AWS Summit Tokyo 2016

GREE流！AWSをお得に使う方法

メンバー紹介

神谷侑司・今井祐介
- ゲームアプリ開発
- storageのコスト削減と効率的な利用

籠島啓介
- 広告システム開発
- 大量のログをなるべく安く簡単にさばく方法

堀口真司
- ゲームインフラ運用
- インフラツールもサーバレスで安く

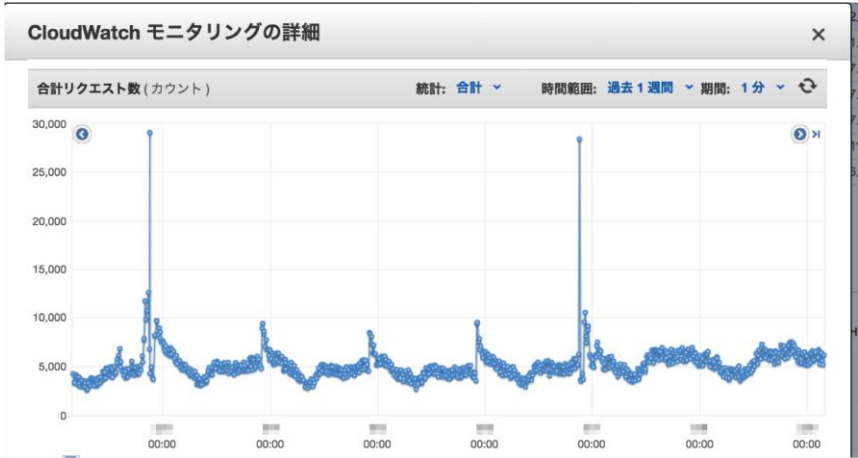
Agenda

- Service & System Overview
- Amazon DynamoDBの話
  - なぜ導入したのか
    - 利用するメリット
  - コストを抑えつつ利用するには
    - 開発事例からみるコストを抑えるためのTips
  - 実際に運用してみて出てきた問題＆課題
    - Throttled & Partitions
  - コストをおさえながら可用性を上げるには
    - Dynamic Dynamo 導入＆メリット
    - Dynamic Dynamo Tips

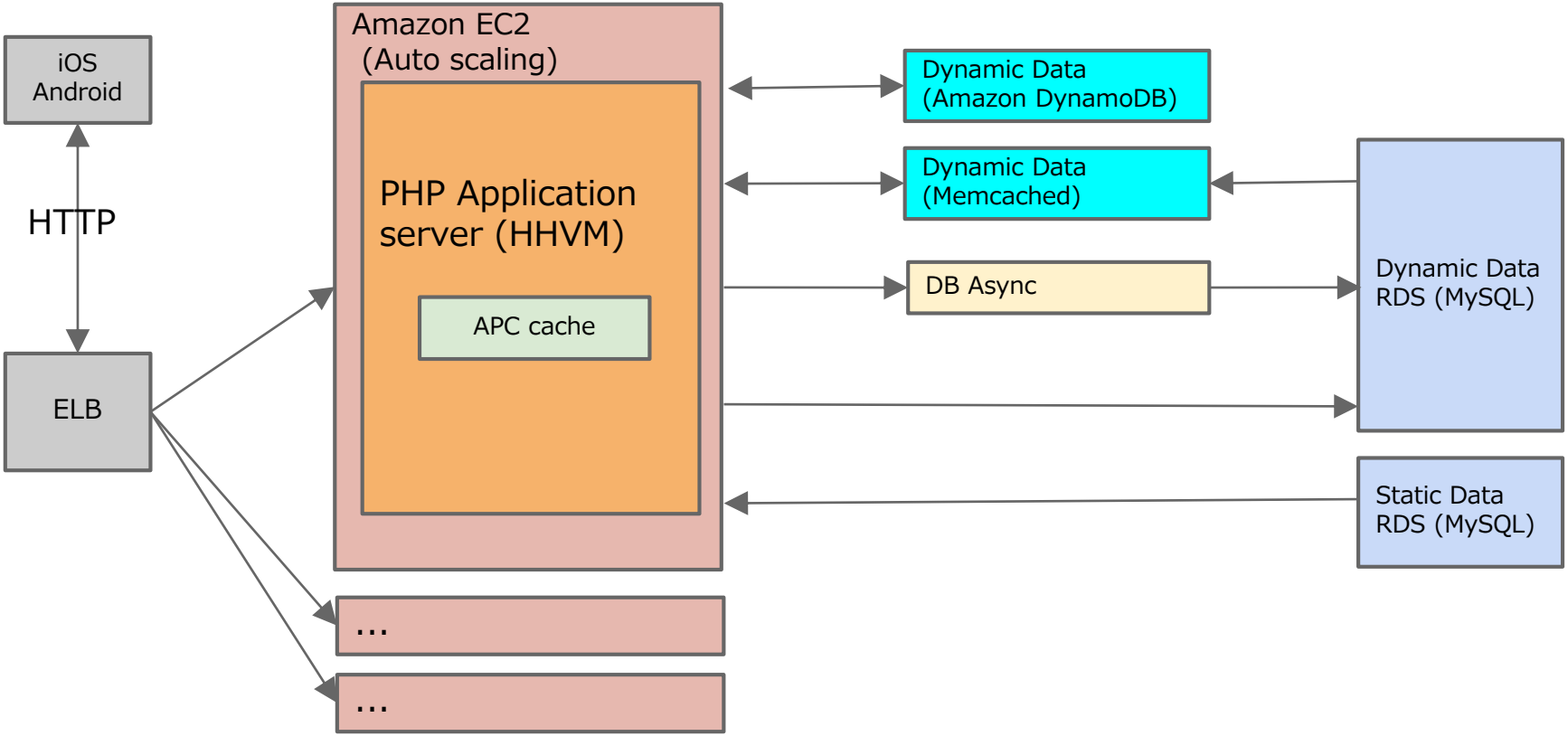
- Amazon RDSの話
  - Amazon Auroraを使ってコスト削減

Service Overview

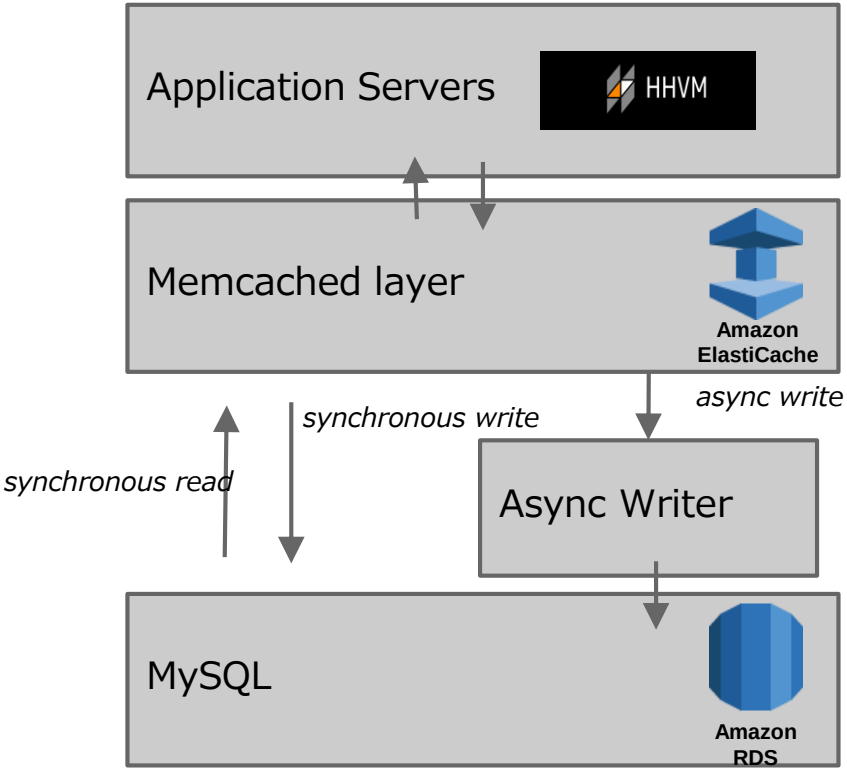
- ソーシャルゲーム(ネイティブアプリ)
  - 時間帯やイベント期間による負荷の差が大きい
    - web server: 12 ~ 50台
    - access: 3000 ~ 30000 / min
- RPG
  - 自分の町を開拓してリソースを獲得しバトルによって得た報酬でプレイヤーを強くしていく
  - 一週間で数種類のイベントを開催
    - イベントの種類によって負荷は異なる



