Auto Scaling x Spot Instances によるスケーラビリティとコストカット（公開用）

2015-06-03

塚田朗弘@akitsukada（無所属）

About me

* 塚田 朗弘 @akitsukada
* 妻と娘の家族
* 好きなAWSサービス ⇨ Lambda
* #iOS #AWS #Rails
  #MySQL #Redis
* CAC→ドワンゴ(ニコ生開発)
  →スタディプラス(CTO) → **無職** (今月から)


無職楽しいです(^q^)

Purpose of today

* 今日お伝えしたいこと
  1. 具体的で比較可能で再現可能な
     モデルケースの提供
     * Auto Scalingについては Min-Max の
       振り幅感覚を、Spot&Reserved Instances に
       ついては使い所と倹約の規模感を
       * 既に導入している方は比較用に
       * 導入検討中の方は先行事例として
  2. 設計方針や注意点といった知見の共有

Purpose of today

* 今日お伝えできないこと
  1. そもそもAuto Scalingとは、Spot Instancesとはといった話
     * そういう話なら：[AWSマイスターシリーズ]
       リザーブドインスタンス&スポットインスタンス
       ⇨ http://www.slideshare.net/AmazonWebServicesJapan/20131023-aws-meisterregenerateriспotpublic
  2. 各機能の斬新な活用方法
     * 使い方自体は典型的と言われる部類の話です(たぶん)
  3. オンプレとの比較の話

Agenda

1. 想定サービス像/システム構成
2. 設計方針と(一部経験に基づく)注意点
3. 設定の具体例
4. 期待できる成果
5. その他のコストカットポイント
6. 今後の方向性、考えてること

会場アンケート

会場アンケート

* 既に日常業務で
  AutoScaling, Spot Instances,
  Reseved Instances を

1.使っていない

2.まぁまぁ使っている
  (が、適切な使い方ができているか分からない)

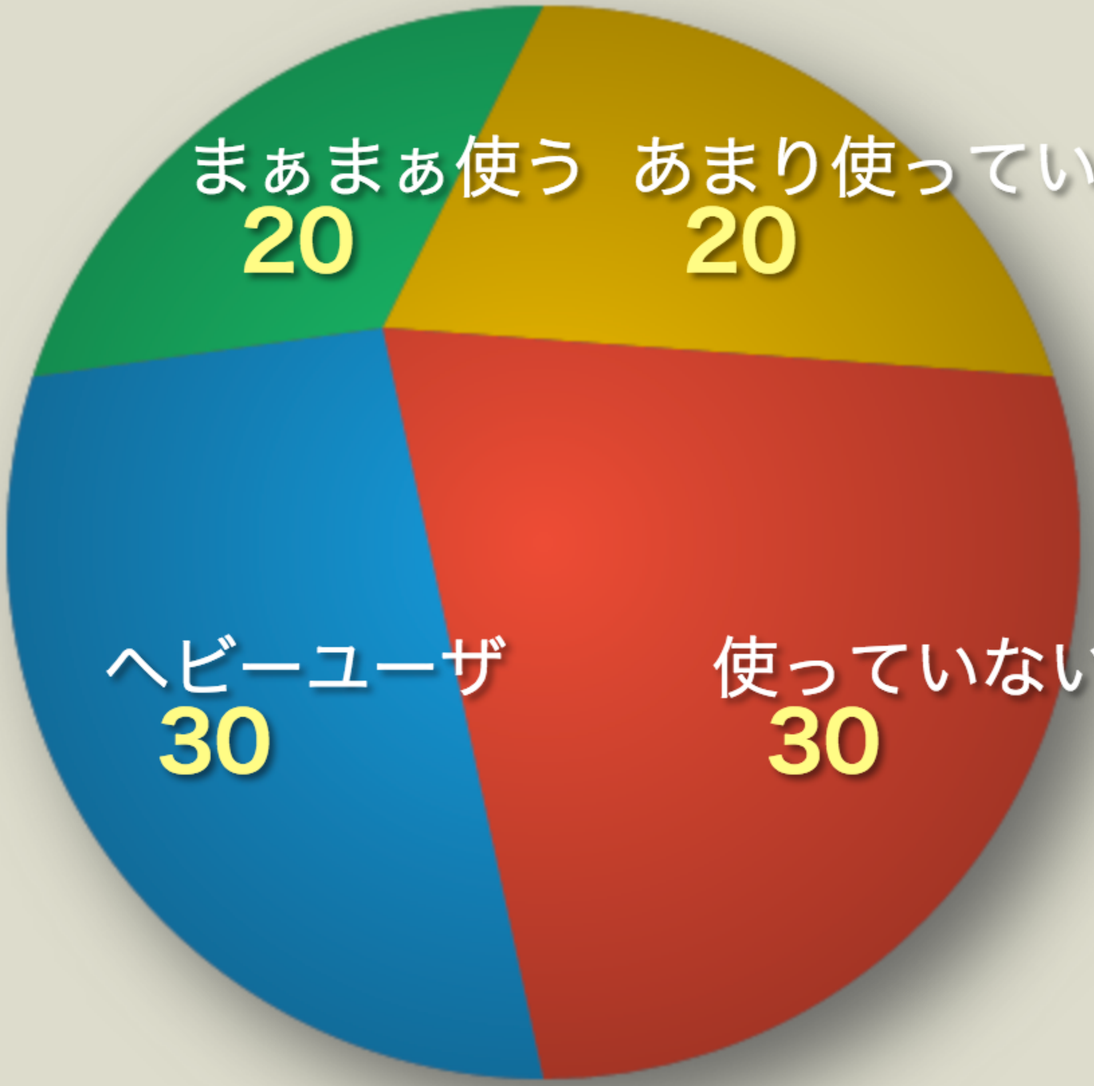
3.バリバリ活用している

会場アンケート結果

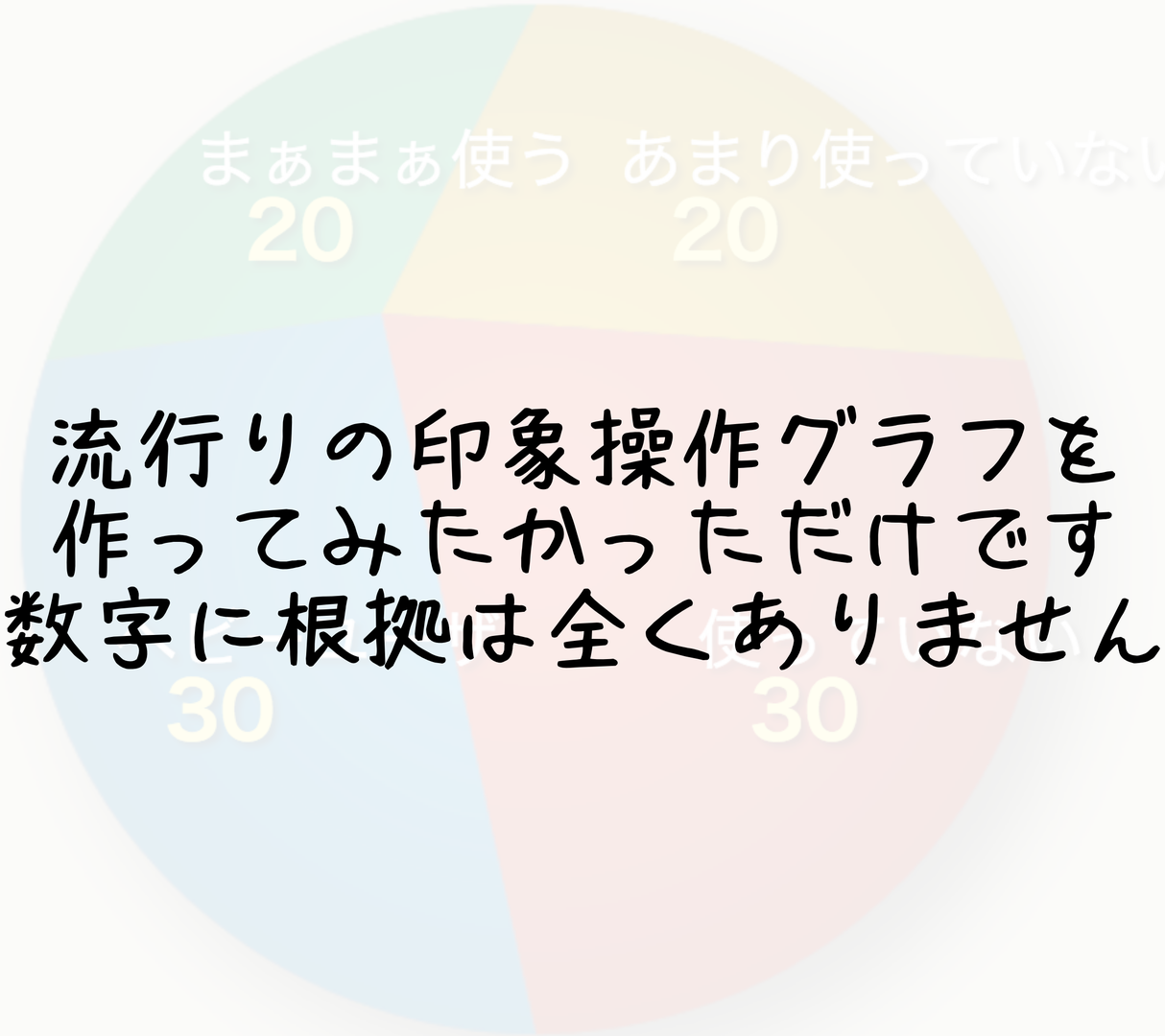
※公開用追記
会場で挙手をお願いした結果大体こんな感じでした

1. 使っていない…4.5割
2. まぁまぁ使っている…4.5割
3. バリバリ活用している…1割

講演者所感（身の回り感）



講演者所感（身の回り感）



流行りの印象操作グラフを
作ってみたかっただけです
（数字に根拠は全くありません）

想定サービス像
システム構成

想定サービス像

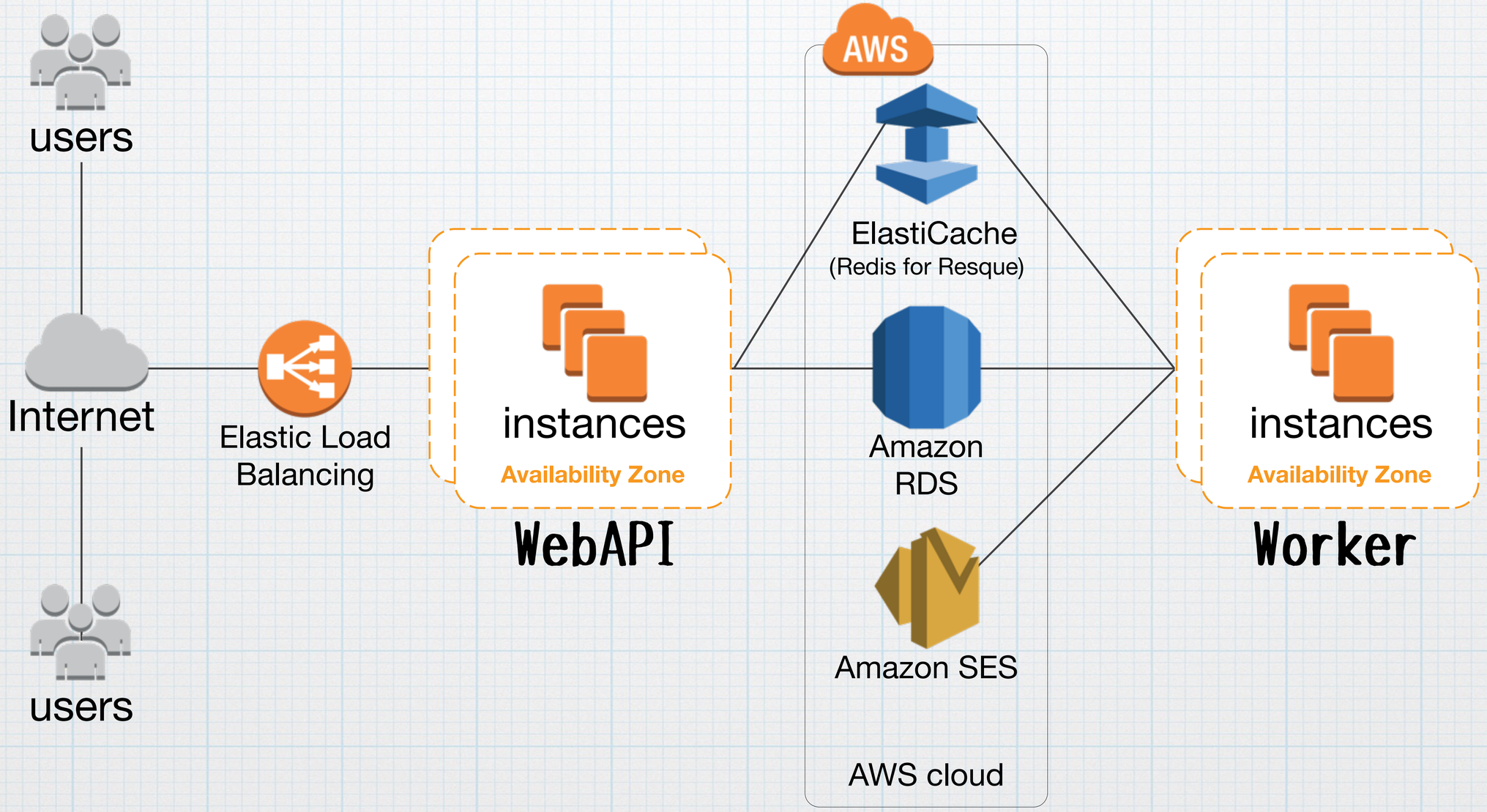
* Twitterライクな成長中の自社サービス(BtoC)
  * ユーザ間コミュニケーションが多い
  * プッシュ通知やメールの一斉配信を週に数回
  * サービス開始から２年経過
* ユーザ数=100万人〜
  * アクティブ率=30〜50%
  * 成長率=110%/月
    (一年後にはユーザ313万人になる計算)
* 悩み: サーバ代が高い

