大規模環境における Ruby on Rails on AWS での最適化事例 ～ 200ms → 100ms への歩み ～

2018/06/01 AWS Summit 2018

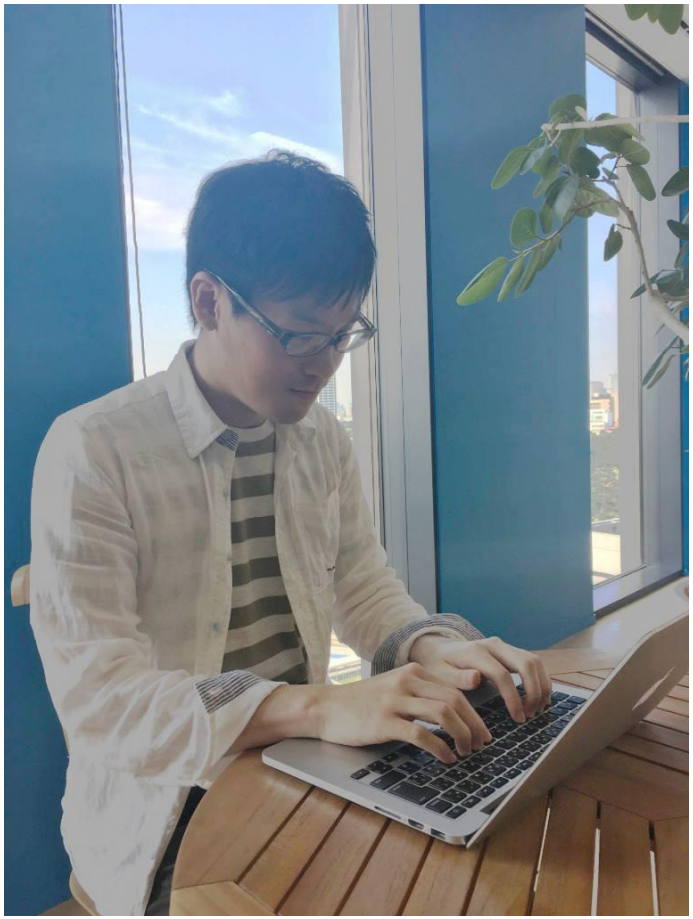
株式会社アカツキ
エンジニア
長井昭裕

自己紹介