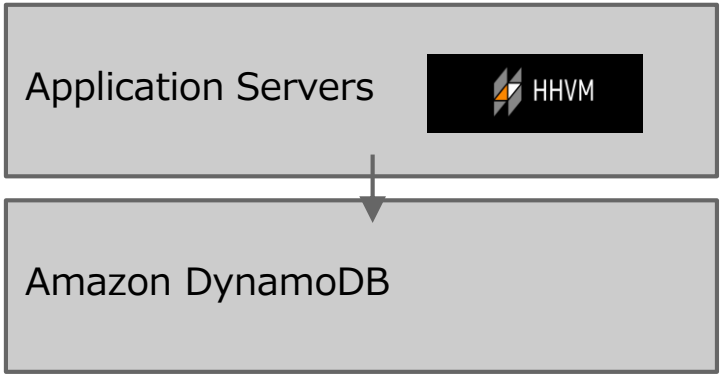
System Overview



System Overview



Current

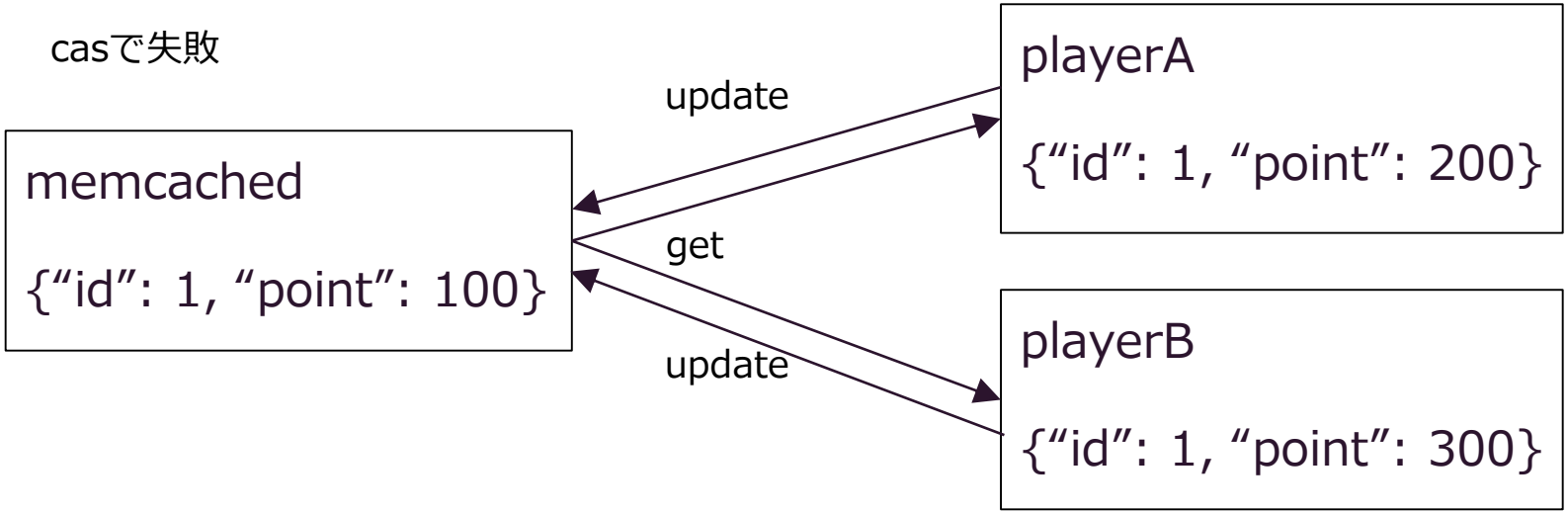


2015年秋頃から既存の一部機能をこちらの構成に移行し、新規機能の開発時はdynamoを利用した構成で開発

Amazon DynamoDB移行の動機

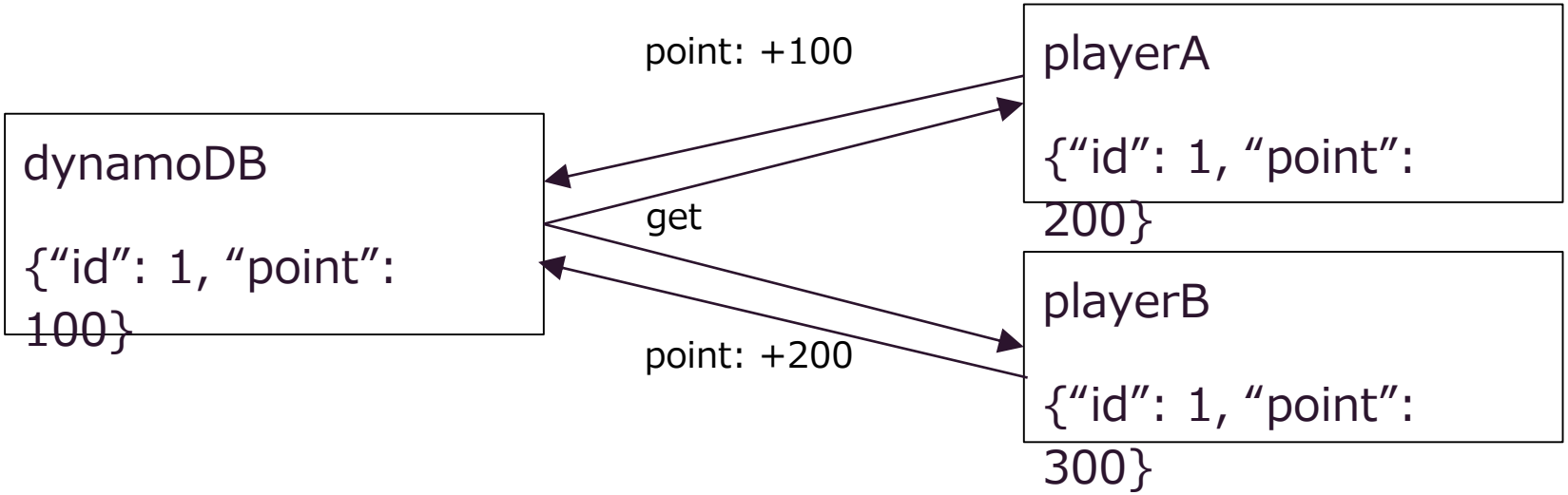
古い構成の問題点：複数プレイヤーが同じデータへ同時に書き込む際の競合

memcachedでplayerデータのjsonを保持していたため、以下の様な競合が発生しやすかった



Amazon DynamoDBのメリット

- atomic counterによる差分save
  - 値の変化分を指定して更新しているため、数値の更新が競合しない
  - Numberの更新の場合、casという操作自体が不必要



Amazon DynamoDBのメリット

- conditional saveによる条件指定
  - item内の特定の属性の値が指定した条件に合致する場合だけ更新

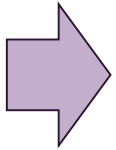
例えば、ゲーム中で

- アイテムAを早いもの勝ちで10個まで買える（1人当たり最大3個）

のようなルールを実現する場合は以下のようにするだけ

item定義

```
item_id: N,
count: N, // 購入時にcountをincrement
```



conditional save

```
UpdateItem
  count: +1
  when: count < 10
```

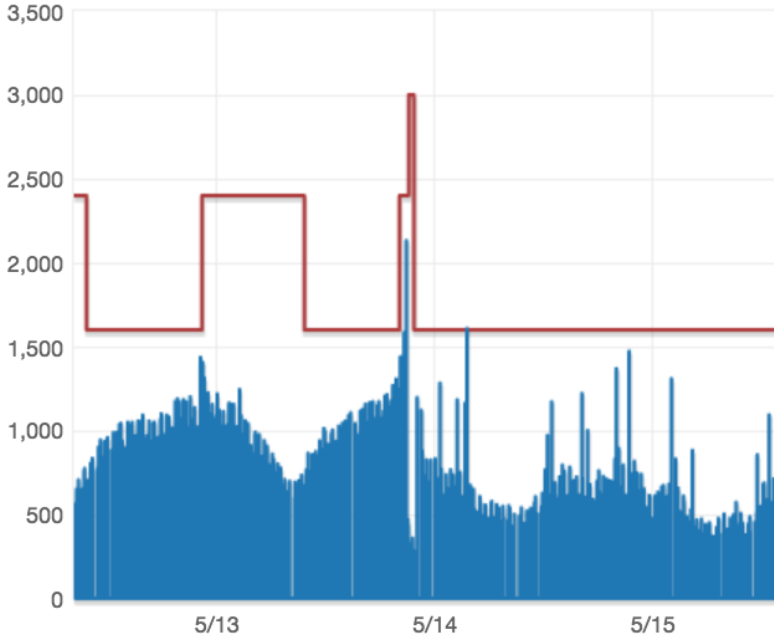
Amazon DynamoDBのメリット

- コスト可視化
  - Read/writeの上限は設定したThroughput Capacityによって制限
  - Throughput Capacityの大きさによって料金が決まる

read/writeの負荷の大きさ＝コスト大きさ

なので

負荷を減らせばコストも減り、
その効果はaws consoleですぐに確認できる!

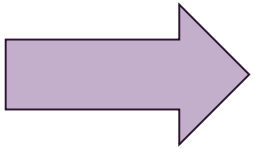


Amazon DynamoDBのコスト

- write: $0.0065 10unit / hour
- read : $0.0065 50unit / hour
- storage size: $0.25 / (GB*month)

つまり

- writeはreadの10倍（結果整合性のあるreadと比較）
- storage容量分は小さい



writeのthroughputを小さく保つことが重要

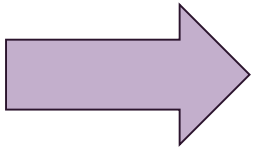
コスト削減の実例

1. **write capacityを抑える**
   - deleteせずに済むならしない
   - secondary indexをみだりに使わない

2. read capacityを抑える
   - randomにitem取得

Write時に消費されるCapacity

- PutItem, UpdateItem, DeleteItem, BatchWriteItemの実行で消費される
- 消費capacity = ceil(操作するitemのデータ量 / 1KB)



writeはitemにつき最低1unit必ず消費する

Tips

- **BatchWriteItemは並列に書き込めるだけで消費capacityは変わらないので注意**
  - ○ latencyの低減に役立つがコストの削減にはならない

コスト削減の実例

1. write capacityを抑える
   - **deleteせずに済むならしない**
   - secondary indexをみだりに使わない

2. read capacityを抑える
   - randomにitem取得

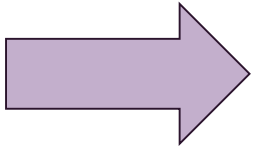
Deleteせずに済むならしない

例えば、historyのようなデータを考える

- ユーザのアクセス毎にPutItem
- 表示に必要なデータは最新の100件だけ

100件を超えた分を毎回deleteするのが簡単だがwrite capacityの消費UP

アクセス増大時のwrite負荷がさらに大きくなってしまう



バッチで非同期にdeleteする
or
定期的に新しいテーブルを作成する設計

コスト削減の実例

1. write capacityを抑える
   - deleteせずに済むならしない
   - **secondary indexをみだりに使わない**

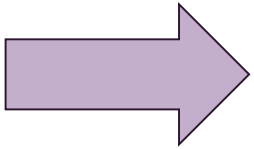
2. read capacityを抑える
   - randomにitem取得

IndexとWrite Capacity

射影されている属性に変化があった場合
indexへの書き込みに必要分のcapacityが消費される

つまり全属性を射影していた場合

indexの個数をnとすると**消費されるcapacityは(n+1)倍**となる



write capacityが数倍になるだけの価値があるか検討すべき

例えば

以下のようなitemを保持するテーブルがあったとする

```
player_id  : N (hash key)
some_id    : N (range key)
filter_key: S
…
```

※アプリ側ではfilter_keyによってfilterした結果がほしい

read時にitemを絞りこまなくとも問題ないのであれば、

indexを張ってwriteが2倍となるよりも全件取得の方がよい

（コストのみについて考えるならば）

コスト削減の実例

1. write capacityを抑える
   - deleteせずに済むならしない
   - secondary indexをみだりに使わない

2. read capacityを抑える
   - **randomにitem取得**

ランダムに結果を取得する

要求：あるイベントに参加しているユーザをランダムに取得する

単純な方法

- 全ユーザ分のitemを取得してアプリ側でfilterをかける

⇒ ユーザ数が多くなれば現実的ではない

ex. itemの大きさ：100B、ユーザ数：10,000、Queryを利用

100B * 10,000 / 4KB = 250uu ※Queryで消費されるcapacity = ceil(read総量 / 4KB)

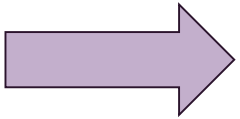
ランダムに結果を取得する

以下のようなkeyを定義し範囲指定で取得

event_id　　: N (hash key)
player_id　　: N (range key)
random_id : N (local secondary index)
…

- UpdateItemの際にrandom_idも更新
- item取得時は以下のようなQueryを発行
  - random_id < 乱数値
  - limit 10

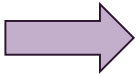
※実際はpartition分割への対策のためhash_keyはもう少し複雑



消費されるcapacityが抑えられる上、
ランダムに取得する処理をコードとして書く必要がない

まとめ

1. write capacityを抑える
   - deleteせずに済むならしない
   - secondary indexをみだりに使わない
2. read capacityを抑える
   - randomにitem取得



read/writeのアクセス数を算出し、必要な場合だけindexを利用する

負荷が膨大にならないのであればreadに負荷を寄せるのも一つの手

Amazon DynamoDBで実際に運用してみて出てきた問題・課題

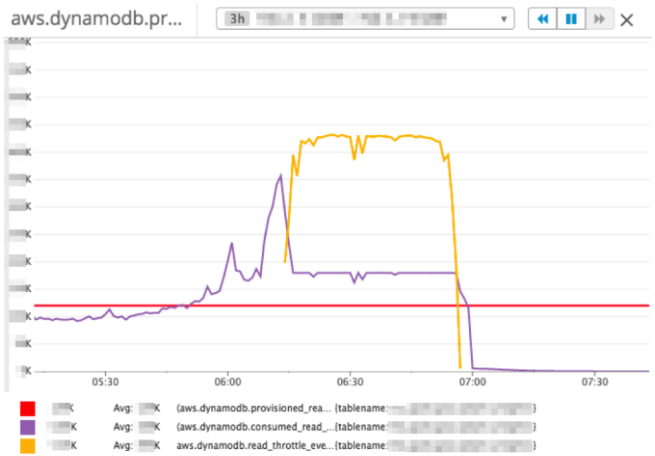
Amazon DynamoDBで実際に運用してみて出てきた課題

Throttling発生問題

- Throughput Capacityを超えるThroughputが発生した時に**400 ProvisionedThroughputExceededException** が返ってくる
  - ただし、過去5分間の空きThroughputを貯蓄する**Burst機能で一時的にカバー**してくれる
  - AWS SDK側で再送信してくれるが一定回数を超えると処理を失敗

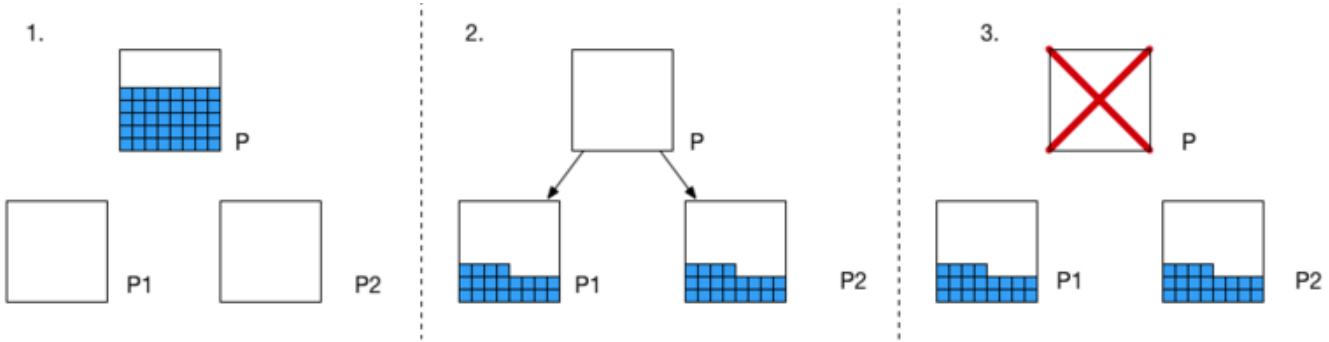
正常な状態

Throttlingしてしまった状態



Amazon DynamoDB Partitioning

- データ容量やThroughput Capacityによってpartition分割される
- ルール　○(Read Capacity / 3,000) + (Write Capacity / 1,000)
  ○10G毎に分割される
- 例　○CEIL(MAX(read:2000 + write:333, size:9G) = 1
  ○CEIL(MAX(read:2000 + write:334, size:9G) = 2
- 注意点
  - **○Capacityも分割される**
  - ○一度割れたら戻らない
  - ○Hotkeyがあるとthrottlingしやすくなる
  - ○Scanすると分割前よりthrottlingしやすくなる