システム構成

- WebAPIサーバ on EC2
  - Ruby on Rails (Apache+Passenger)
    - エンドユーザからのHTTP(S)リクエストを処理
    - 時間がかかる処理はキューイングしてWorkerに任せる
  - m3.2xlarge 8台@1a,1c
- Worker on EC2*
  - Rails+Resque**,
    Rails+Resque+ResqueScheduler***
  - WebAPIやResqueSchedulerがキューイングしたJobを捌く
  - c4.2xlarge 10台@1a,1c

*Workerについて…2014/08, AWS Startup Tech Meetup http://j.mp/20140817_aws_tsukada
**Resque…GitHub製非同期処理ライブラリ Redisを使う
***ResqueScheduler…cronライクな記述でResqueのJobをKickしてくれるライブラリ

システム構成



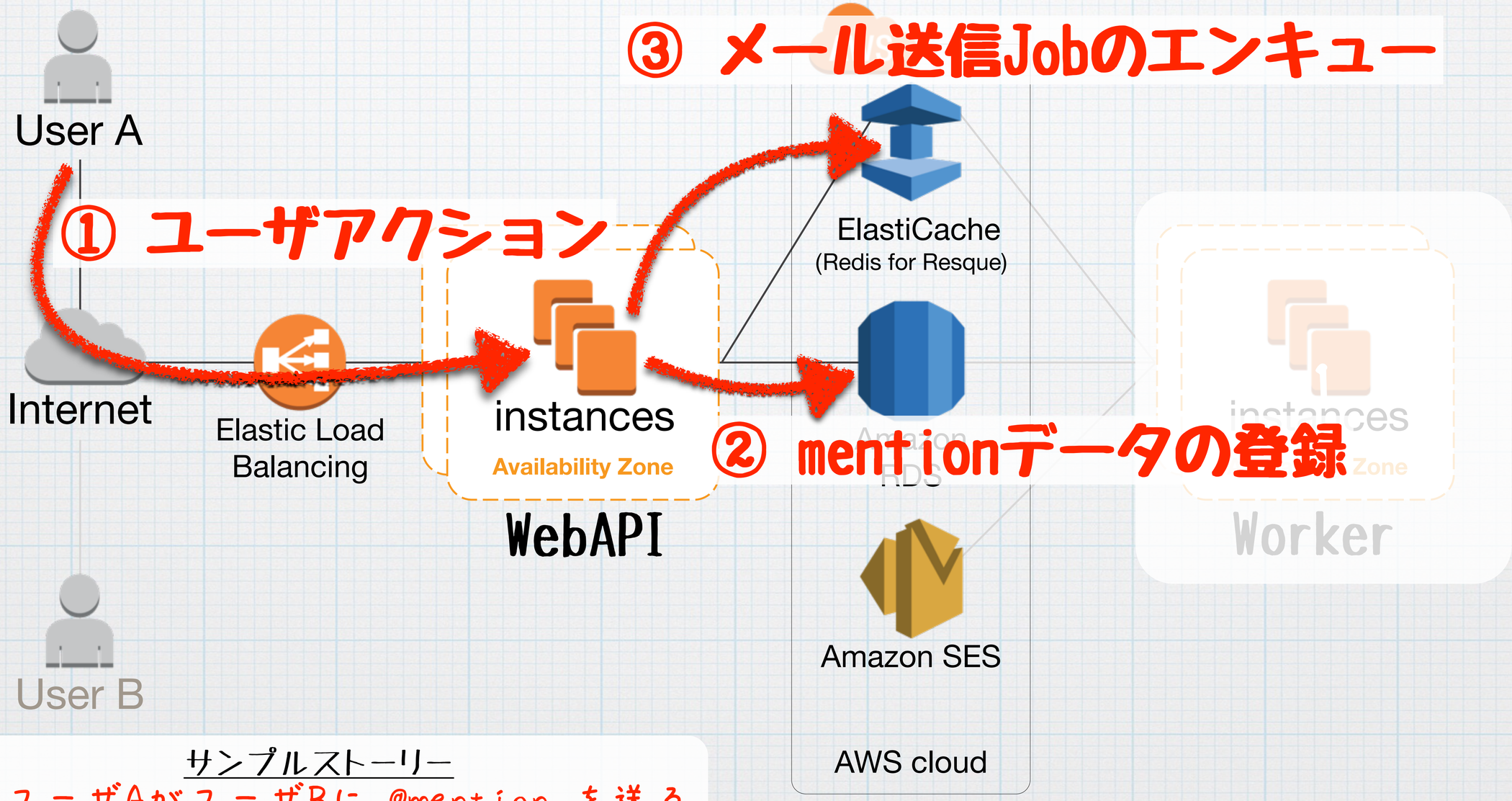
想定サービス像/システム構成

システム構成

サンプルストーリー

1. ユーザAがユーザBに @mention を送る
2. ユーザBは通知メールを受信する

システム構成

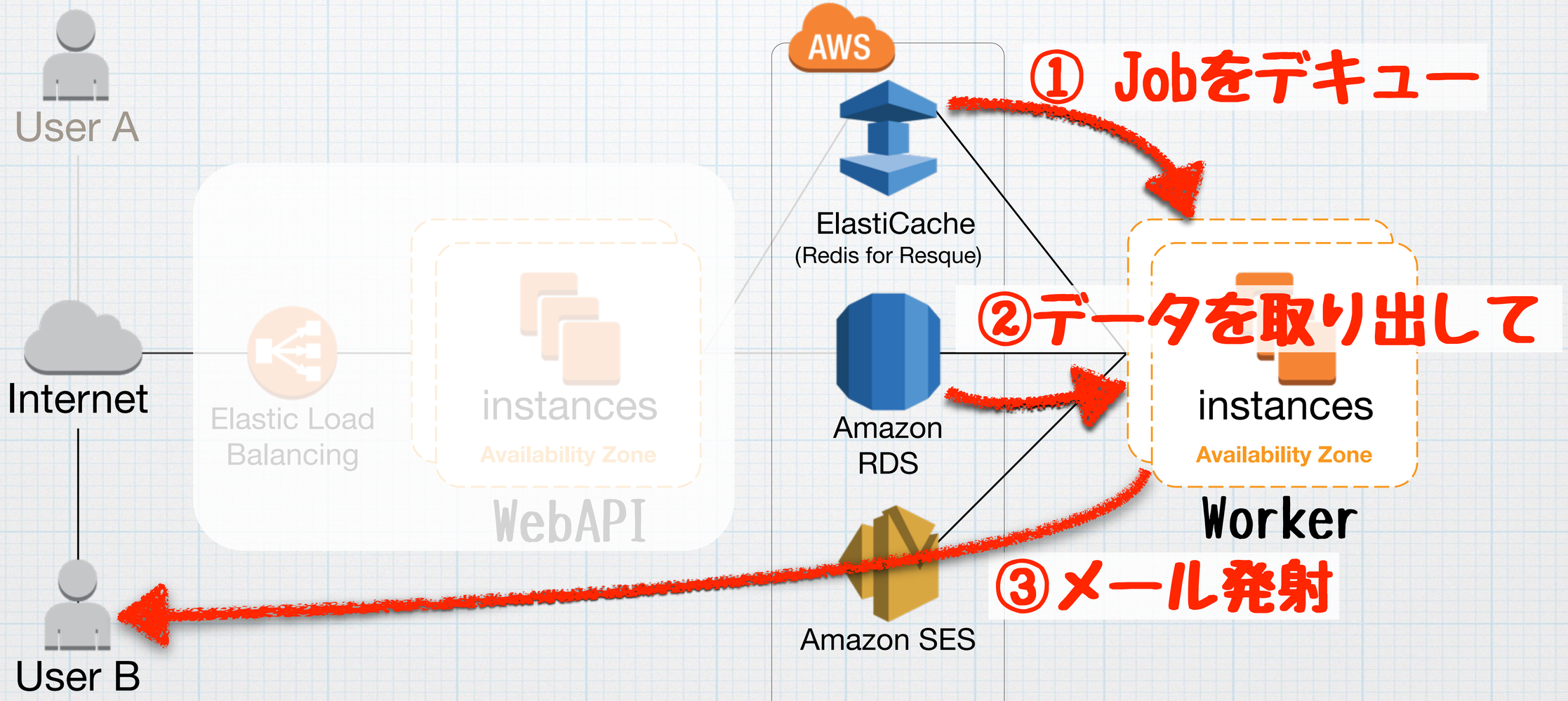


サンプルストーリー

1. ユーザAがユーザBに @mention を送る
2. ユーザBは通知メールを受信する

想定サービス像/システム構成

システム構成



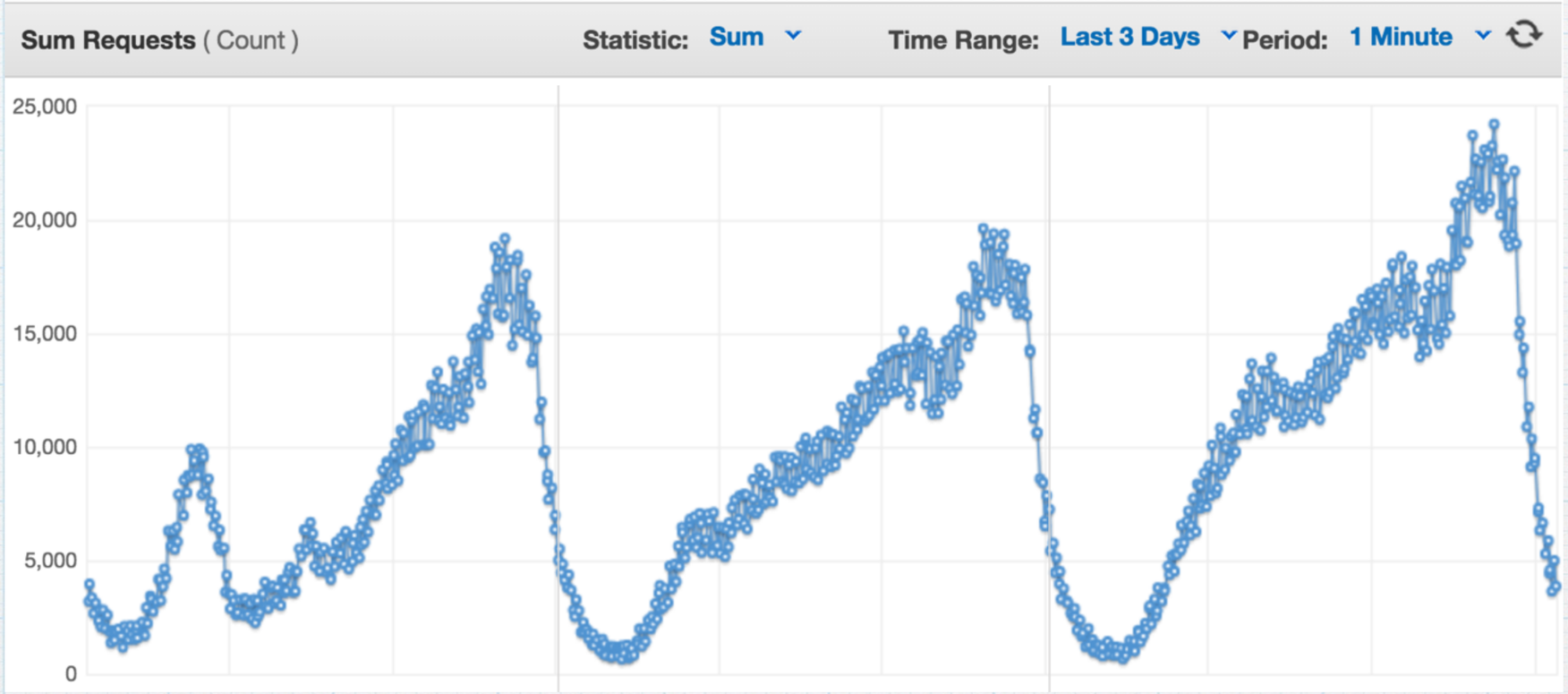
サンプルストーリー

1. ユーザAがユーザBに @mention を送る
2. ユーザBは通知メールを受信する

想定サービス像/システム構成

負荷推移傾向

WebAPI
Sum
Requests
(ELB)