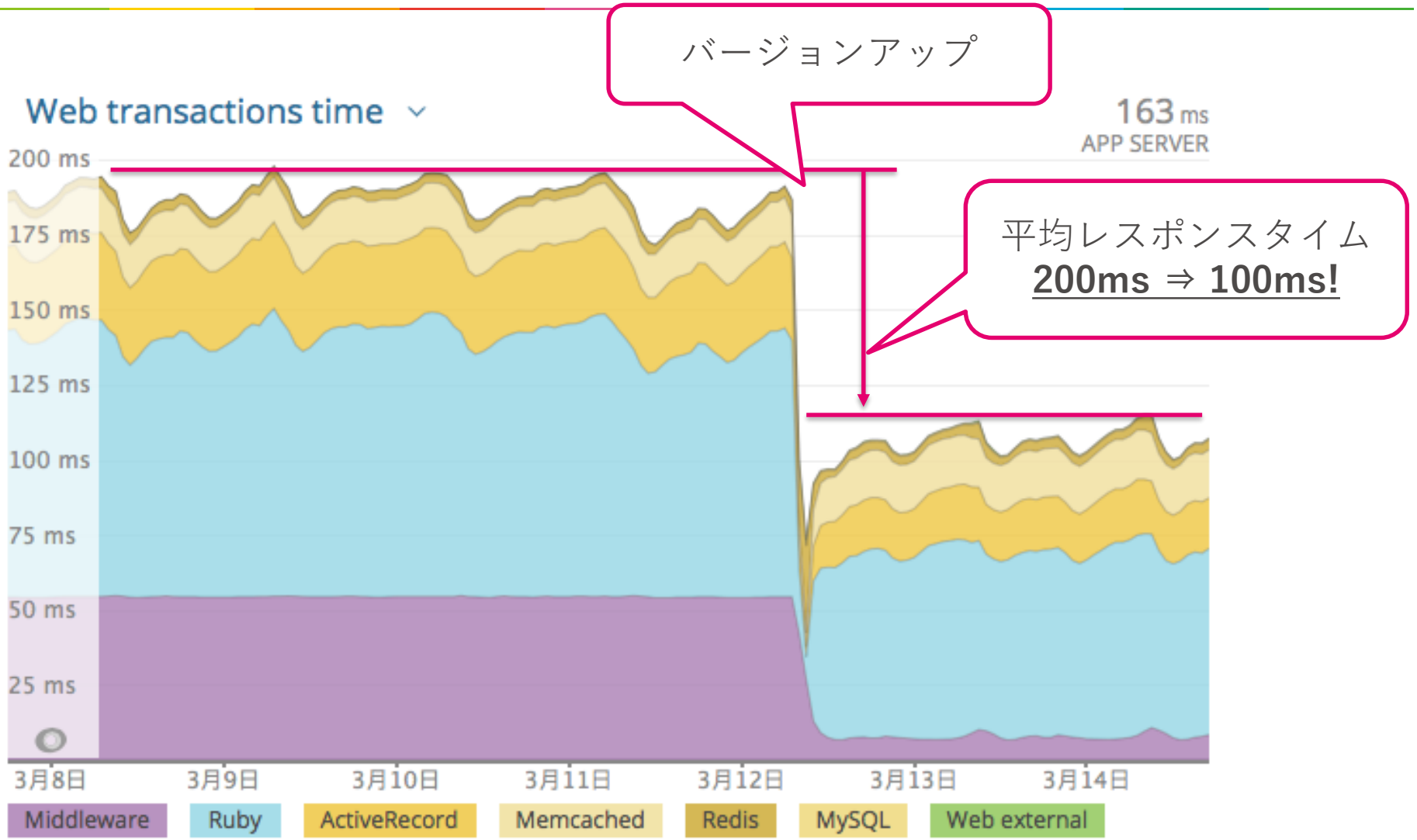
長井昭裕 (Akihiro Nagai)

経歴:
2016年にアカツキに入社
モバイルゲームの
インフラ構築・運用(AWS, GCP),
サーバサイドアプリケーション開発(Rails),
開発環境整備, 分析基盤構築,
その他諸々を担当

趣味は🍣 (2017年実績 318食/年)