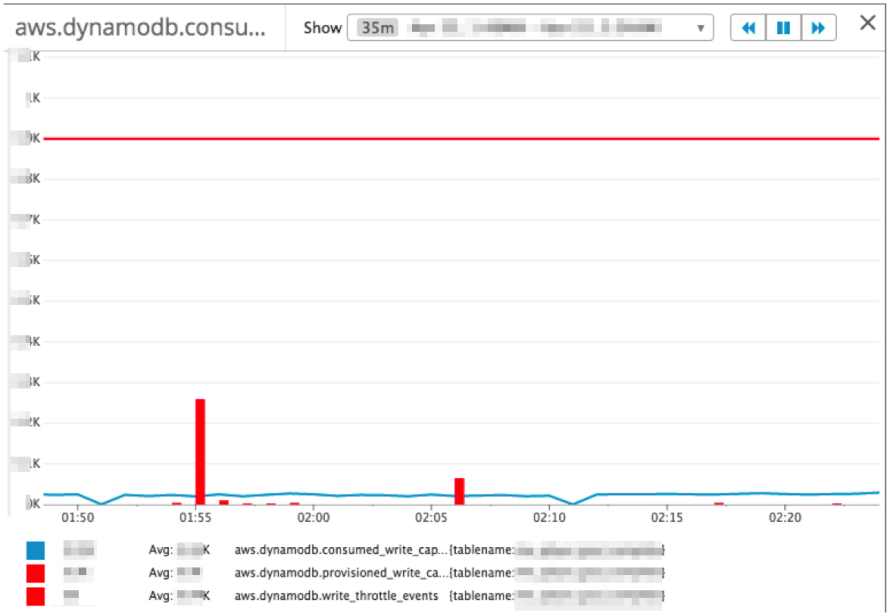


Amazon DynamoDBで実際に運用してみて出てきた課題

Hot Key/PartitionによりThrottlingが発生

Partition分割されているTableで、Throughputが一つのPartitionに集中してThroughput Capacityを超えてしまった

Throughput Capacity超えてないように見えるがThrottlingしてしまった状態



Partition数：30
Write Capacity：900
Partition毎のCapacity：**30**

Amazon DynamoDB Partitions - Hot Key/Partition Issue

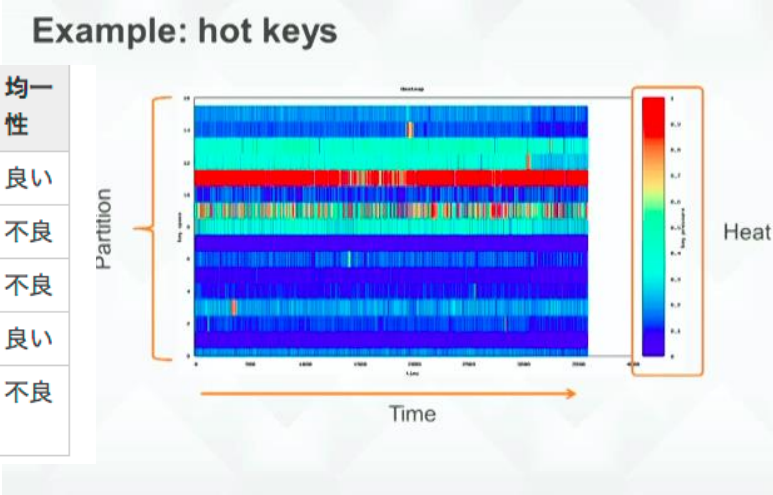
Hot Keys/Partition問題

- 一つのPartitionにアクセスが集中することで、そのPartitionでThrottlingが発生しサービスに影響が出しまう。

基本

- Table設計時にkeyが均等にアクセスされるようにする

パーティションキーの値	均一性
ユーザー ID（アプリケーションに多くのユーザーがある場合）	良い
ステータスコード（可能性のあるステータスコードが少しだけある場合）	不良
項目の作成日（直近の期間（日、時、分など）に切り上げられます）	不良
デバイス ID（各デバイスが比較的類似した間隔でデータにアクセスする場合）	良い
デバイス ID（追跡中のデバイスが大量にある場合でも、1 つのデバイスが他のすべてのデバイスよりもずっと人気がある場合）	不良



Hot Key/Partition Issue - どう対応したか (その１)

- 問題
  Tableサイズ増加でPartitionが細かく割れてしまい少しの偏りでThrottlingが発生
- 対応
  ○費用を考えるとCapactiyを上げたくはない
  ○アーカイブ用Tableを作りそこにBatchで少しずつ移動していった
  ○古いDataを同時に取得することがなくなったのでThrottlingが減った

Hot Key/Partition Issue - どう対応したか (その２)

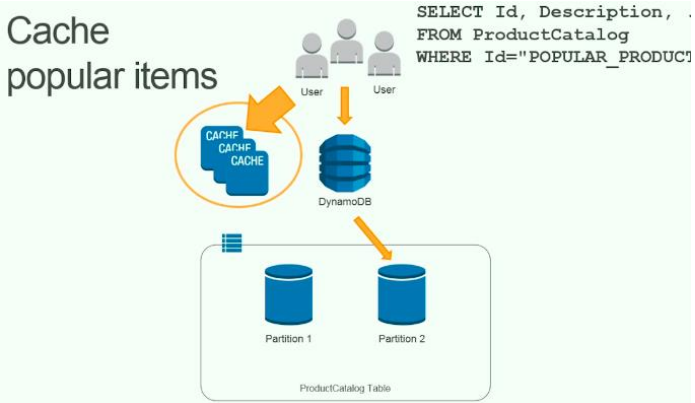
- 問題

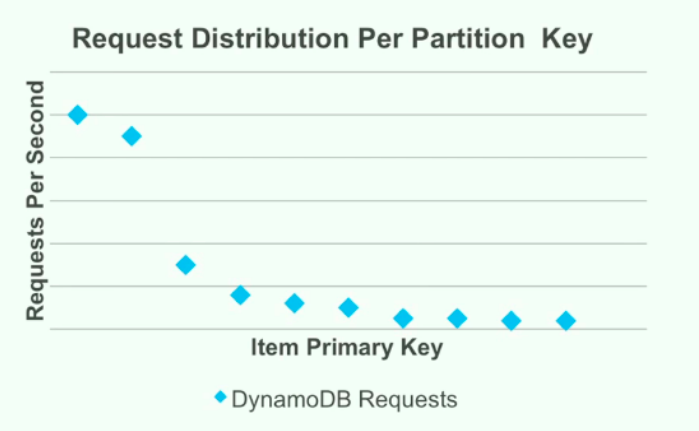
  一部のKeyの**読み込み**が非常に激しくThrottlingしていた

- 対応

  Memcached / APC cache等に読み込みをキャッシュ

  NoSQLとはいえAmazon DynamoDBはThroughput毎にお金がかかるため　Cacheに載せたほうが安い





※Amazon DynamoDBの仕様上1 Paritionの利用可能な最大Read Throughputは6000 units/秒

Hot Key/Partition Issue - どう対応したか (その３)

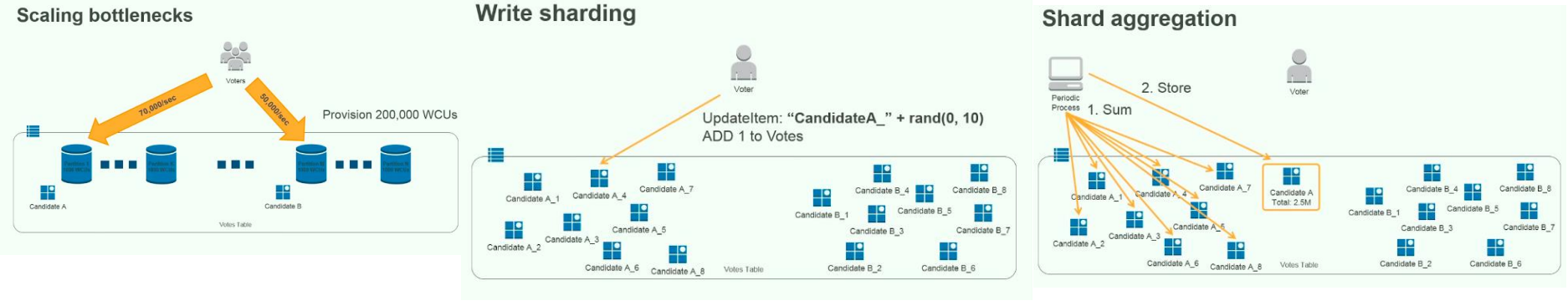
- 問題

  一部のKeyの**書き込み**が非常に激しくThrottlingしていた

- 対応

  TableをShardingして、Writeを複数Tableに分散させた

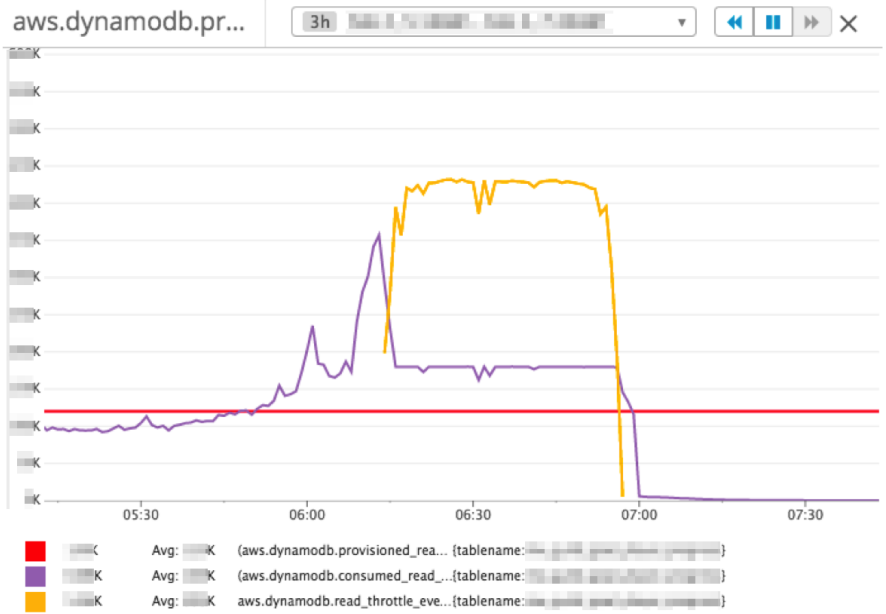
  ※読み込み時は複数TableからDataを取る必要がある



※Amazon DynamoDBの仕様上1 Paritition の利用可能な最大 Write Throughtputは1000 units/秒

Amazon DynamoDB Throughput Bursting - 気をつける事

- 過去5分間の空きThroughputを貯蓄してくれる。（急激な上昇への対策）

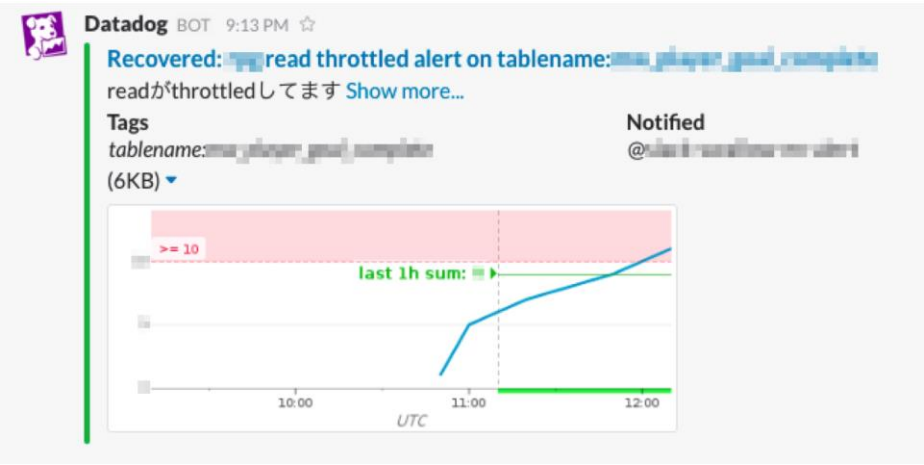


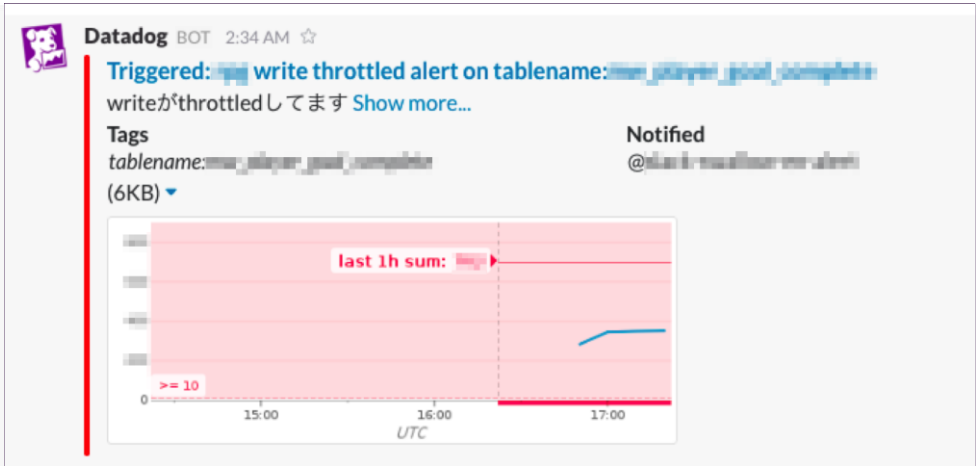
ただしBurst機能は常に保証していないので前提にしてはならない

- Capacityを上げる時等バックグラウンド処理が走るとThrottlingされた事も（GSI生成/Partition分割時等）

可用性を維持しながらコストを抑えるには

- Amazon DynamoDBはTable毎に明確にCapacityを設定する必要がある
  - ○ イベント毎に負荷の見積もり行うが想定外の盛り上がりの可能性もある
  - ○ コストは抑えたいが可用性のためにCapacityは下げたくない
  - ○ 見積もりの人的コストも毎回発生する
  - ○ **AutoScaleしてほしい**