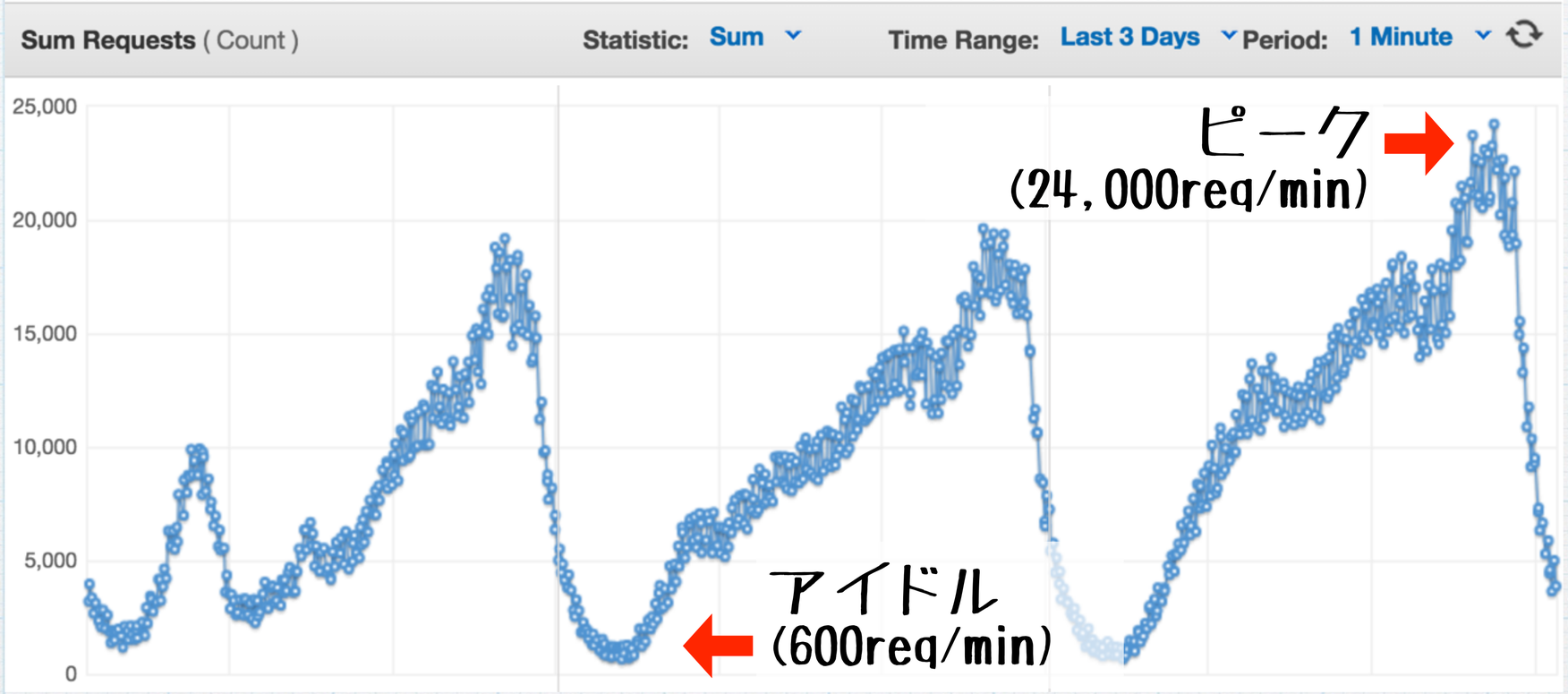


※このデータおよびグラフは今回の発表用に作成したものです。

想定サービス像/システム構成

負荷推移傾向

WebAPI
Sum
Requests
(ELB)



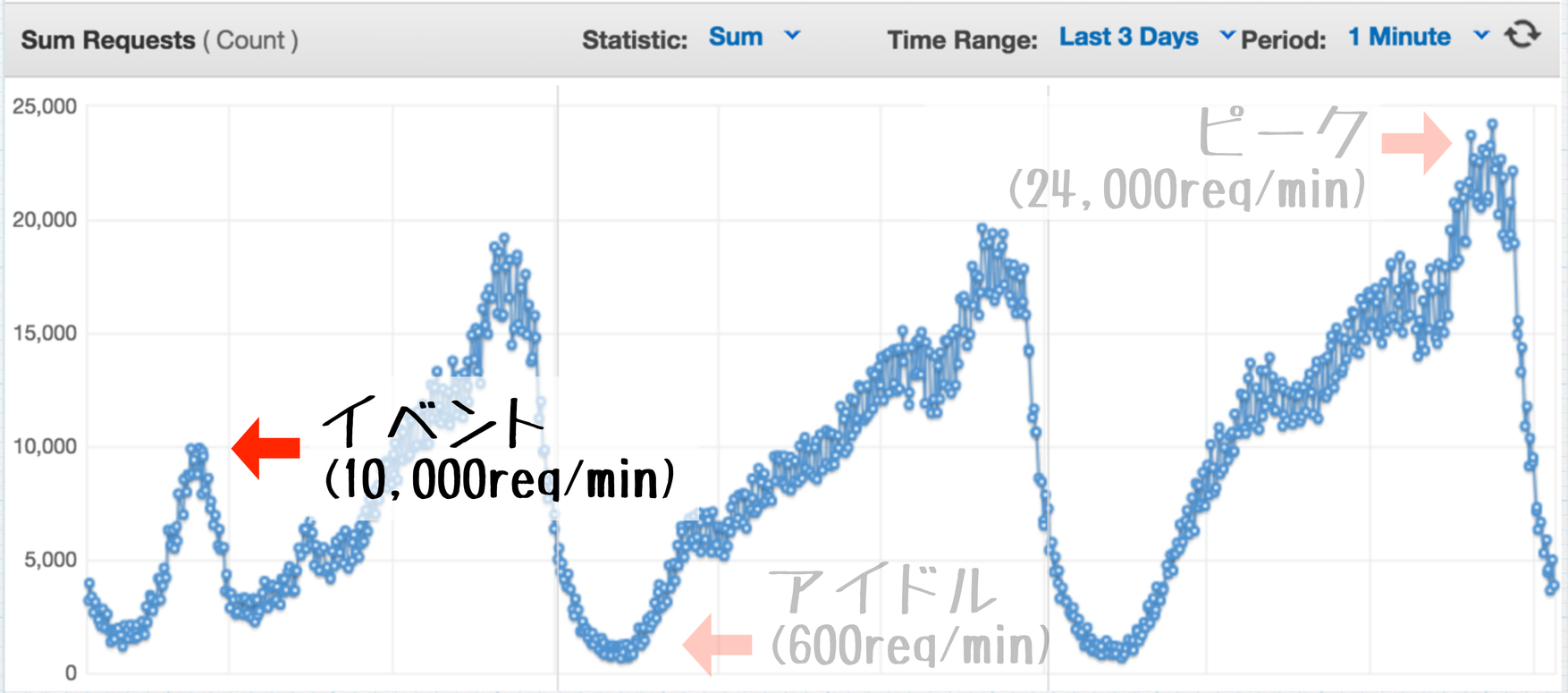
※このデータおよびグラフは今回の発表用に作成したものです。

* 毎日、割と規則正しく幅の大きいピーク/アイドルタイムが存在する(最大40倍程度)

想定サービス像/システム構成

負荷推移傾向

WebAPI
Sum
Requests
(ELB)



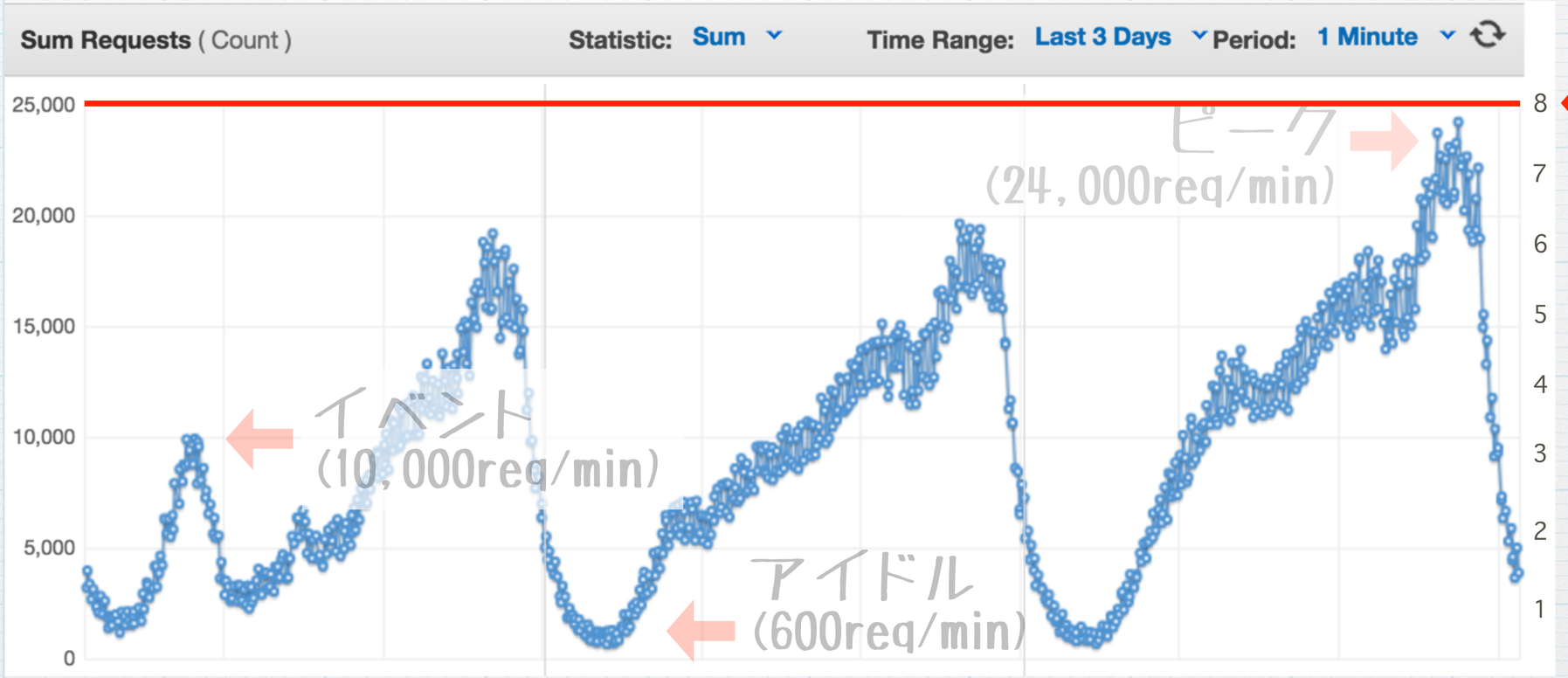
※このデータおよびグラフは今回の発表用に作成したものです。

* 毎日、割と規則正しく幅の大きいピーク/アイドルタイムが存在する(最大40倍程度)
* (予定された)イベント等によって一斉プッシュ通知等の配信があり、通常ピークタイム外に負荷がかかることがある