とある日のバージョンアップ



モバイルゲームのサーバサイドにおいて、

いかにしてこのような高速化を実現し、

その結果どのようなメリットがあったかご紹介いたします

アジェンダ

- 高速化の意義
- モバイルゲームにおけるサーバサイドの役割
- インフラ構成
- 大規模環境下でのRuby on Rails高速化事例
- まとめ

高速化の意義

我々はなぜ高速化するのか

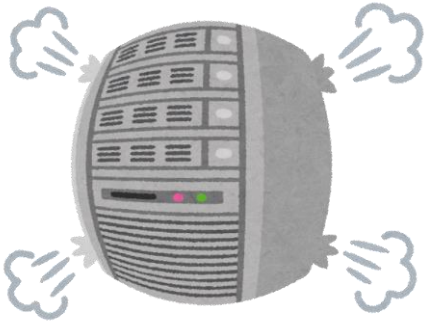
レスポンスタイムの高速化はいかなる場合も正義

UXの向上

- ストレス低減
- 定着率の改善

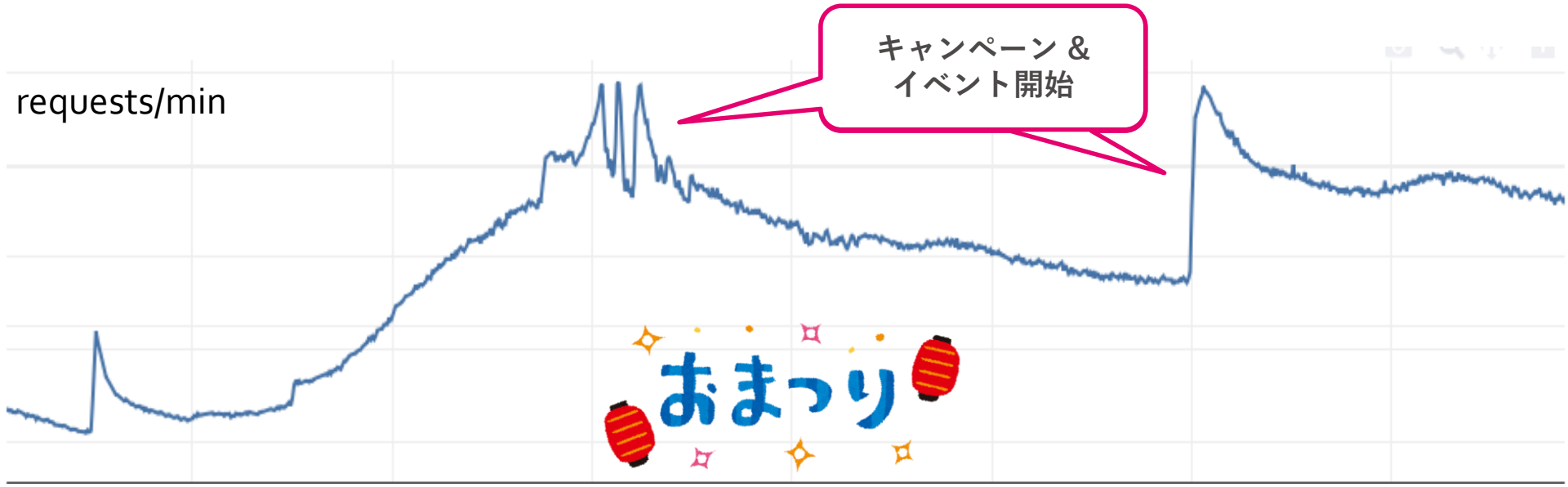
インフラの高集約化

- キャパシティの増加
- 低コスト化



高速化はユーザ・プロバイダともにメリットがある

モバイルゲーム運用中頻出のアクセススパイク



新しいイベントのオープンやキャンペーン開始とともに
ユーザが急増するといった現象は日常茶飯事

高速化によるキャパシティ担保は使命

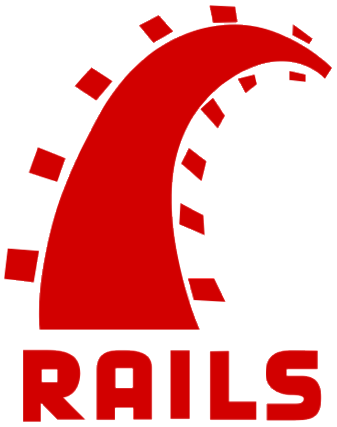
今回の高速化のターゲット

- 運用開始して数年経つモバイルゲームのサーバサイドアプリケーション
  - Webフレームワークとして Ruby on Rails を採用
  - AWS上で動作 (EC2 + RDS + Elasticache等を利用)

- 一般的な高速化はやりつくされている
  - N+1クエリの撲滅
  - DBからの大量レコード取得回避
  - DBへの適切なindex設計 ...etc

Ruby on Rails

- 非常に有名なWebフレームワーク
- 高い生産性と柔軟性を持つ
  - スモールスタートできるため
    スタートアップでの採用が多い



反面、一般に大規模環境向けではない・高速ではない
と言われている

**Ruby on Railsを用いたシステムが
AWS上で大規模にスケールアウトし、かつ高速に動作する
ことが可能であることをお伝えします**

モバイルゲームにおけるサーバサイドの役割

モバイルゲームにおけるサーバサイドの役割

APIサーバとして動作

・クライアントへのゲームデータ返却

- 現在チャレンジ可能なイベント一覧 & イベントの内容

・ゲームロジックの計算

- ログインボーナスやミッションの達成条件判定

・ユーザデータの管理

- イベントの達成状況管理

- ユーザの所持アイテム管理

…etc

ユーザが何かアクションをする度にサーバサイドとの通信が必要

レスポンスタイムの長短はUXに大きく関わる

モバイルゲームのサーバサイド高速化における技術的課題

ユーザデータが大量である

- ユーザが持つキャラクタ数百体 & キャラクタの育成状態
- イベント数百個のクリア状況 ...etc

ゲームロジックが複雑である

- このイベントは、別のイベントA,Bをクリアしていないと挑戦不可
- このミッションは特定のキャラクタを編成し、かつ特定のアイテムを使用した場合にのみ達成 ...etc