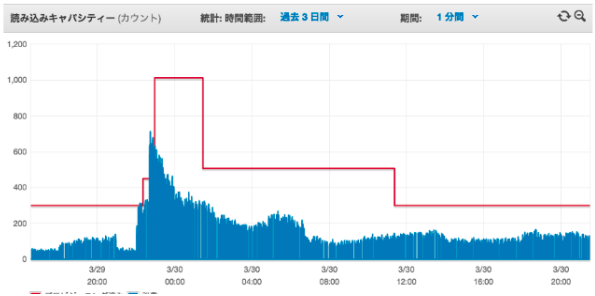


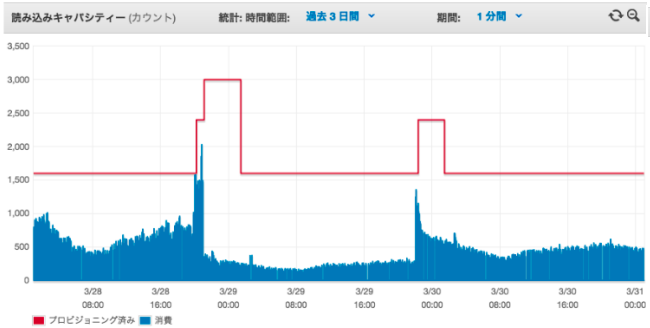
可用性を維持しながらコストを抑えるには

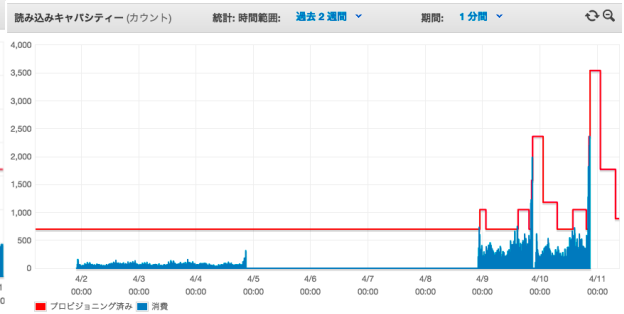
- ということでDynamic Dynamoの導入しました

Dynamic Dymamoとは

Amazon DynamoDBのCapacityを自動調整してくれるサービス。Amazon CloudWatchのCosumedCapacity Metricsの数値を元にRead/Write Capacityの設定を自動的に上げ下げしてくれる。細かい設定が可能で最小・最大なども指定可能







可用性を維持しながらコストを抑えるには

Dynamic DynamoとDynamo DB Partitionsの関係

- **一度分割されたら戻らないのでScale downした時にThrottlingされる可能性**
  - 900 -> 1100 でScale upした時にPartitionが割れるので1Partition内のCapacityは900 -> 450に落ちる。Hot keyがある場合 Throttlingされる可能性あり

- **Hot Key/Partition問題の解決にはならない**
  - Hot Key/Partitionの場合Amazon CloudWatchではCapacity上限に近いわからないのでScale upできない

- **大規模なプロモを実施する前には事前にマニュアルで上げておく必要がある**
  - Amazon CloudWatchは１分毎に集計、Dynamic Dynamoも数分毎に実行するのでCapacityが上がるまで数分かかる。事前にマニュアルでCapacityを上げて大事な時間のサービスに影響が出ないように準備する必要がある。

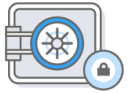
Amazon Auroraを使ったコスト削減

Amazon RDS for MySQLのコスト削減について

コスト削減を目的としてAmazon Auroraを使ってmaster統合を実施

- MySQL compatibilityなのでCodeの変更は不要
- masterとslaveの数を減らす事でinstance費用が減る
- 移行後にinstanceもdowngradeできればさらに減る
  - Amazon RDS for MySQLに比べてIO-boundになりにくいためinstanceのCPUを使えるようになった

高性能



高い安全性

MySQL との互換性

高いスケーラビリティ

高可用性と耐久性

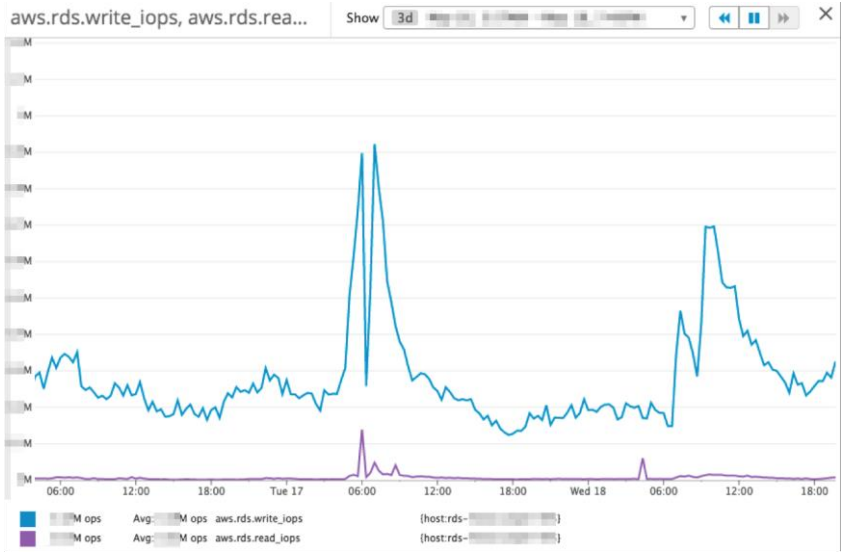
完全マネージド型

Amazon RDS for MySQLのコスト削減について

コスト削減と可用性向上を目的としてmaster統合を実施

- IOPS費用がProvisioned IOPS（月額固定）からほぼ従量課金になるので負荷の変化が大きいゲームにおいてはIOPS費用が減る

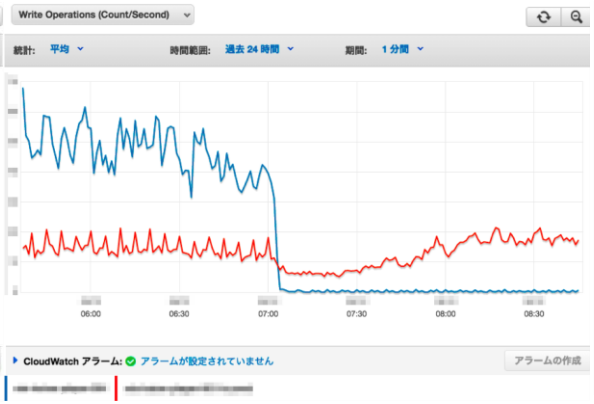
イベント開始・終了の負荷が非常に高いが、平常時は落ち着いている

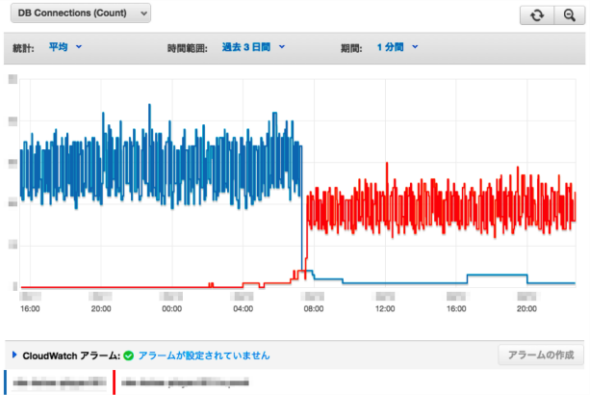


Amazon RDS for MySQLのコスト削減について

移行した結果

- Write Latency/IOPSとDB Connectionsが減少
  - Binlogをoffにしている事が関係していると思われる。同じIOPS前提でコストを計算していたため、想定よりコストが下がりそう。





  - その他
    - Amazon CloudWatchのMetricsが非常に増えているので地味に便利
    - 当GameではWriteがAsyncだが、Sync WriteなGameの場合Latencyの向上が見込めそう

Amazon RDS for MySQLのコスト削減について

移行でのTips

- RRはすべて一時利用不可に
  - Amazon RDS for MySQLの場合
    - Master down時でもslaveは利用可能
  - Amazon Auroraの場合
    - Master down時には数秒間全slaveが利用不可

- Amazon RDS for MySQL 5.5とはReplicationできない
  - 事前に5.6にUpdateが必要

大量のログをなるべく安く簡単にさばく方法

Glossom 開発部 籠島啓介

目次

- Glossom 会社概要
- Glossomで扱うログ
- AWSをなるべく安く便利に使う
- 最後に

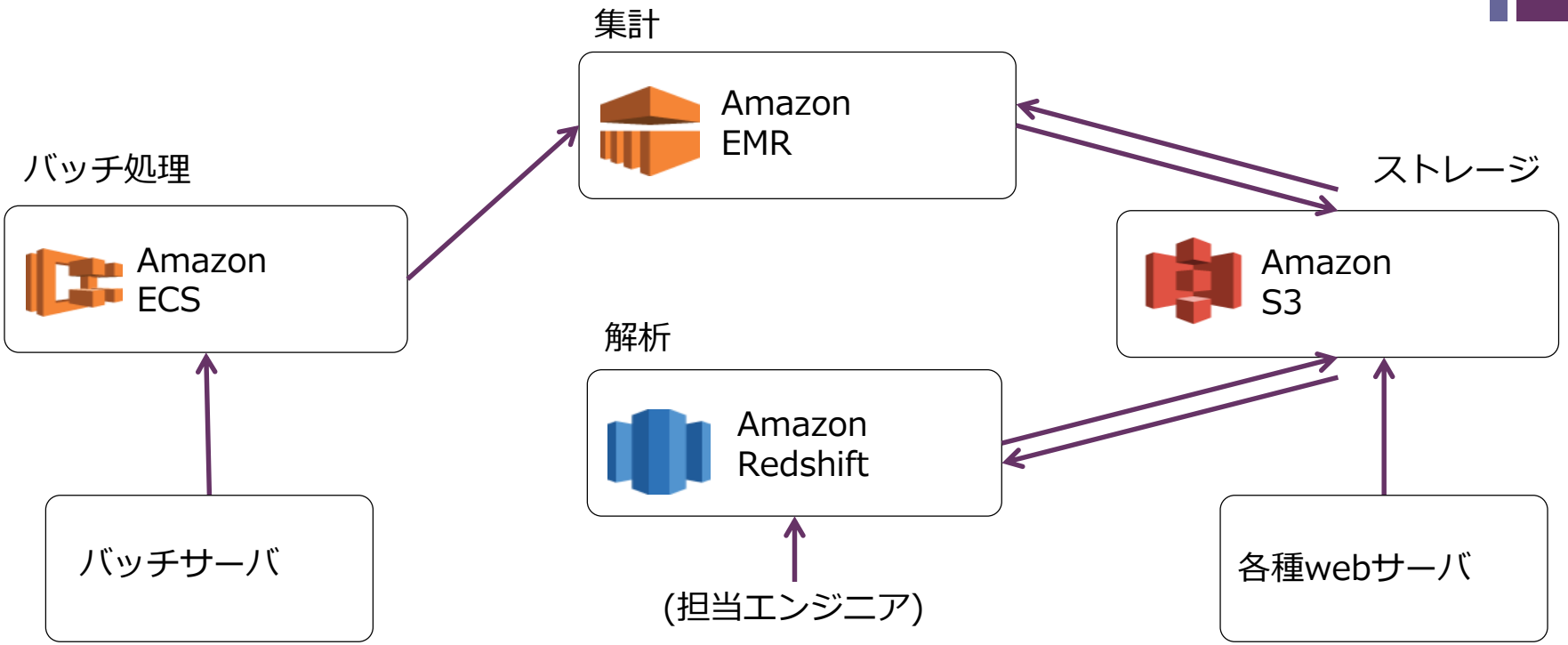
Glossom 会社概要

- 元グリー広告統括部が分社化
- 広告のシステムの開発・運用など

Glossomで扱うログ

- 10億弱/日 の 入札ログ (DSP)
- 毎時集計してレポートに反映
- 商品づくりのために入札ログの解析も必要(頻度低)
- これらのログの処理にAWSを利用