想定サービス像/システム構成

負荷推移傾向

WebAPI
Sum
Requests
(ELB)



インスタンスは常時8台！

※このデータおよびグラフは今回の発表用に作成したものです。

* 毎日、割と規則正しく幅の大きいピーク/アイドルタイムが存在する(最大40倍程度)
* (予定された)イベント等によって一斉プッシュ通知等の配信があり、通常ピークタイム外に負荷がかかることがある

想定サービス像/システム構成

課題

オフピーク時のリソースの無駄が多い

&

高い

仮に前述の構成を

・全てオンデマンド

・Auto Scaling なし

で実現するとEC2だけで

$9,195≒113万円/月

前後に＼(^o^)／

*2015年05月現在の為替、EC2料金での概算


振り幅が大きく
柔軟なスケーリング
&
圧倒的倹約
needed

設計方針と
(一部経験に基づく)注意点

設計方針

WebAPI

* リクエスト数に応じてAuto Scaling。
* Spot Instancesは基本的に使わない。
  * 突然死が許容できないので。
  * ピークタイムに少数台入れる、とかはしてもいいが少数ではあまり旨味がない。

Worker

* Job数に応じてAuto Scaling。
* Spot Instancesを(超)優先的に使う。
  * Spotは別Auto Scaling Group。
  * Spot以外は基本Reserved1台だけ。
  * 突然死後のリカバリを考慮しておく。

共通

* Spot Instancesを使わない場合も極力オンデマンドでなく Reserved Instancesを適用。購入数はAuto Scaling設定後の各統計から算出。
* 小さめのインスタンスタイプを使って細かく増減させて節約。
  * m3.2xlarge (8vCPU/26ECU/30GiB) ⇨ m3.large x 4
  * c4.2xlarge (8vCPU/31ECU/15GiB) ⇨ c4.large x 4
* 予定されたアクセス増(Push配信後とか)にはScheduled Actionで対応。
* 予測不可能なスパイクアクセスの対応は今日は考えない。(これはこれで色々あるのでまた今度)

設計方針と注意点

設計方針

WebAPI

* リクエスト数に応じてAutoScaling。
* Spot Instancesは基本的に使わない。
  * 突然死が許容できないので。
  * ピークタイムに少数台入れる、とかはしてもいいが少数ではあまり旨味がない

Worker

* Job数に応じてAutoScaling。
* Spot Instancesを(超)優先的に使う。
  * Spotは別Auto Scaling Group。
  * Spot以外は基本Reserved1台だけ。
  * 突然死後のリカバリを考慮しておく。

共通

* Spot Instancesを使わない場合も極力オンデマンドでなく Reserved Instancesを適用。購入数はAutoScaling設定後の稼働時間から算出。
* 小さめのインスタンスタイプを使って細かく増減。
  * m3.2xlarge (8vCPU/26ECU/30GiB) ⇒ m3.large x 4
  * c4.2xlarge (8vCPU/31ECU/15GiB) ⇒ c4.large x 4
* 予定されたアクセス増(Push配信後とか)にはScheduled Actionで対応。
* 予測不可能なスパイクアクセスの対応は今日は考えない。(これはこれで色々あるのでまた今度)

設計方針と注意点

“Spotを(超)優先する” とは

* オンデマンド/Reserved より Spot Instance を
  * 少し早くスケールアウトさせ
  * より遅くスケールインさせる
    * これにより
      「Spotが買える限りSpotのみが増減する」状況を作る
    * Auto Scaling Groupを分けることでこれができる
* 価格高騰してSpot Instanceが全部落ちたらどうするの？
  * 最低1台は立っている Reserved が、問題ない規模まで
    スケールアウトしてくれる(その所要時間、Max20分程度は許容)
  * 処理中にSpotが落ちてStuckしたJobを定期的に回収、
    リトライするJobを書いておけば無問題 …と開き直りました

“Spotを(超)優先する” とは

176 of 245 Workers Working

The list below contains all workers which are currently running a job.

	Where	Queue	Processing
	[redacted]	[redacted]	Waiting for a job...
	[redacted]	[redacted]	[redacted] about 2 hours ago
	[redacted]	[redacted]	[redacted] about 2 hours ago

* 処理中にSpotが落ちてStuckしたJobを定期的に回収、リトライするJobを書いておけば無問題 ...と開き直りました

設計方針と注意点

(一部経験に基づく)注意点

WebAPI

* スティッキーにしない。
  ELBもアプリもステートレスに。
* 急なアクセス増の準備時にはインスタンス数だけでなくELBのスケール*にも注意。

Worker

* 仮にSpotが全く買えなくても
  オンデマンド/Reservedで捌けるようアプリを設計しておく。
* ELBのHealth Checkに頼れない
  ので自前監視＆自殺機構必須。

*http://bit.ly/1KxDZBx - ELBでスパイクアクセスと戦う[Qiita]

共通

* スケールアウトのトリガーからサービスインまでを極力短く(2〜5分)する。
  * 時間が掛かる初期化処理はAMIに入れておく。
    * 全部をchefで頑張ったり**しない。
* 適切な threshold はサービスの仕様、規模、ユーザ層の変化等によって少しずつ変わっていく。定期的に設定を見直し、「あれ、ピークなのに増えてくれない」を避ける。
* インスタンス数、Spot Request数等の上限***に注意。
  * 「なんで起動しないんや…」ってなります。

**http://amzn.to/1EniqIm - AutoScale×ゲーム ～運用効率化への取り組み～
***http://aws.amazon.com/jp/ec2/faqs/#How_many_instances_can_I_run_in_Amazon_EC2

具体例

*Auto Scaling設定例

*パラメータチューニング要領

*運用方法

*Reserved Instancesの試算要領

Auto Scaling設定例/WebAPI

具体例

WebAPI

Auto Scaling Group

* Name: “WEB-API-ASG”
* ELB: “web-api-balancer”
* Min: 4
* Max: 36
* Health Check Grace Period: 300
* Availability Zones: “ap-northeast-1a”, “ap-northeast-1c”

Launch Configuration

* Name: “WEB-API-LC-v1”
* AMI: “ami-XXXXXXX”
* Instance Type: “m3.large”

Scaling Policy

* Adjustment Type: “ChangeInCapacity”
* Scaling Adjustment: 2
* Cooldown: 300

Cloud Watch Alarm

* Scale In Alarm
  * Comparison Operator: “LessThanThreshold”
  * Evaluation Periods: 3
  * Period: 300
  * Statistic: “Maximum”
  * Metric Name: “CPUUtilization”
  * Threshold: 30
  * Unit: “Percent”
* Scale Out Alarm
  * Comparison Operator: “GreaterThanThreshold”
  * Evaluation Periods: 1
  * Period: 300
  * Statistic: “Maximum”
  * Metric Name: “CPUUtilization”
  * Threshold: 65
  * Unit: “Percent”

具体例/Auto Scaling設定例/WebAPI

WebAPI

Scheduled Actions

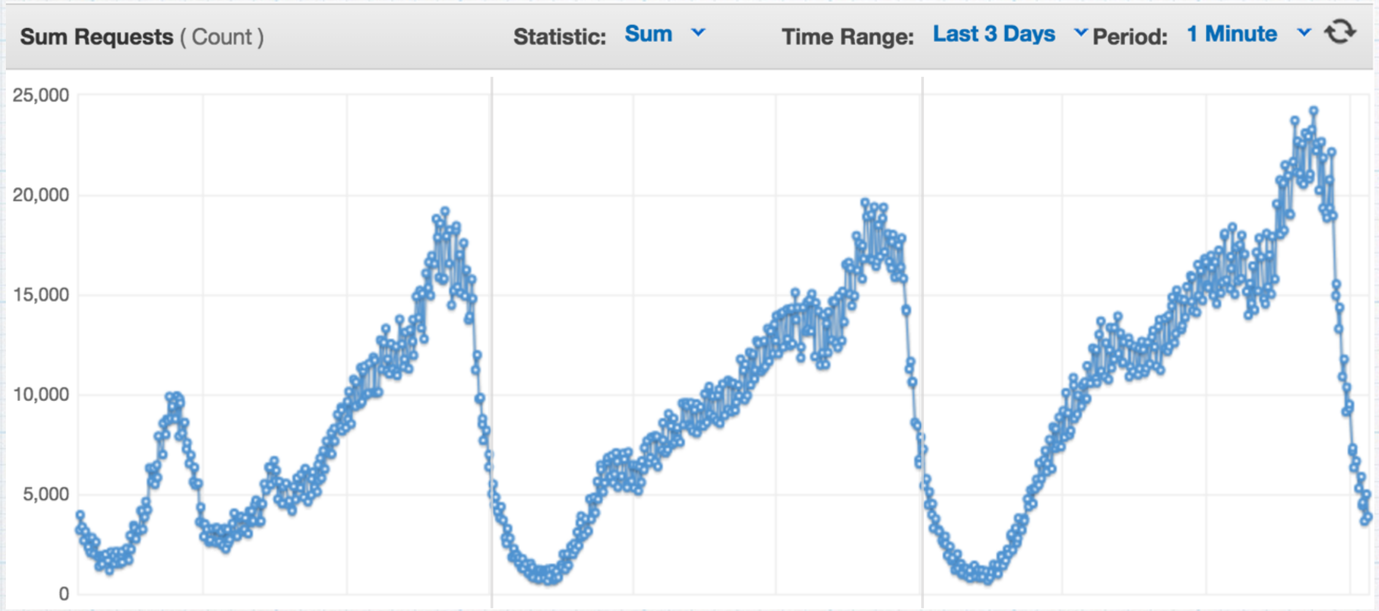
```
*Name: “event-push-start”
 *Min: 22
 *Max: 36
 *Reccurence: “45 1 * * 1” # 月曜日10:45(JST)に発動
*Name: “event-push-stop”
 *Min: 4                   # 元のMin/Maxに戻す(実際はyamlの参照を使ってDRYに書く)
 *Max: 36
 *Reccurence: “30 2 * * 1” # 11:30(JST)に発動
```

* これらの config をyaml に書いておき、
  自前のツールからAPI を叩いて
  設定更新するという運用スタイル
* この config を適用すると…

具体例/Auto Scaling設定例/WebAPI

Auto Scalingイメージ

WebAPI
Sum
Requests
(ELB)



