステムを AWS で運用してる
- 運用面での工数が減る
  - サーバの故障 (インスタンス障害) が比較的少ない
  - キャパシティプランニングが楽に

ぜんぶ AWS でやるデメリット

- 自分たちだけでどうにもできない要素が増える
  - プラットフォーマーとしての説明責任が果たせない
- 大規模になってくるとコストメリットが見いだしにくい
  - 規模が大きいと大量にバルクで仕入れ可能
  - RI ありでもコスト優位にはならない

まとめ

- 考えてみると AWS 移行するのに機能的な壁は少ない
  - それだけ AWS のサービスは完成度が高い
- 一方で、金銭的な面で移行メリットがみえない状況
  - 規模が大きいと見積もりも難しい一面もあるが
  - 大きなメリットがないと開発側の工数確保も難しい

Thank you :D