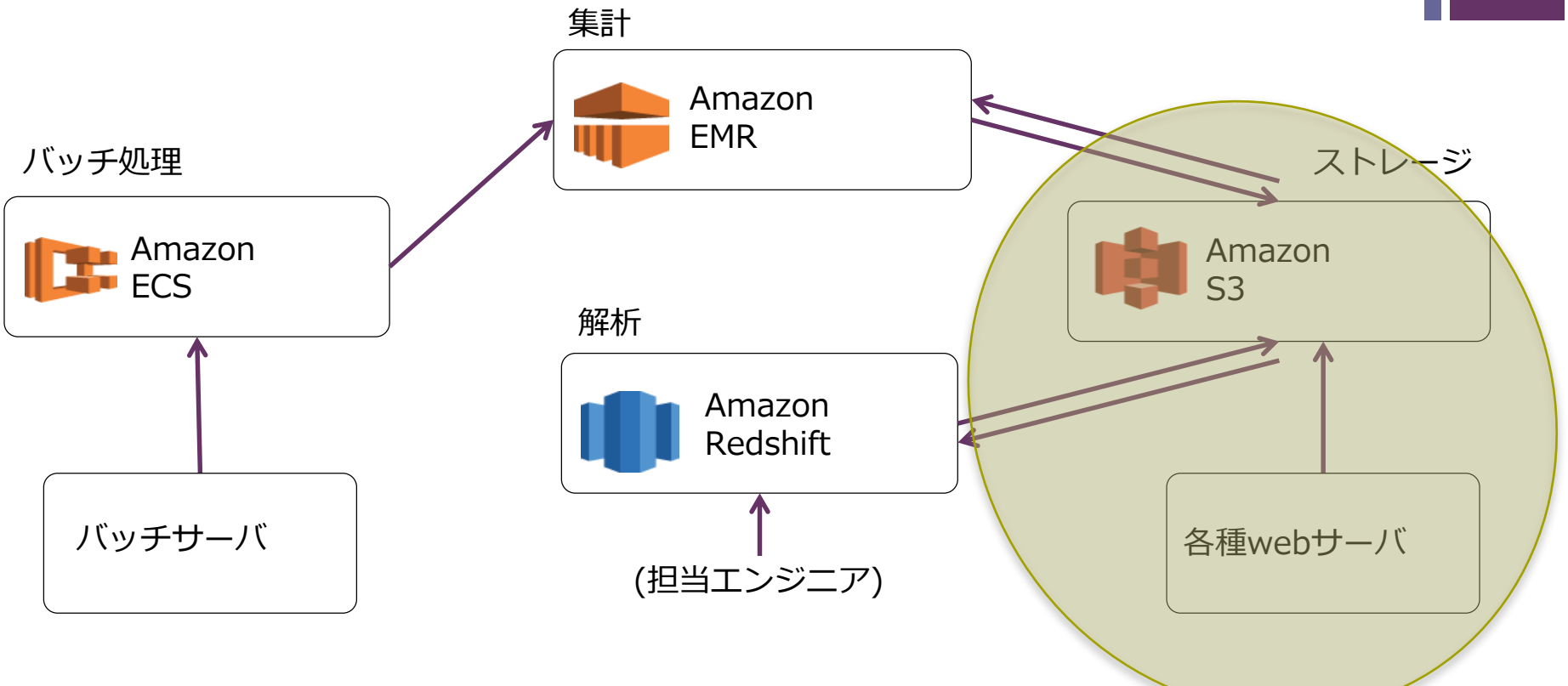
集計システム全体図


集計
Amazon
EMR
バッチ処理
Amazon
ECS
ストレージ
Amazon
S3
解析
Amazon
Redshift
バッチサーバ
(担当エンジニア)
各種webサーバ

AWSをなるべく安く便利に使う

- 生ログをltsv形式にする
- 生ログをs3に直接fluentdでアップロード
- batchにAmazon ECSを使う
- Amazon EMR + SPOTインスタンス
- Amazon EMR + S3Dist Cp
- Amazon Redshiftのインスタンス制御

集計システム全体図


集計
Amazon
EMR
バッチ処理
Amazon
ECS
ストレージ
Amazon
S3
解析
Amazon
Redshift
バッチサーバ
(担当エンジニア)
各種webサーバ

生ログをltsvにする

{“time”:1462174589,“column1”:”value1”, … “columnN”:”valueN”}

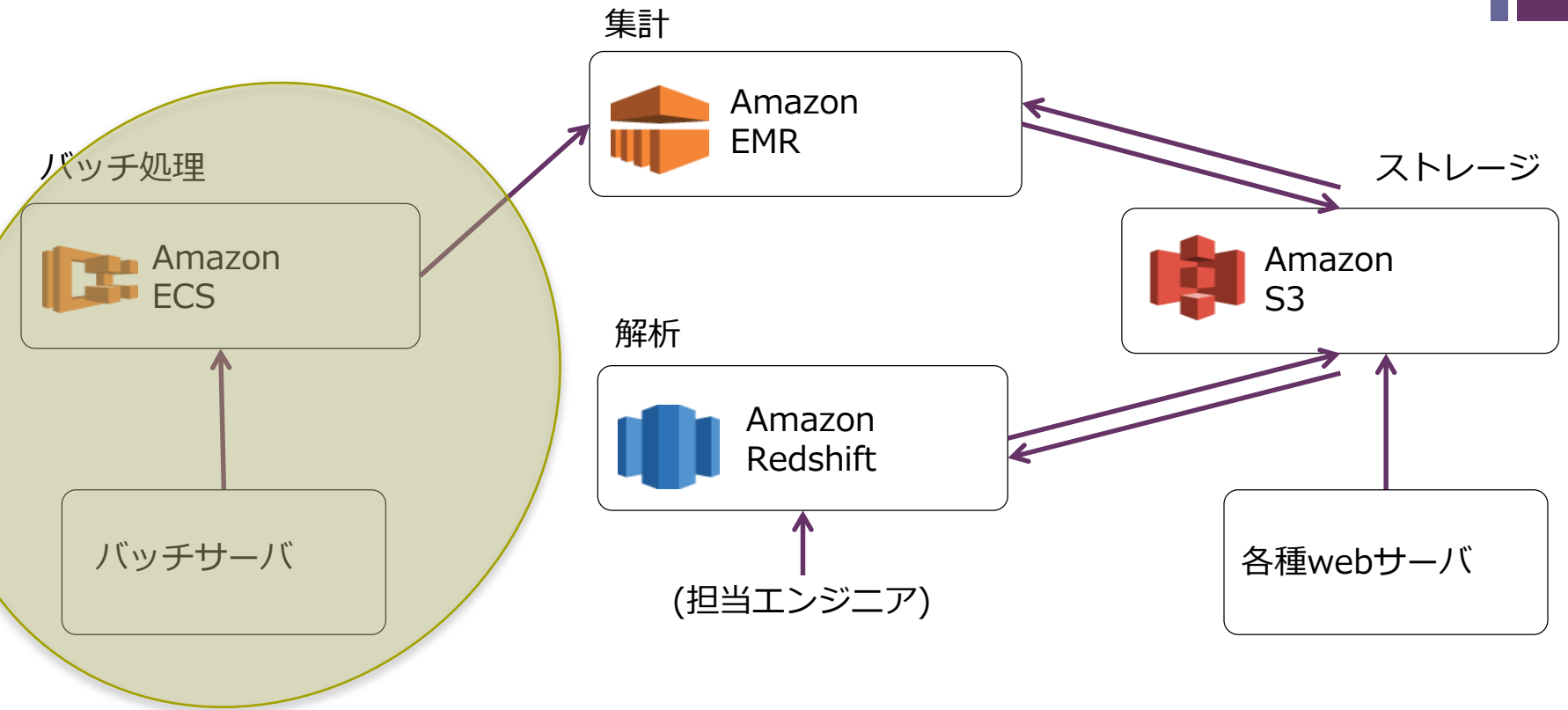
time:1462174589[tab]column1:value1[tab] … [tab]”columnN”:”valueN”

- ltsv形式だとparseの計算量が少ない
  - 40カラムほどのデータの単純集計で集計時間が約半分になった

生ログをAmazon S3に直接fluentdでアップロード

- Amazon S3はコスパが良い→積極的に利用
  - input時 転送量無料
- fluentdのreceiverは立てない→障害ポイントが減る
- fluentdのプラグインが便利
  - fluent-plugin-s3 (td-agent標準で付属)
- サービス要件上リアルタイム性が不要なのでkinesis等は使わず

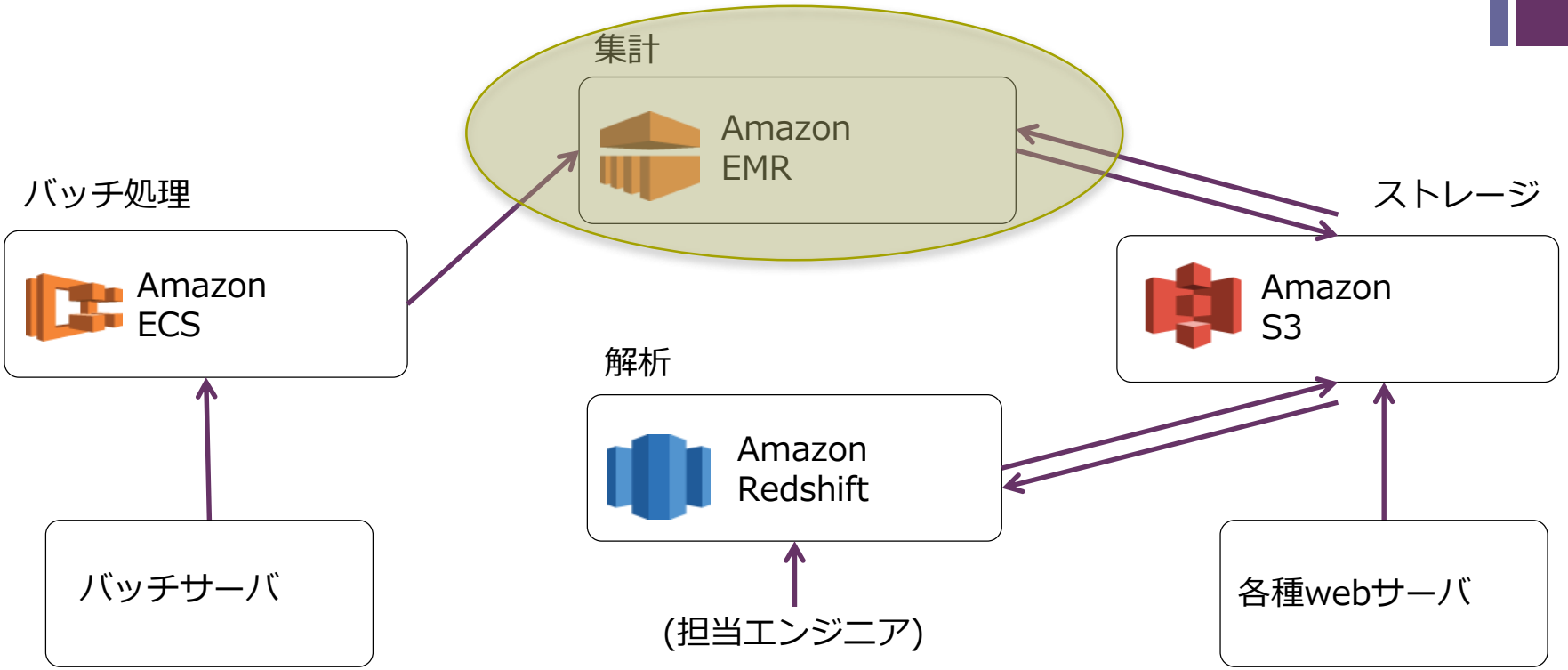
集計システム全体図


集計
Amazon
EMR
バッチ処理
Amazon
ECS
ストレージ
Amazon
S3
解析
Amazon
Redshift
バッチサーバ
(担当エンジニア)
各種webサーバ

batchにAmazon ECSを使う

- batch処理をするインスタンスとして Amazon ECSのクラスタを利用
  - batchの冗長化
  - スケーラビリティ向上
  - batch処理ごとに異なる環境を作れる