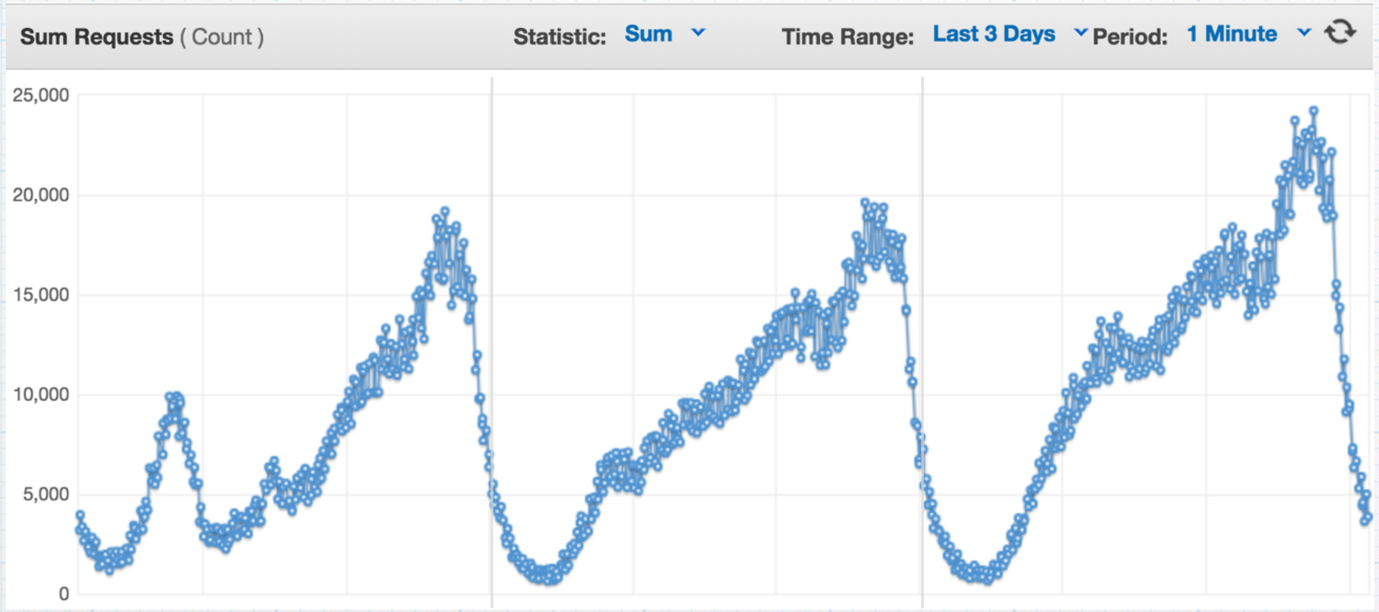
このReq数に対して

※このデータおよびグラフは今回の発表用に作成したものです。

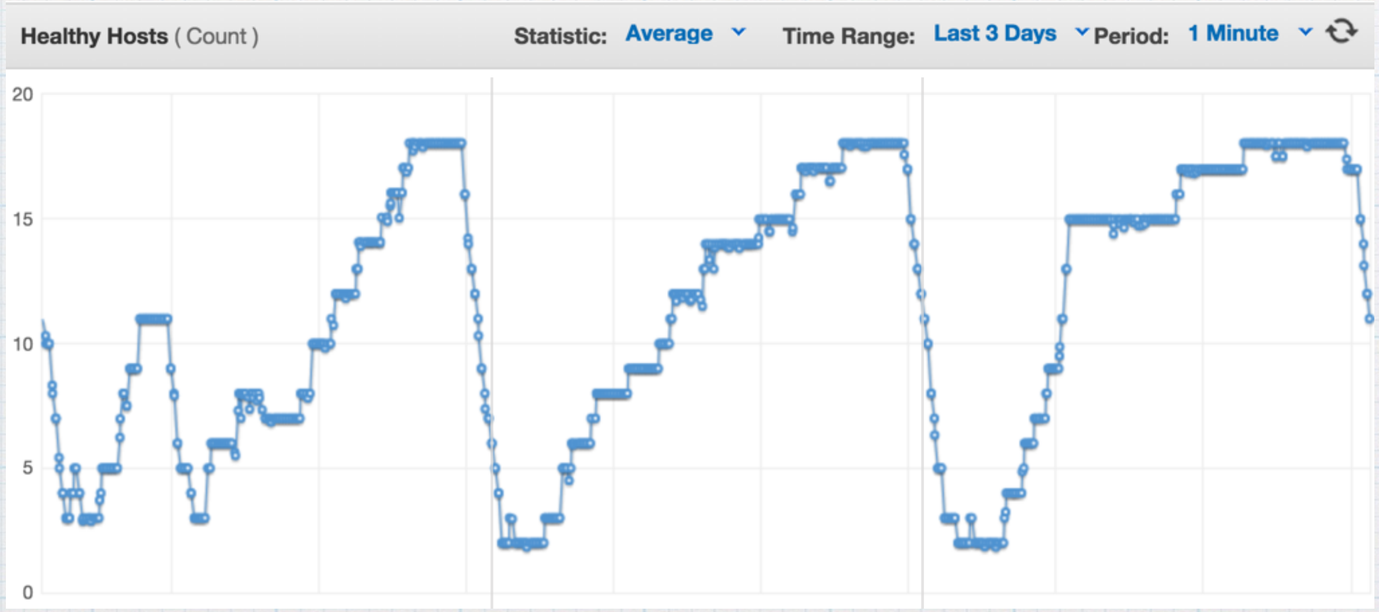
具体例/Auto Scaling設定例/WebAPI

Auto Scalingイメージ

WebAPI
Sum
Requests
(ELB)



WebAPI
Healthy
Hosts
(ELB)

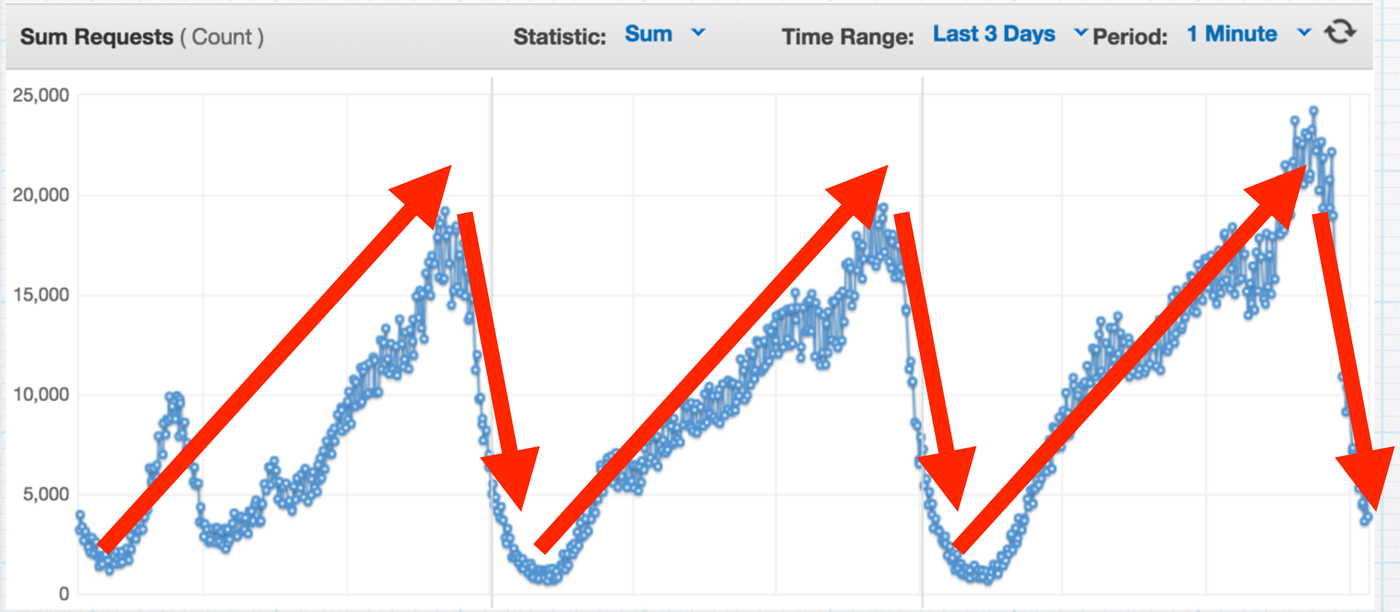


※このデータおよびグラフは今回の発表用に作成したものです。

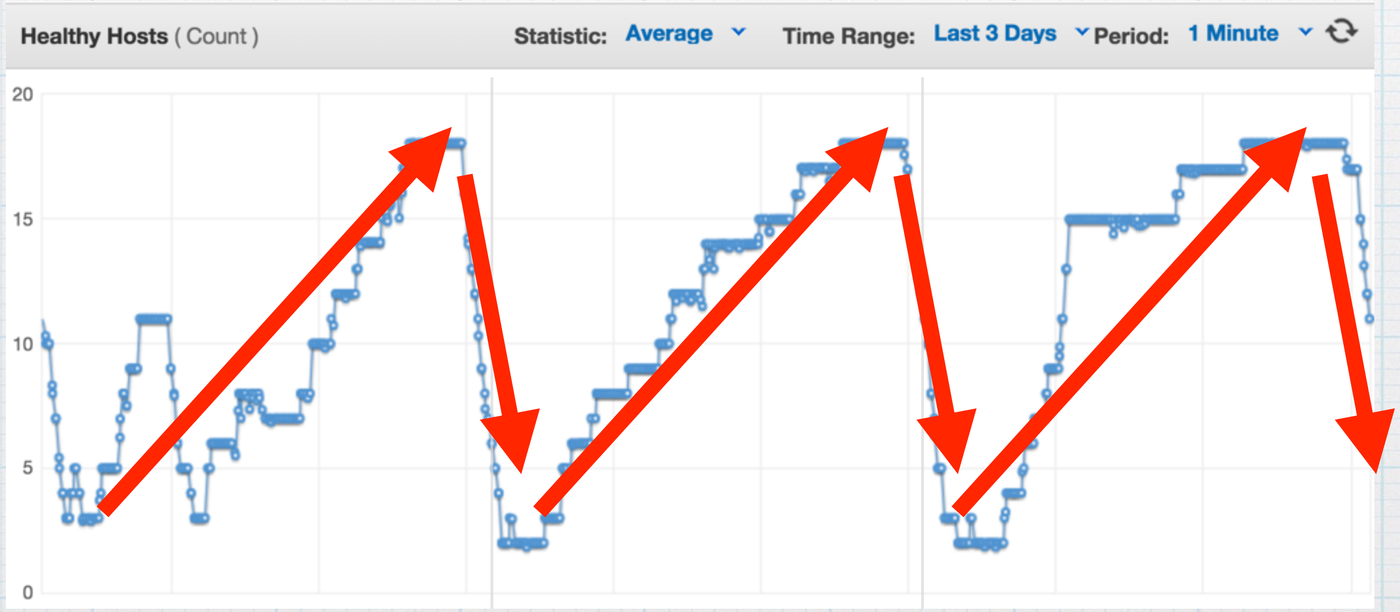
具体例/Auto Scaling設定例/WebAPI

Auto Scalingイメージ

WebAPI
Sum
Requests
(ELB)



WebAPI
Healthy
Hosts
(ELB)