集計システム全体図


集計
Amazon
EMR
バッチ処理
Amazon
ECS
ストレージ
Amazon
S3
解析
Amazon
Redshift
バッチサーバ
(担当エンジニア)
各種webサーバ

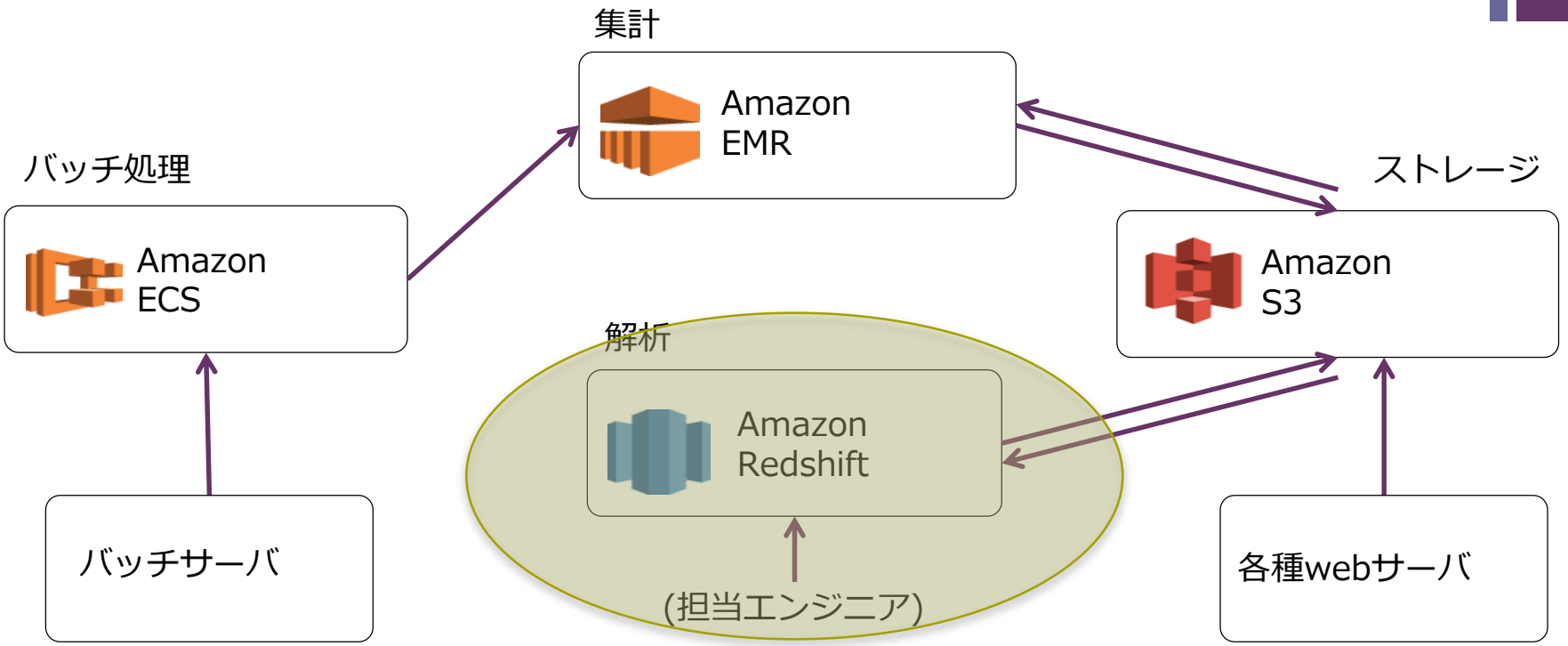
Amazon EMRでSPOTインスタンス

- 利点
  - 価格を大幅に抑えられる (1/2 ～ 1/8 程度)
- 欠点
  - インスタンスを上げられなかった時のハンドリングが面倒
  - 10分経っても立ち上がらなかったらON DEMAND

Amazon EMR + S3DistCp

- 集計処理前にAmazon EMRクラスタのhdfsにデータを入れておく
  - クラスタのDisk IOをフルに活用

集計システム全体図


集計
Amazon
EMR
バッチ処理
Amazon
ECS
ストレージ
Amazon
S3
解析
Amazon
Redshift
バッチサーバ
(担当エンジニア)
各種webサーバ