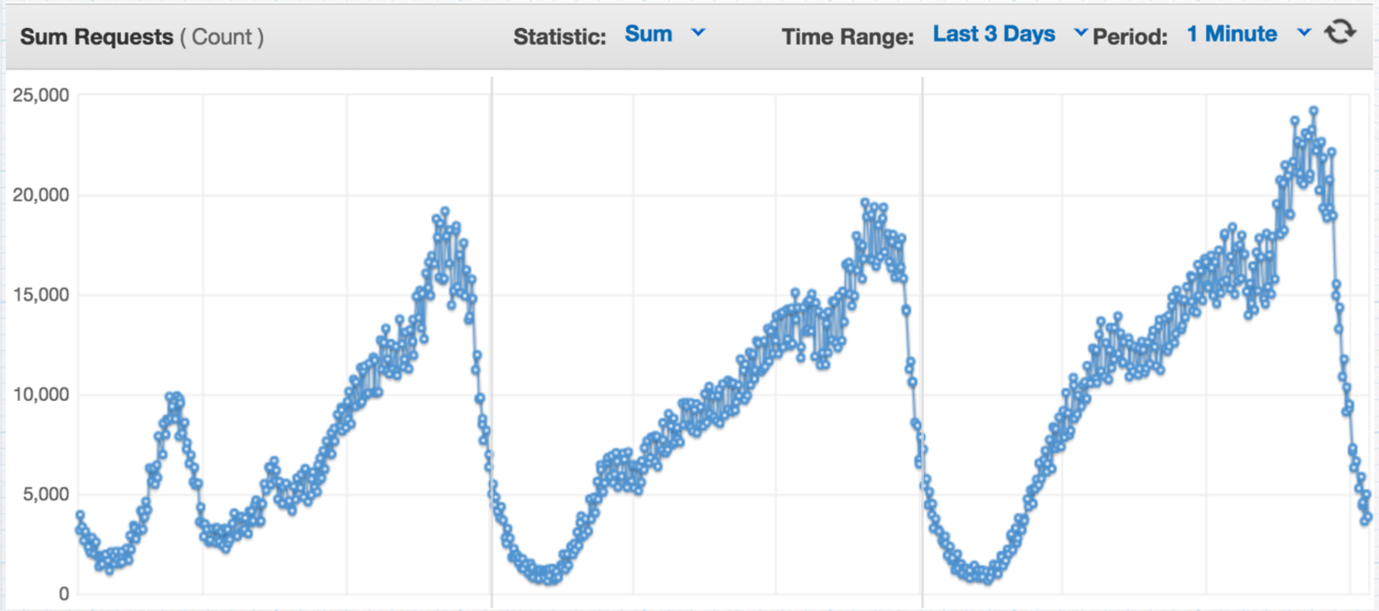
* ほぼReq数と同型のインスタンス増減

※このデータおよびグラフは今回の発表用に作成したものです。

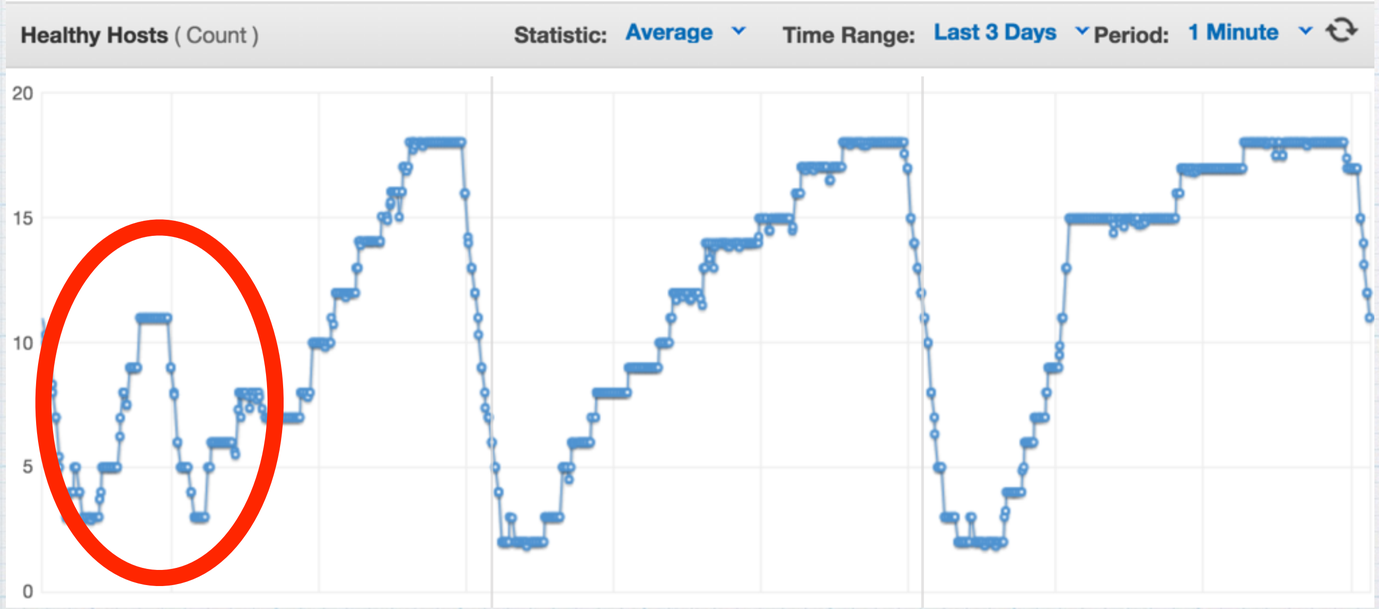
具体例/Auto Scaling設定例/WebAPI

Auto Scalingイメージ

WebAPI Sum Requests (ELB)



WebAPI Healthy Hosts (ELB)



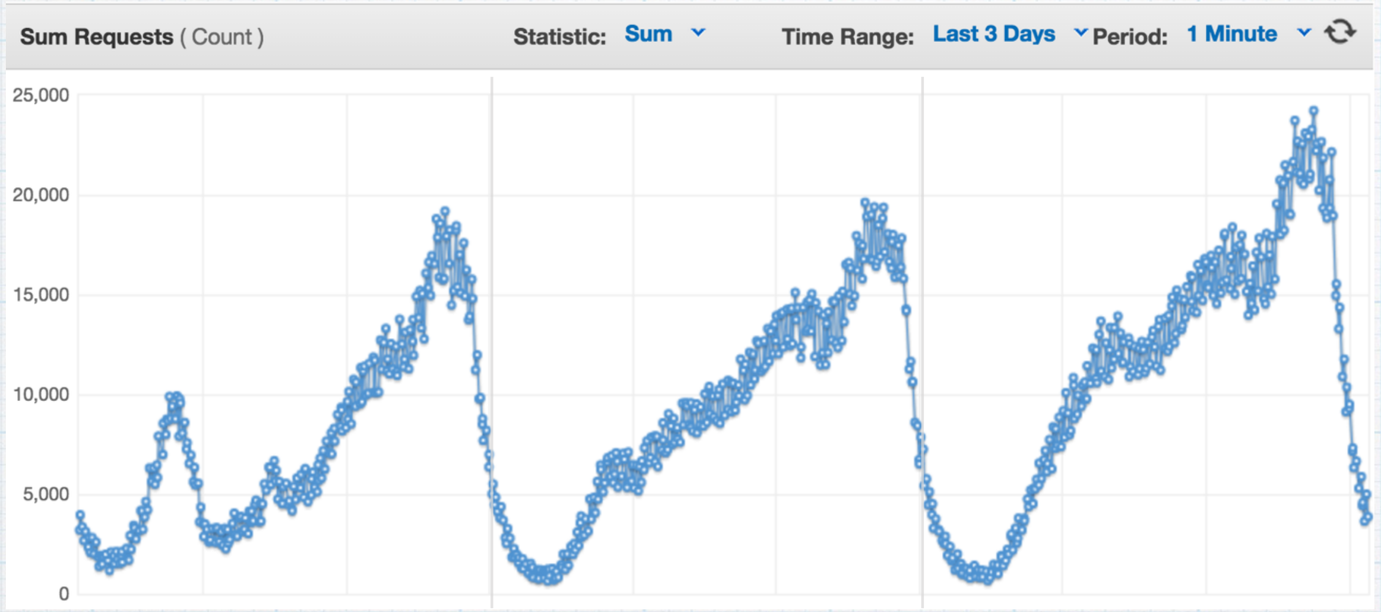
※このデータおよびグラフは今回の発表用に作成したものです。

* ほぼReq数と同型のインスタンス増減
* Scheduled-Action も効いてる

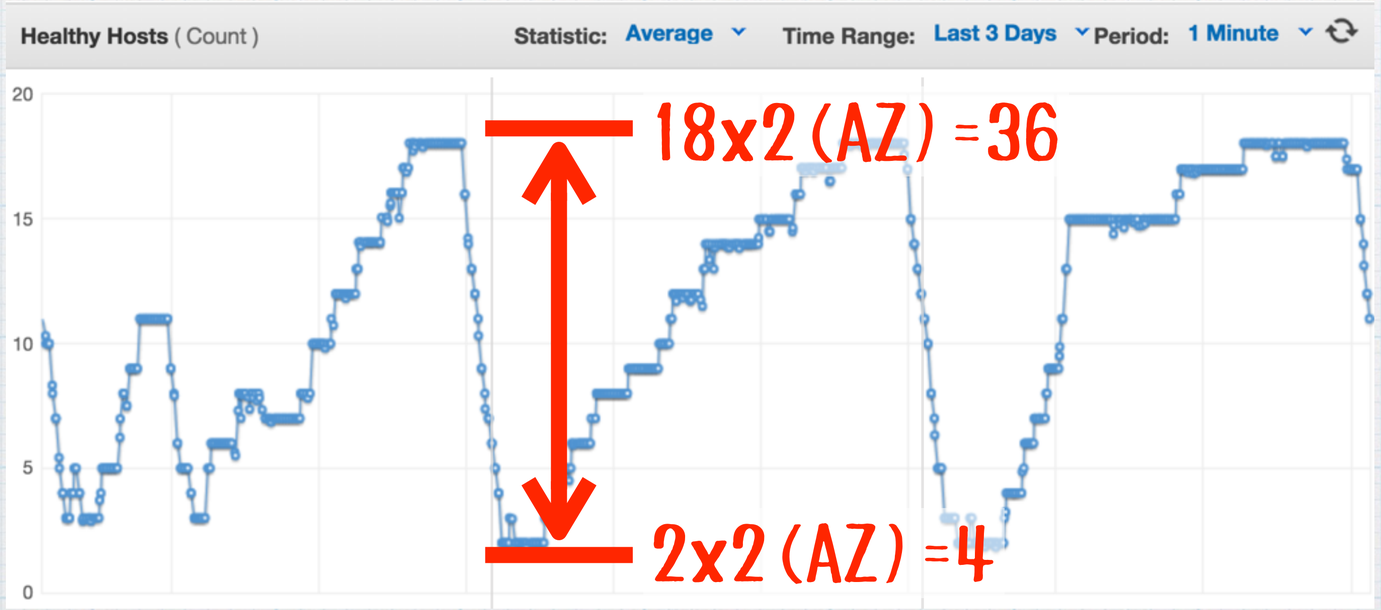
具体例/Auto Scaling設定例/WebAPI

Auto Scalingイメージ

WebAPI Sum Requests (ELB)



WebAPI Healthy Hosts (ELB)



※このデータおよびグラフは今回の発表用に作成したものです。

* ほぼReq数と同型のインスタンス増減
* Scheduled-Action も効いてる
* Hostsは全AZのAverageで出ており、今回は1a&1cを指定しているので、合計台数は4～36

具体例/Auto Scaling設定例/WebAPI

Auto Scaling設定例/ Worker&Worker-Spot

具体例

Worker

Auto Scaling Group

- Name: "WORKER-ASG"
- ELB: -
- Min: 1
- Max: 26
- Health Check Grace Period: 300
- Availability Zones
  - "ap-northeast-1a" , "ap-northeast-1c"

Launch Configuration

- Name: "WORKER-LC-v1"
- AMI: "ami-XXXXXXX"
- Instance Type: "c4.large"
- Spot Price: -

Scaling Policy

- Adjustment Type: "ChangeInCapacity"
- Scaling Adjustment: 2
- Cooldown: 300

Cloud Watch Alarm

- Scale In Alarm
  - Comparison Operator: "LessThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Average"
  - Metric Name: "CPUUtilization"
  - Threshold: 35
  - Unit: "Percent"
- Scale Out Alarm
  - Comparison Operator: "GreaterThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Average"
  - Metric Name: "CPUUtilization"
  - Threshold: 70
  - Unit: "Percent"

Worker-Spot

Auto Scaling Group

- Name: "WORKER-ASG-SPOT"
- ELB: -
- Min: 1
- Max: 26
- Health Check Grace Period: 300
- Availability Zones
  - "ap-northeast-1a", "ap-northeast-1c"

Launch Configuration

- Name: "WORKER-SPOT-LC-v1"
- AMI: "ami-XXXXXXX"
- Instance Type: "c4.large"
- Spot Price: "0.147"

Scaling Policy

- Adjustment Type: "ChangeInCapacity"
- Scaling Adjustment: 2
- Cooldown: 300

Cloud Watch Alarm

- Scale In Alarm
  - Comparison Operator: "LessThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Maximum"
  - Metric Name: "CPUUtilization"
  - Threshold: 25
  - Unit: "Percent"
- Scale Out Alarm
  - Comparison Operator: "GreaterThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Maximum"
  - Metric Name: "CPUUtilization"
  - Threshold: 60
  - Unit: "Percent"

Worker-Spot

Auto Scaling Group

- Name: "WORKER-ASG-SPOT"
- ELB: -
- Min: 1
- Max: 26
- Health Check Grace Period: 300
- Availability Zones
  - "ap-northeast-1a", "ap-northeast-1c"

Launch Configuration

- Instance Type: c4.large
- Spot Price: "0.147"

入札戦略：オンデマンドより高くなったら買わない
http://www.slideshare.net/AmazonWebServicesJapan/20131023-aws-meisterregenerateriспotpublic/72

Scaling Policy

- Scaling Adjustment: 2
- Cooldown: 300

Cloud Watch Alarm

- Scale In Alarm
  - Comparison Operator: "LessThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Maximum"
  - Metric Name: "CPUUtilization"
  - Threshold: 25
  - Unit: "Percent"
- Scale Out Alarm
  - Comparison Operator: "GreaterThanThreshold"
  - Evaluation Periods: 1
  - Period: 300
  - Statistic: "Maximum"
  - Metric Name: "CPUUtilization"
  - Threshold: 60
  - Unit: "Percent"