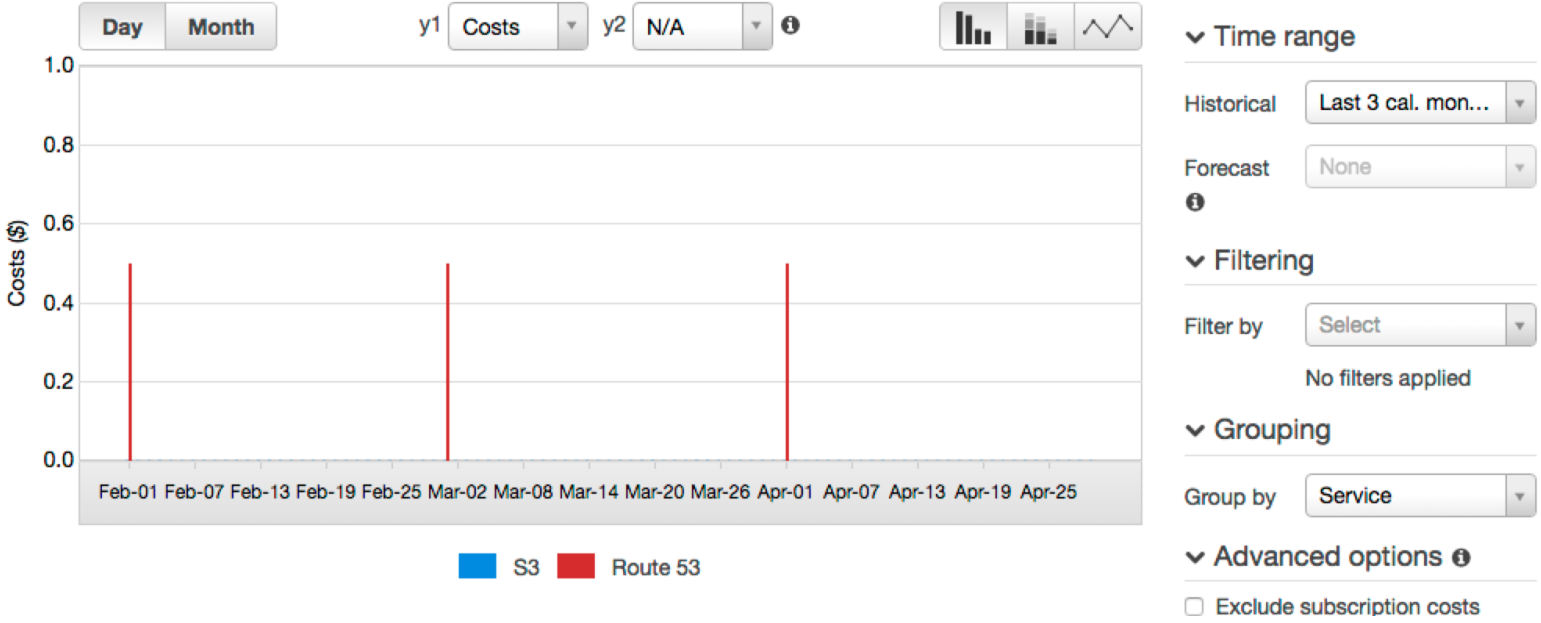
Amazon Redshiftのインスタンス制御

- 使う時だけインスタンスを立ち上げる
- データのインポートスクリプトを用意
- 元データは集計時に予め作っておきAmazon S3へ

最後に

- AWSは使い方を工夫することで
  安く便利に利用できる
- 一番大事なのはこまめなコストチェック

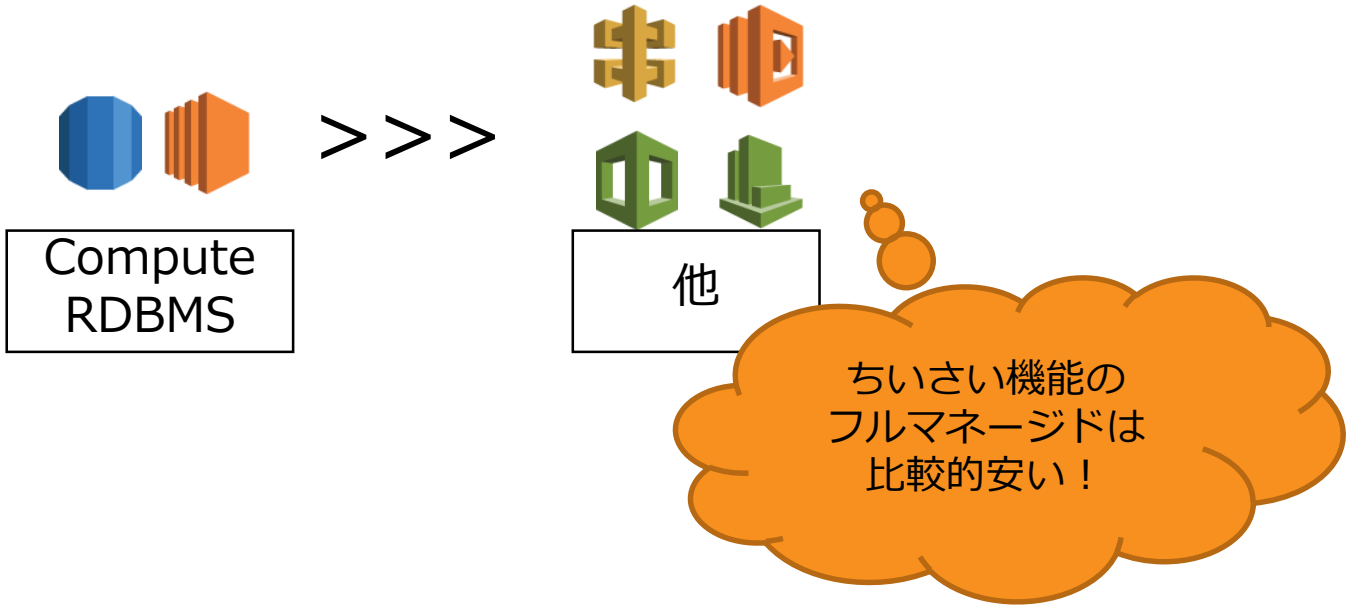
Cost Explorer



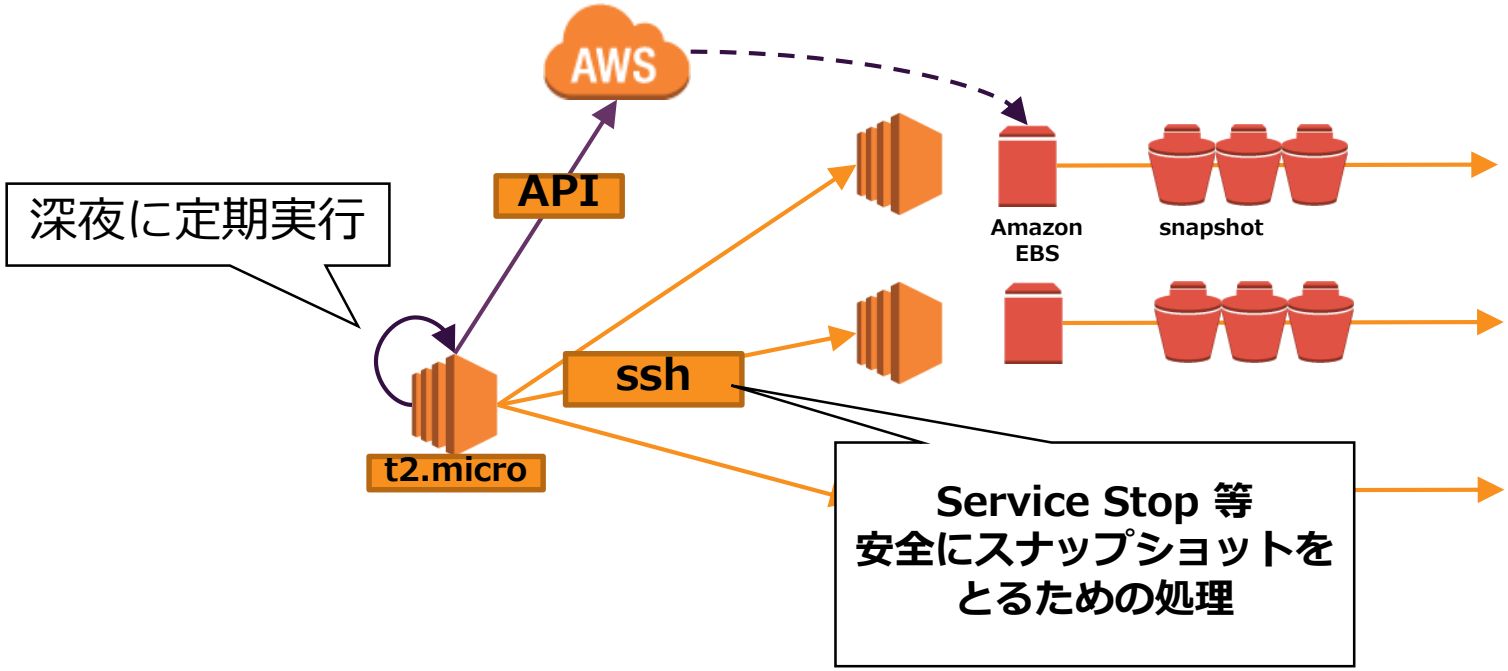
インフラツールも
サーバレスで安く

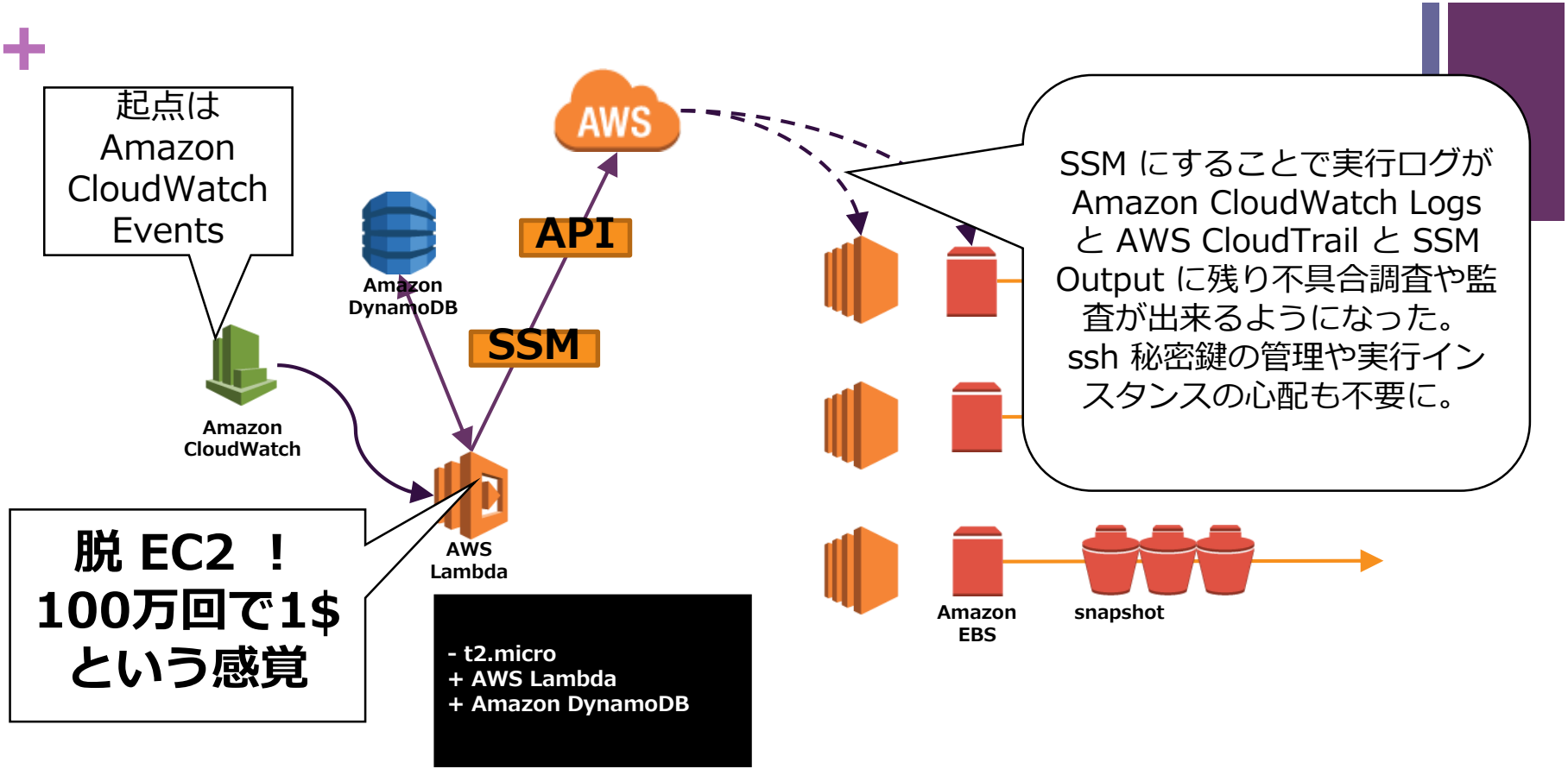
開発本部 インフラストラクチャ部
堀口 真司

お高い順



Amazon EBS スナップショット取得事例

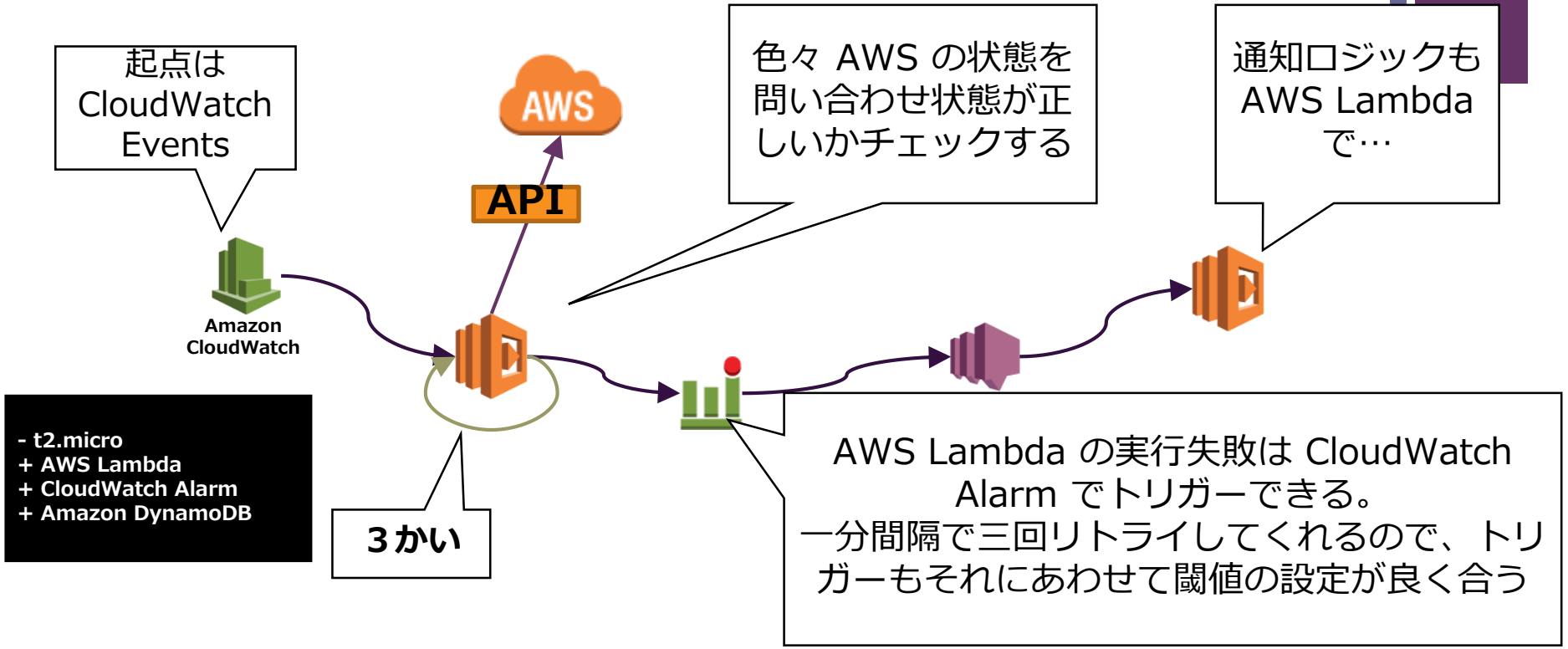



起点は
Amazon
CloudWatch
Events
Amazon
CloudWatch
Amazon
DynamoDB
AWS
API
SSM
AWS
Lambda
SSM にすることで実行ログが
Amazon CloudWatch Logs
と AWS CloudTrail と SSM
Output に残り不具合調査や監
査が出来るようになった。
ssh 秘密鍵の管理や実行イン
スタンスの心配も不要に。
Amazon
EBS
snapshot
脱 EC2 ！
100万回で1$
という感覚
- t2.micro
+ AWS Lambda
+ Amazon DynamoDB

コツ

- AWS Lambda 実行が失敗する可能性がある
  - 状態を Amazon DynamoDB に入れておく
  - 進捗するたびに Amazon DynamoDB に Put する
  - (ただし、一度も実行時の障害を観測したことは無い)
- 同時実行される可能性がある
  - Amazon DynamoDB で悲観ロックを行う
- 時間がかかる可能性がある
  - 寿命は最大でも５分、関数を分ける
  - ５分以内に終わりそうな関数 → 成功したら次の関数

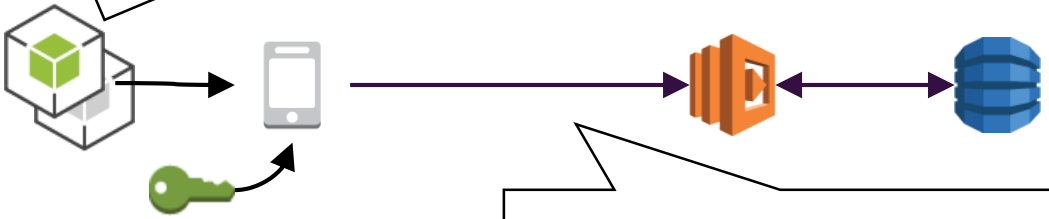
監視ロジックの例


起点は
CloudWatch
Events
AWS
API
色々 AWS の状態を
問い合わせ状態が正
しいかチェックする
通知ロジックも
AWS Lambda
で…
Amazon
CloudWatch
- t2.micro
+ AWS Lambda
+ CloudWatch Alarm
+ Amazon DynamoDB
3かい
AWS Lambda の実行失敗は CloudWatch
Alarm でトリガーできる。
一分間隔で三回リトライしてくれるので、トリ
ガーもそれにあわせて閾値の設定が良く合う

AWS Lambda とゲーム API 系

- Amazon EC2 を使わない
  - 高いので API Gateway も使わない
  - 小さいレスポンスなら1リクエストあたり、10倍ぐらい差がつくことも。
- ゲームは HTTP(S) である必要が無いので AWS-SDK を使う
  - 直接 Invoke する。しかし Native SDK の内部実装では libcurl+openssl
  - Invoke できる Credentials をアプリに入れる
- ストレージはもちろん Amazon DynamoDB