Worker-Spot

Auto Scaling Group

- Name: "WORKER-ASG-SPOT"
- ELB: -
- Min: 1
- Max: 26
- Health Check Grace Period: 300
- Availability Zones
  - "ap-northeast-1a", "ap-northeast-1c"

Launch Configuration

- Instance Type: c4.large
- Spot Price: "0.147"

入札戦略：オンデマンドより高くなったら買わない
http://www.slideshare.net/AmazonWebServicesJapan/20131023-aws-meisterregeneraterispotpublic/72

Scaling Policy

- Scaling Adjustment: 2
- Cooldown: 300

Cloud Watch Alarm

常にWorkerよりWorker-Spotを優先させるための調整

Scale In Alarm

- Evaluation Periods: 1
- Period: 300
- Statistic: "Maximum"
- Metric Name: "CPUUtilization"
- Threshold: 25
- Unit: "Percent"

Scale Out Alarm

- Evaluation Periods: 1
- Period: 300
- Statistic: "Maximum"
- Metric Name: "CPUUtilization"
- Threshold: 60
- Unit: "Percent"

パラメータチューニング要領*

* 以下の考え方を元にやってみて細かく調整
  1. Max=ピーク時の必要インスタンス数
  2. Min=Max*(アイドル時Req数/ピーク時Req数)+α
  3. Threshold(CPU Utilization)
     * Top=ピーク時の平均CPU使用率
     * Bottom=ピーク1時間前後にMax台が稼働している場合の平均CPU使用率
  4. Period, Evaluation Periods, Cooldown
     * アプリやサーバの性質によるため一般化が難しい

*これらは講演者の経験則であり、絶対的な解ではありません。

具体例/Auto Scaling設定例/パラメータチューニング要領

Reserved Instancesの試算

具体例/Reserved Instancesの試算

Reserved Instancesの試算

激しいAutoScalingの下では

一気に計算が面倒になる…が、

インスタンス数のグラフ、

Billing Infomation、

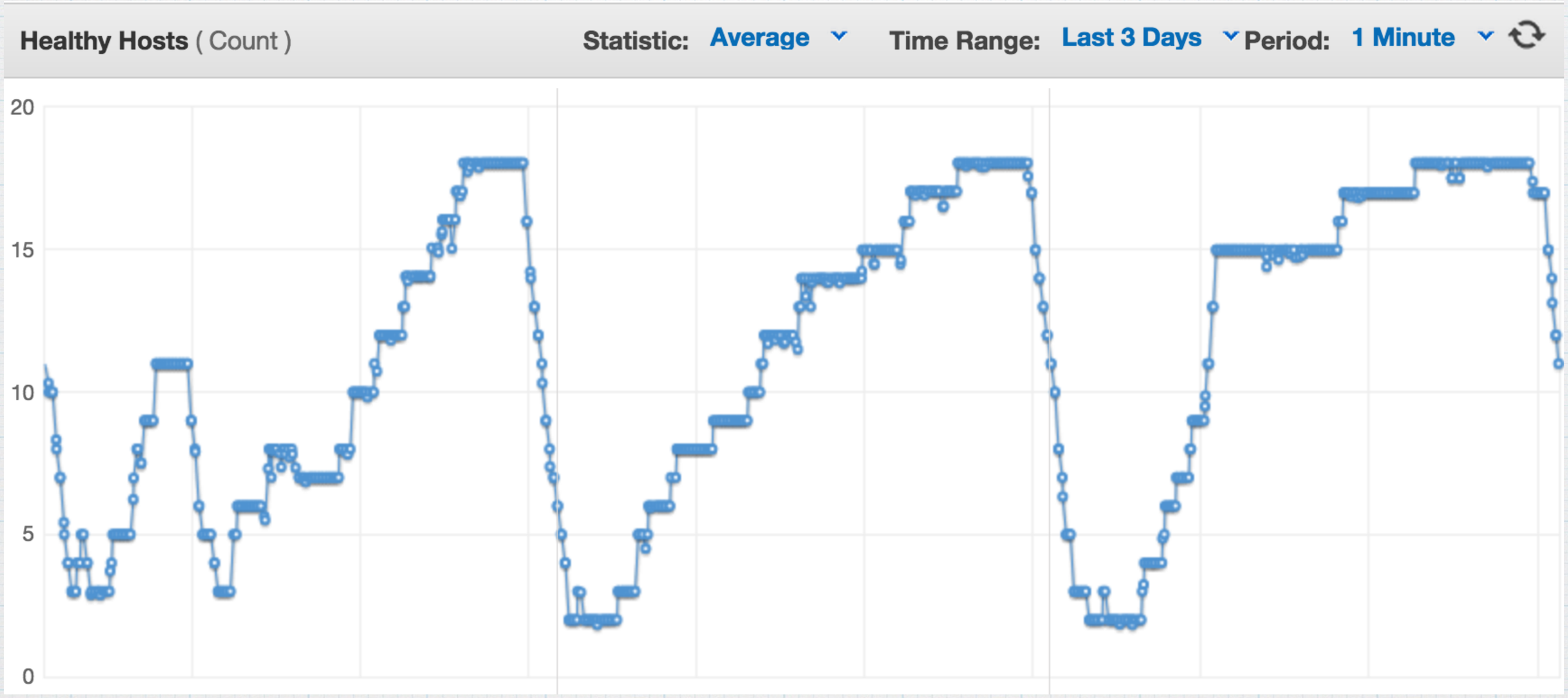
見積もりツール*と

にらめっこしながら頑張る。

*見積もりツール http://calculator.s3.amazonaws.com/index.html

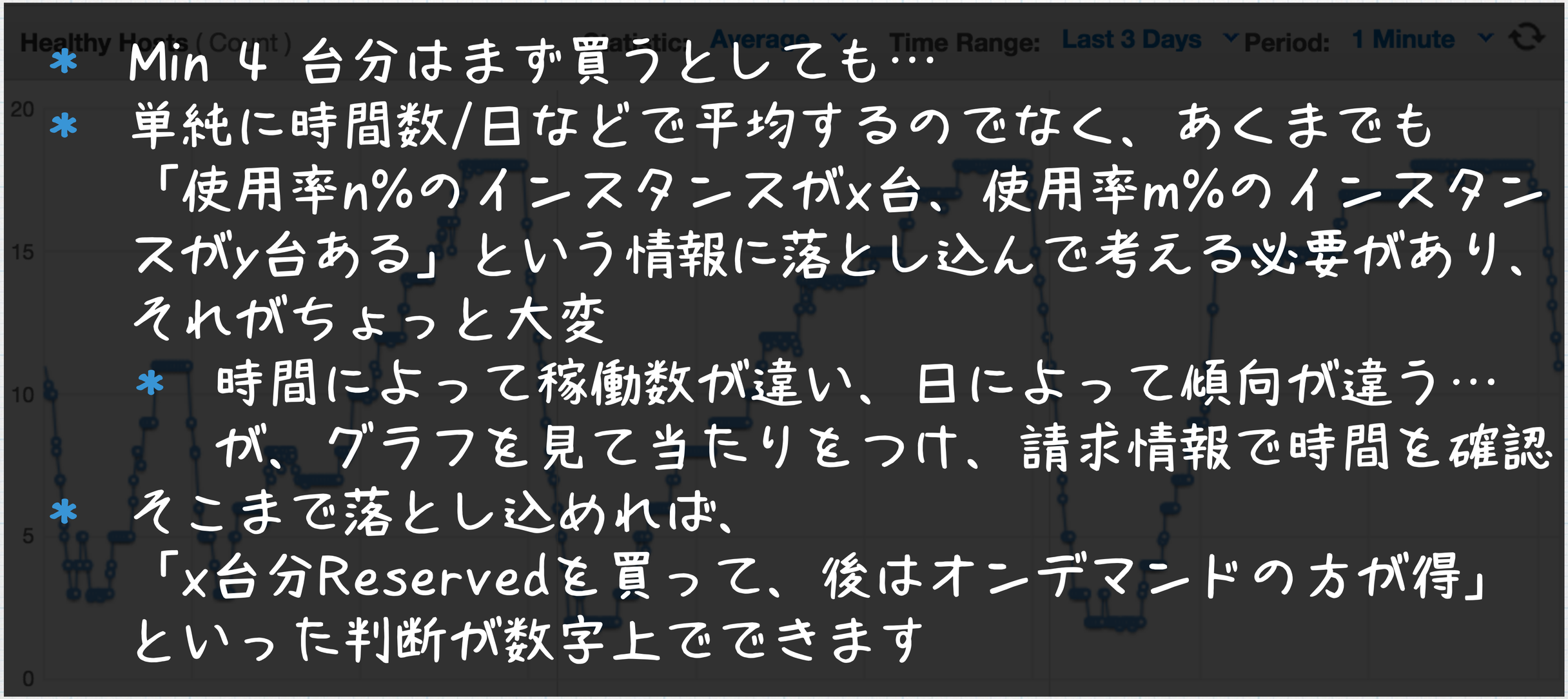
具体例/Reserved Instancesの試算

Reserved Instancesの試算



具体例/Reserved Instancesの試算

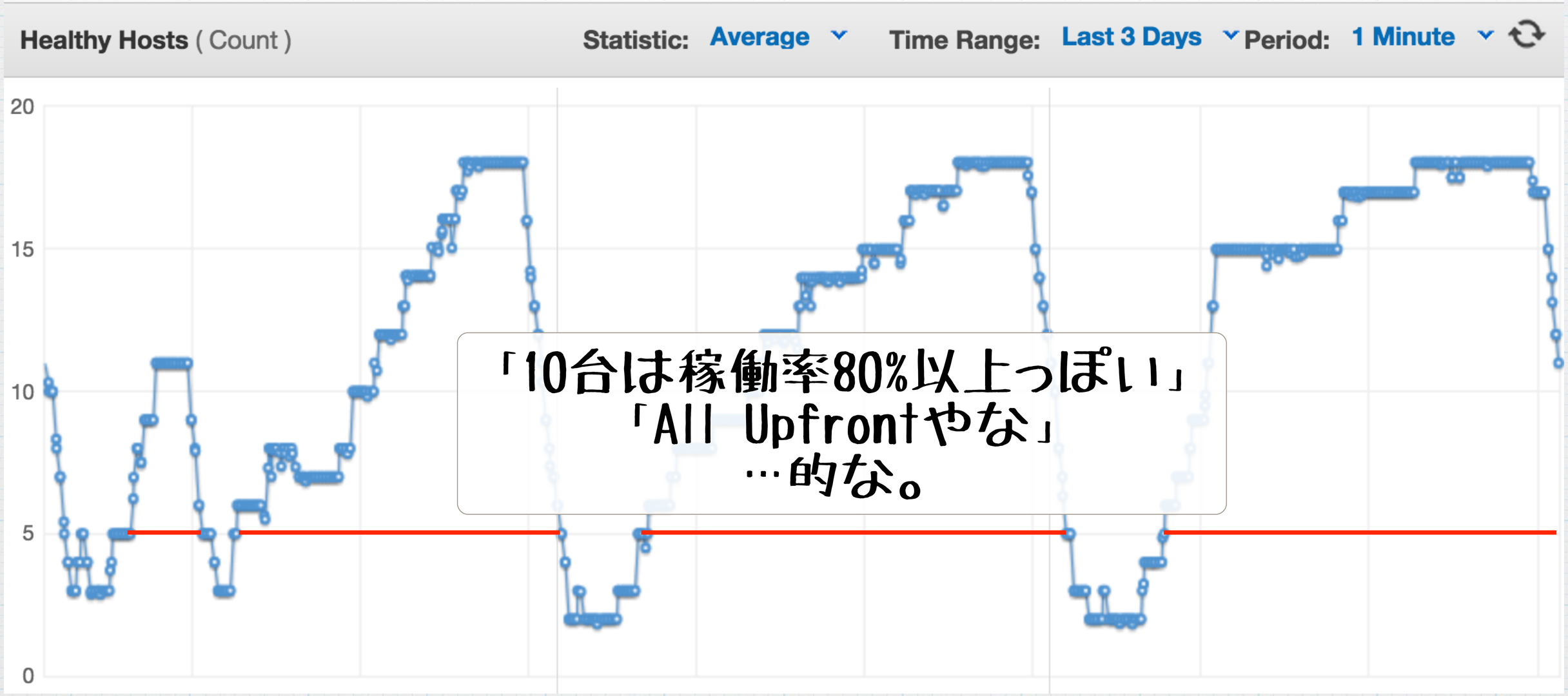
Reserved Instancesの試算



* Min 4 台分はまず買うとしても…
* 単純に時間数/日などで平均するのでなく、あくまでも「使用率n%のインスタンスがx台、使用率m%のインスタンスがy台ある」という情報に落とし込んで考える必要があり、それがちょっと大変
  * 時間によって稼働数が違い、日によって傾向が違う…
    が、グラフを見て当たりをつけ、請求情報で時間を確認
* そこまで落とし込めれば、「x台分Reservedを買って、後はオンデマンドの方が得」といった判断が数字上でできます

具体例/Reserved Instancesの試算

Reserved Instancesの試算



具体例/Reserved Instancesの試算

期待できる成果

http://j.mp/summit2015_tsukada_sheet

成果→http://j.mp/summit2015_tsukada_sheet

Before

用途	タイプ	購入モデル	時間単価	数	稼働時間/月 (31日)	イニシャル コスト	ランニング コスト
WebAPI	m3.2xlarge	オンデマンド	$0.810	8	5,952	$0.00	$4,821.12
Worker	c4.2xlarge	オンデマンド	$0.588	10	7,440	$0.00	$4,374.72
						$0.00 **¥0.00** **(0円)**	**$9,195.84** **¥1,131,088.32** **(113万1088円)**

After

用途	タイプ	購入モデル	時間単価	数	稼働時間/月 (31日)	イニシャル コスト	ランニング コスト
WebAPI	m3.large	Reserved (All Upfront)	$0.000	4	2,976	$4,000.00	$0.00
WebAPI	m3.large	Reserved (Partial Upfront)	$0.000	22	16,368	$10,714.00	$979.66
WebAPI	m3.large	オンデマンド	$0.203	(7.5台/日)	5,620	$0.00	$1,140.86
Worker	c4.large	Reserved (All Upfront)	$0.000	1	744	$862.00	$0.00
Worker	c4.large	Spot	$0.0205	(6.1台/日)	4,550	$0.00	$93.28
		※Reservedは全て1年間購入				**$15,576.00** **¥1,915,848.00** **(191万5848円)**	**$2,213.80** **¥272,296.79** **(27万2296円)**

成果→http://j.mp/summit2015_tsukada_sheet

Before

用途	タイプ	購入モデル	時間単価	数	稼働時間/月 (31日)	イニシャルコスト	ランニングコスト
WebAPI	m3.2xlarge	オンデマンド	$0.810	8	5,952	$0.00	$4,821.12
Worker	c4.2xlarge	オンデマンド	$0.588	10	7,440	$0.00	$4,374.72
						$0.00	**$9,195.84**
						[illegible]	**¥1,131,088.32**
						[illegible]円)	**(113万1088円)**

月々のお支払いが
(2ヶ月目からは)
-85万892円!!

After

用途	タイプ	購入モデル	時間単価	数	稼働時間/月	イニシャルコスト	ランニングコスト
WebAPI	m3.large	Reserved (All Upfront)	$0.000	4	[illegible]	$4,000.00	$0.00
WebAPI	m3.large	Reserved (Partial Upfront)	$0.000	2	[illegible]	$10,714.00	$979.66
WebAPI	m3.large	オンデマンド	$0.203	(7.5台/日)	5,620	$0.00	$1,140.86
Worker	c4.large	Reserved (All Upfront)	$0.000	1	744	$862.00	$0.00
Worker	c4.large	Spot	$0.0205	(6.1台/日)	4,550	$0.00	$93.28
		※Reservedは全て1年間購入				$15,576.00	**$2,213.80**
						[illegible]	**¥272,296.79**
						(191万[illegible]円)	**(27万2296円)**

成果→http://j.mp/summit2015_tsukada_sheet

before					
	1ヶ月目	2	3	4	5
月課金額	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00
累計額	¥1,131,088.00	¥2,262,176.00	**¥3,393,264.00**	**¥4,524,352.00**	**¥5,655,440.00**

after					
	1ヶ月目	2	3	4	5
月課金額	¥2,188,144.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00
累計額	**¥2,188,144.00**	**¥2,460,440.00**	¥2,732,736.00	¥3,005,032.00	¥3,277,328.00
before比	**-¥1,057,056.00**	**-¥198,264.00**	¥660,528.00	¥1,519,320.00	¥2,378,112.00

6	7	8	9	10	11	12
¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00
¥6,786,528.00	**¥7,917,616.00**	**¥9,048,704.00**	**¥10,179,792.00**	**¥11,310,880.00**	**¥12,441,968.00**	**¥13,573,056.00**

6	7	8	9	10	11	12
¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00
¥3,549,624.00	¥3,821,920.00	¥4,094,216.00	¥4,366,512.00	¥4,638,808.00	¥4,911,104.00	¥5,183,400.00
¥3,236,904.00	¥4,095,696.00	¥4,954,488.00	¥5,813,280.00	¥6,672,072.00	¥7,530,864.00	¥8,389,656.00

成果→http://j.mp/summit2015_tsukada_sheet

before			
	1ヶ月目	2	3
月課金額	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00
累計額	¥1,131,088.00	¥2,262,176.00	**¥3,393,264.00**

after			
	1ヶ月目	2	3
月課金額	¥2,188,144.00	¥272,296.00	¥272,296.00
累計額	**¥2,188,144.00**	**¥2,460,440.00**	¥2,732,736.00
before比	**-¥1,057,056.00**	**-¥198,264.00**	¥660,528.00

3ヶ月目からは
プラス収支に！

6	7	8	9	10	11	12
¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00
¥6,786,528.00	**¥7,917,616.00**	**¥9,048,704.00**	**¥10,179,792.00**	**¥11,310,880.00**	**¥12,441,968.00**	**¥13,573,056.00**

6	7	8	9	10	11	12
¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00	¥272,296.00
¥3,549,624.00	¥3,821,920.00	¥4,094,216.00	¥4,366,512.00	¥4,638,808.00	¥4,911,104.00	¥5,183,400.00
¥3,236,904.00	¥4,095,696.00	¥4,954,488.00	¥5,813,280.00	¥6,672,072.00	¥7,530,864.00	¥8,389,656.00

成果→http://j.mp/summit2015_tsukada_sheet

before			
	1ヶ月目	2	3
月課金額	¥1,131,088.00	¥1,131,088.00	¥1,131,088.00
累計額	¥1,131,088.00	¥2,262,176.00	**¥3,393,264.00**

after			
	1ヶ月目	2	3
月課金額	¥2,188,144.00	¥272,296.00	¥272,296.00
累計額	**¥2,188,144.00**	**¥2,460,440.00**	¥2,732,736.00
before比	**-¥1,057,056.00**	**-¥198,264.00**	¥660,528.00

3ヶ月目からは
プラス収支に！

12ヶ月後には
+838万9656円に！

	12
	¥1,131,088.00
	¥13,573,056.00

	12
	¥272,296.00
	¥5,183,400.00
	¥8,389,656.00

その他の
コストカットポイント

その他のコストカットポイント

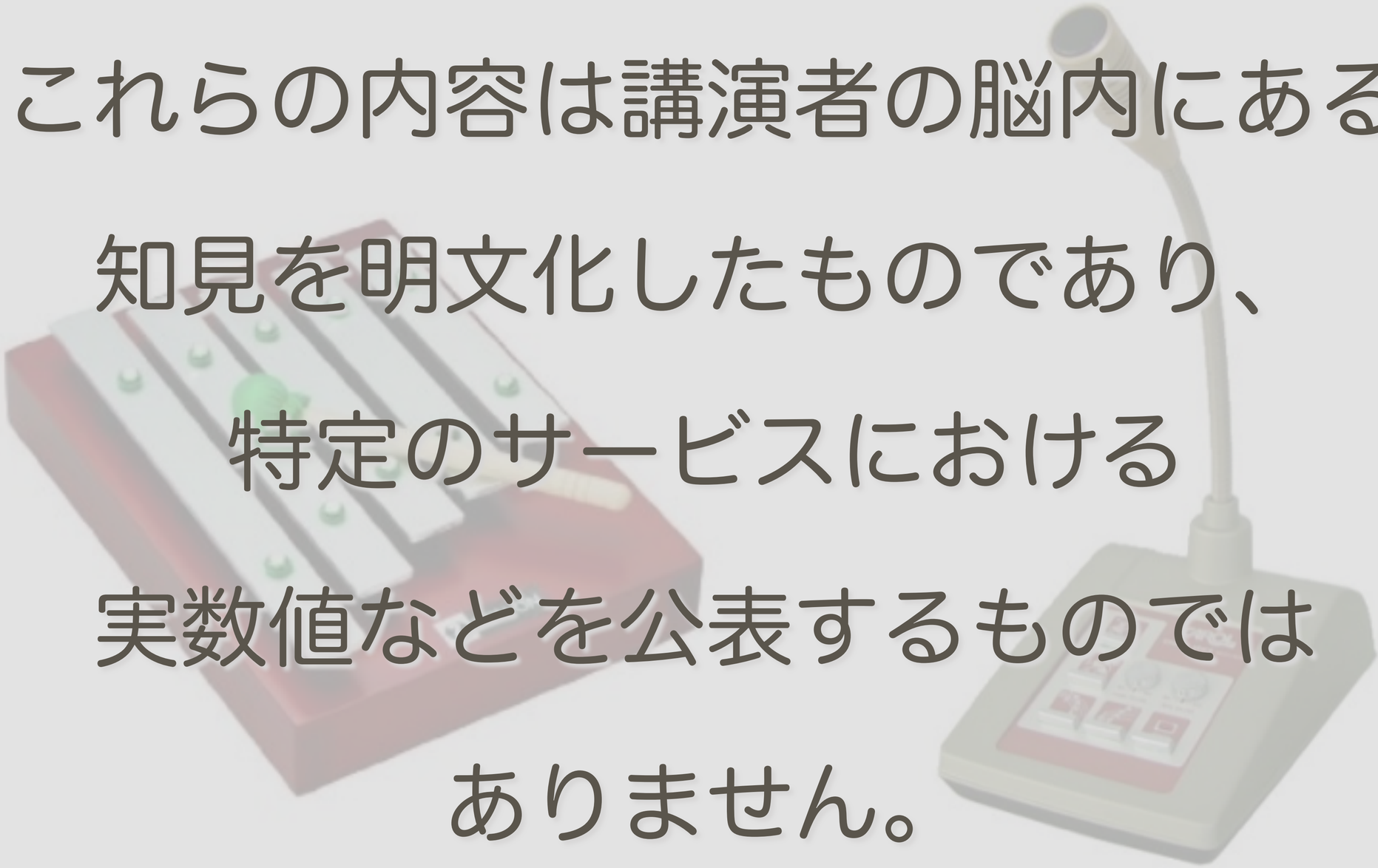
* 今日話してない各サービスの購入オプションも活用しましょう
  * RDS(Reserved Instances), ElastiCache(Reserved Cache Nodes), Cloudfront(Reserved Capacity), DynamoDB(Reserved Capacity) 等々
  * AWS Black Belt Techシリーズ リザーブドインスタンス & スポットインスタンス
    * http://www.slideshare.net/AmazonWebServicesJapan/20150325-aws-blackbeltrispotpublic
* 高騰しにくいタイプのSpot Instancesを使うと安心だったりして
  * 旧世代、c1.medium とか狙い目だったり
* Worker に食わせてたような処理、ものによっては Lambda で捌くとさらにめっちゃ安かったりする…なかなかクレイジーだと思います
  * 「今ならLambdaで実装可能/実装するわー」と思うJobある
* 今ならSpotの管理はSpotFleet APIを使うとさらに手軽に低価格インスタンスを徹底できるかなー
* (当然ながら)アプリケーションのチューニング。
  * 例えばSQL改善すればRDSのスペック下げられるとか。

今後の方向性など

今後の方向性、考えてること

* 職に就きます!!!!!!!!!!
* Lambdaをフルに使ったさらなる倹約アーキテクチャ作ってみたい
* さすがにそろそろ来る HTTP/2 時代に、ステートレスを前提にしていたScale戦略は果たして…？

Notes



これらの内容は講演者の脳内にある
知見を明文化したものであり、
特定のサービスにおける
実数値などを公表するものでは
ありません。

劇終
THE END

2015-06-03

塚田朗弘@akitsukada（無所属）