AWS SDK をアプリへ組み込み



Credentials も入れちゃう

API Gateway をつかわず
Lambda 直行で激安

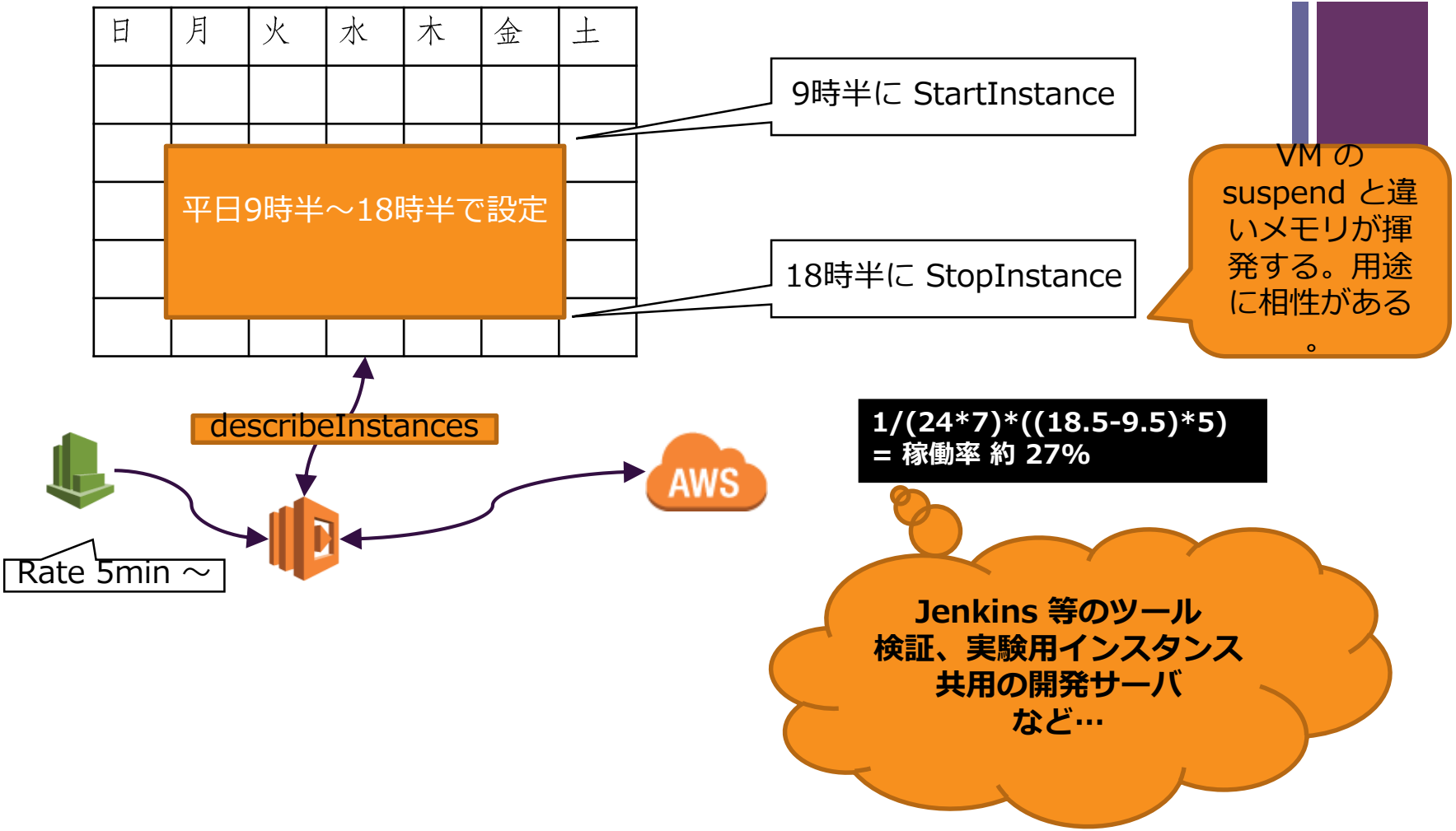
- t2.micro
- Elastic Load Balancing
- サーバ証明書
+ AWS Lambda
+ Amazon DynamoDB

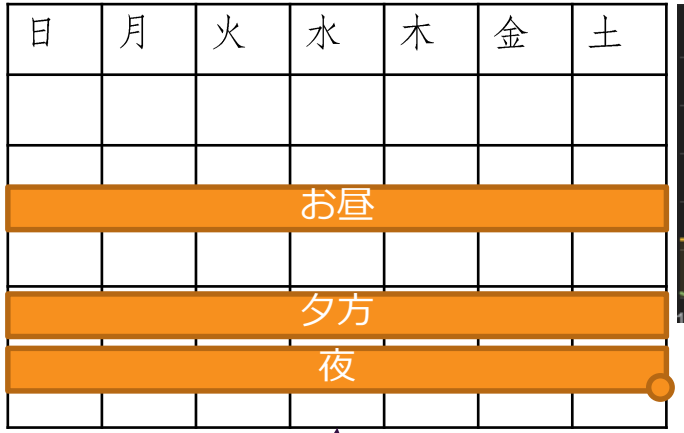
コツ

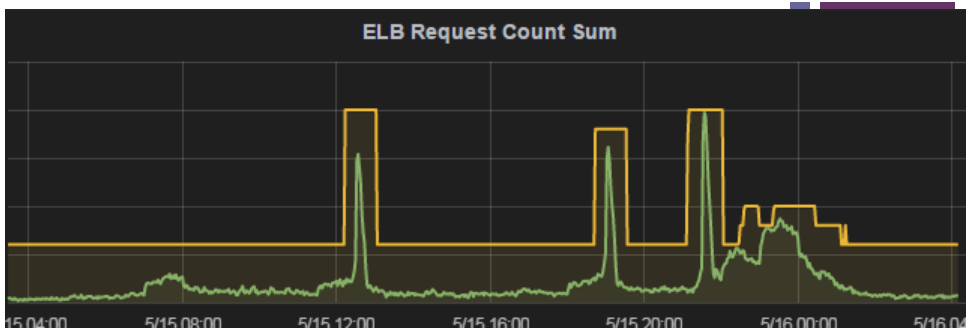
- Credentials は流出する可能性がある
  - アプリの解析、http ヘッダ解析
  - 流出前提で最低限の Policy のみ設定する
  - でもふつうの https を叩かれるよりは敷居が高いはず
- AWS Lambda 内の非同期処理は並列に行う
  - 実行時間で課金されるため、なるべく待ち時間を減らす

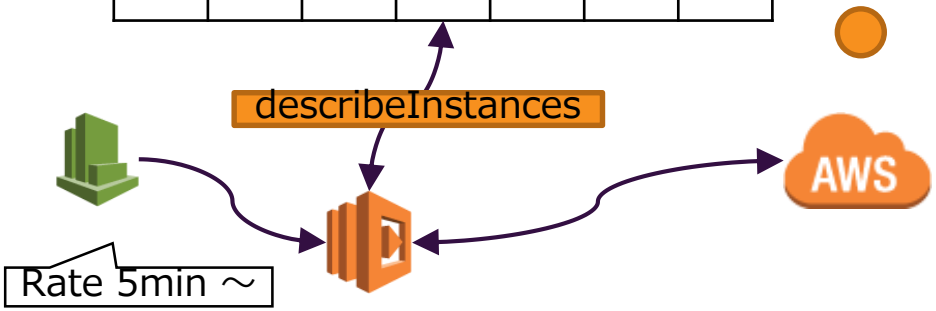
Amazon EC2 – 時間指定の動作

- 勤務時間だけ動作させるのが、当初のモチベーション
- CloudWatchEvents で個別指定ではない
  - スケジュールをかんたんに設定できるように Tags 操作
- アプリサーバにも応用
  - 毎日の予測できる急激な負荷対応
  - 予測できないもの、ゆるやかなものは Auto Scaling Group へ


日
月
火
水
木
金
土
平日9時半～18時半で設定
9時半に StartInstance
18時半に StopInstance
VM の suspend と違いメモリが揮発する。用途に相性がある。
describeInstances
AWS
Rate 5min ～
1/(24*7)*((18.5-9.5)*5)
= 稼働率 約 27%
Jenkins 等のツール
検証、実験用インスタンス
共用の開発サーバ
など…

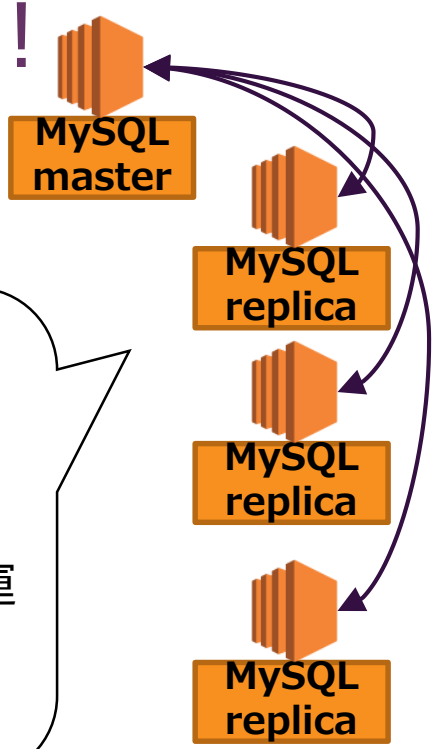






仕組みや設定が
単純なので DevOps でい
うところの
Dev におまかせ。
ゆるやかな部分は ASG
にて運用

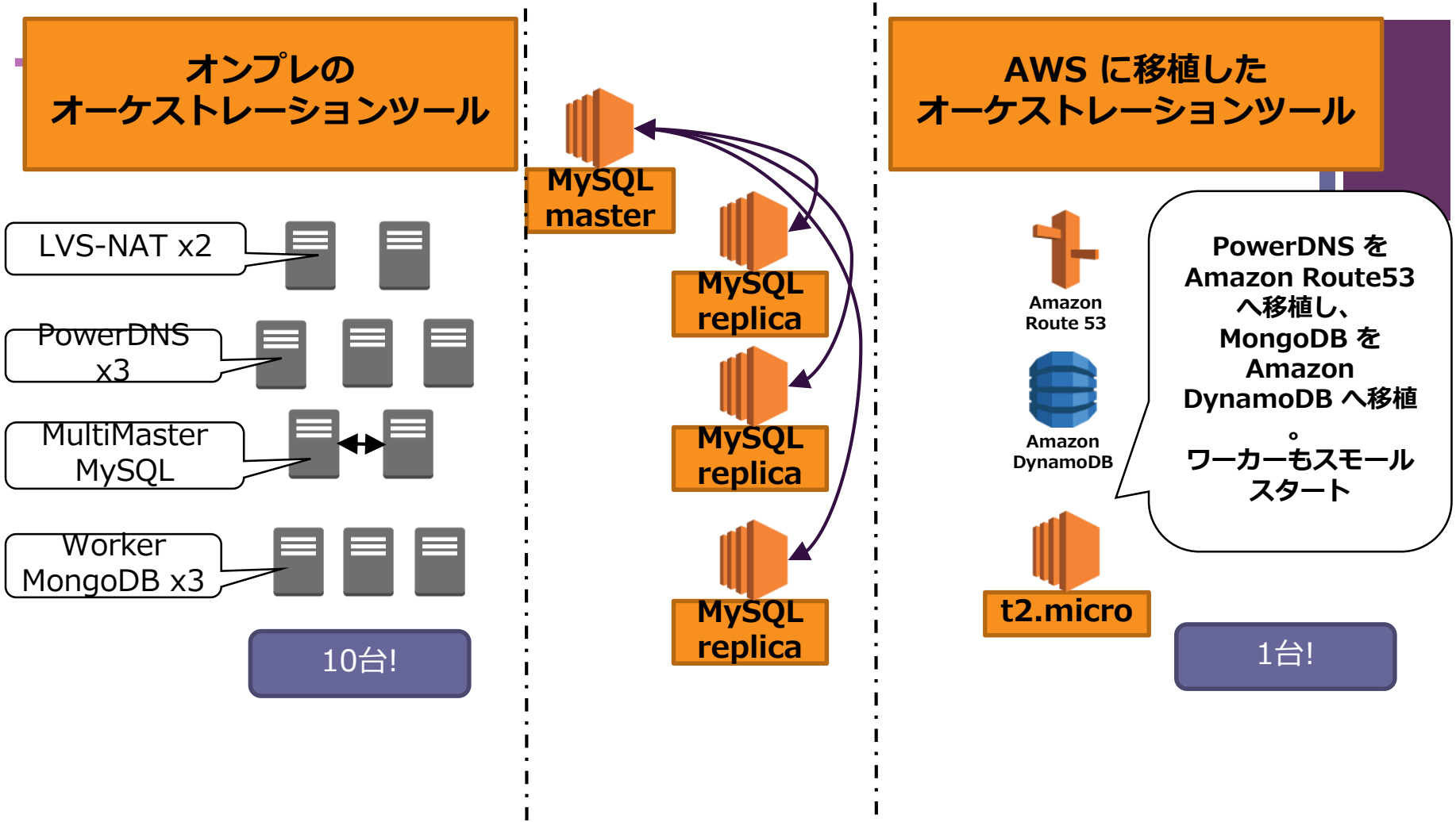
RDS より安く！



普通のレプリケーション。
しかし、安いとはいえ、
事実上構成管理ツール
が必須なのでこのままでは運
用できない。

↑
気合のある誰か
Or
自動でうごく何か
↓

費用面だけな
ら三割ぐらい
安い


オンプレの
オーケストレーションツール
LVS-NAT x2
PowerDNS x3
MultiMaster MySQL
Worker MongoDB x3
10台!
MySQL master
MySQL replica
MySQL replica
MySQL replica
AWS に移植した
オーケストレーションツール
Amazon Route 53
Amazon DynamoDB
t2.micro
PowerDNS を Amazon Route53 へ移植し、MongoDB を Amazon DynamoDB へ移植。ワーカーもスモールスタート
1台!

まとめ

- 安い順に
  1. AWS Lambda と Amazon CloudWatch Events
  2. AWS Lambda と SSM で Amazon EC2 利用
  3. AWS Lambda と API Gateway, S3 等イベントソース
  4. AWS Lambda VPC 内実行 で Amazon EC2 連携
     1. NAT や Proxy の都合がつくなら結構安いかも
  5. ちょっとしたものは Amazon DynamoDB

ありがとうございました