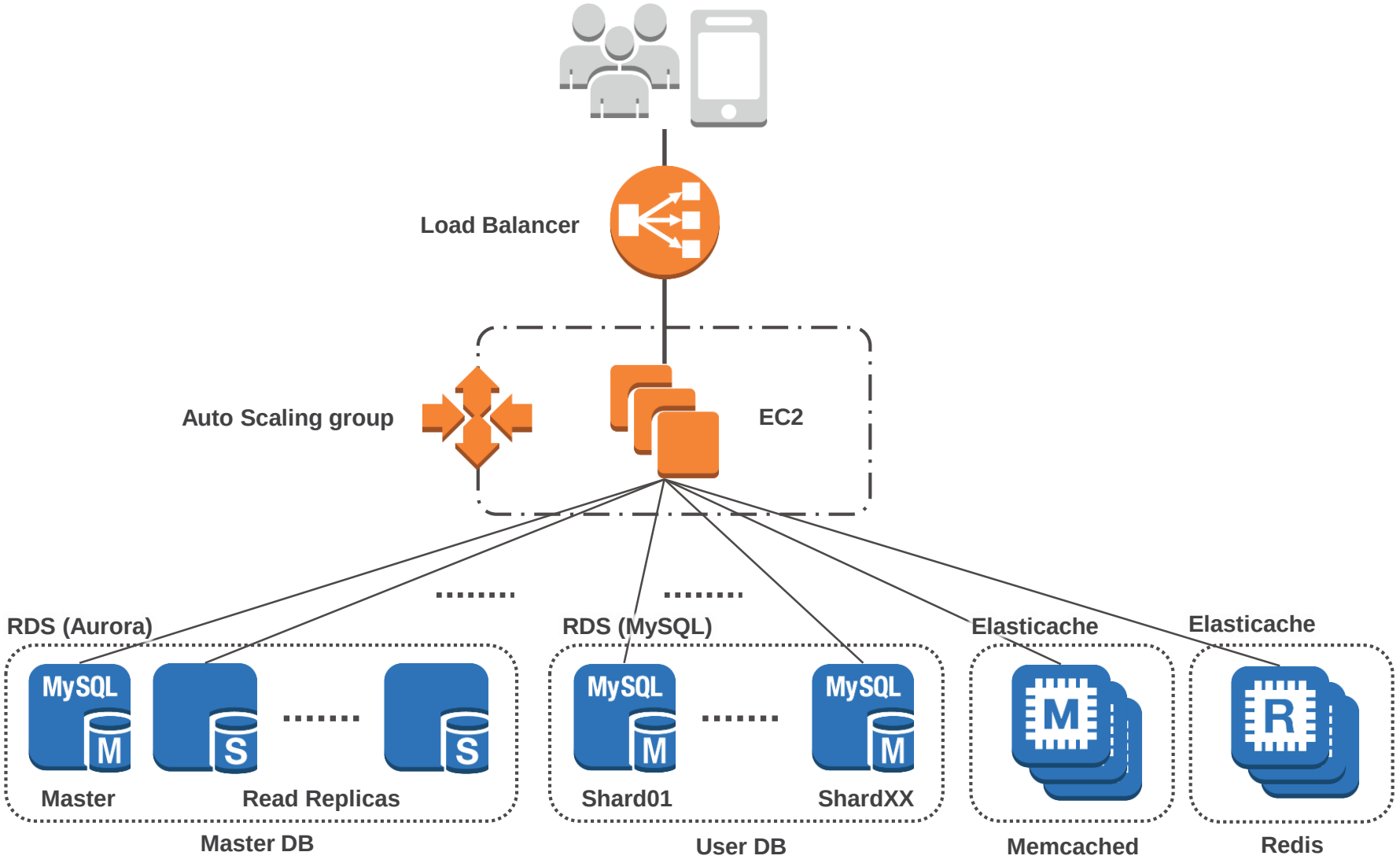
Write intensive な特性である

- アイテムの取得・経験値の獲得...etc 逐一書き込みが発生
- キャッシュを並べてスケールアウトといった戦略は採りづらい

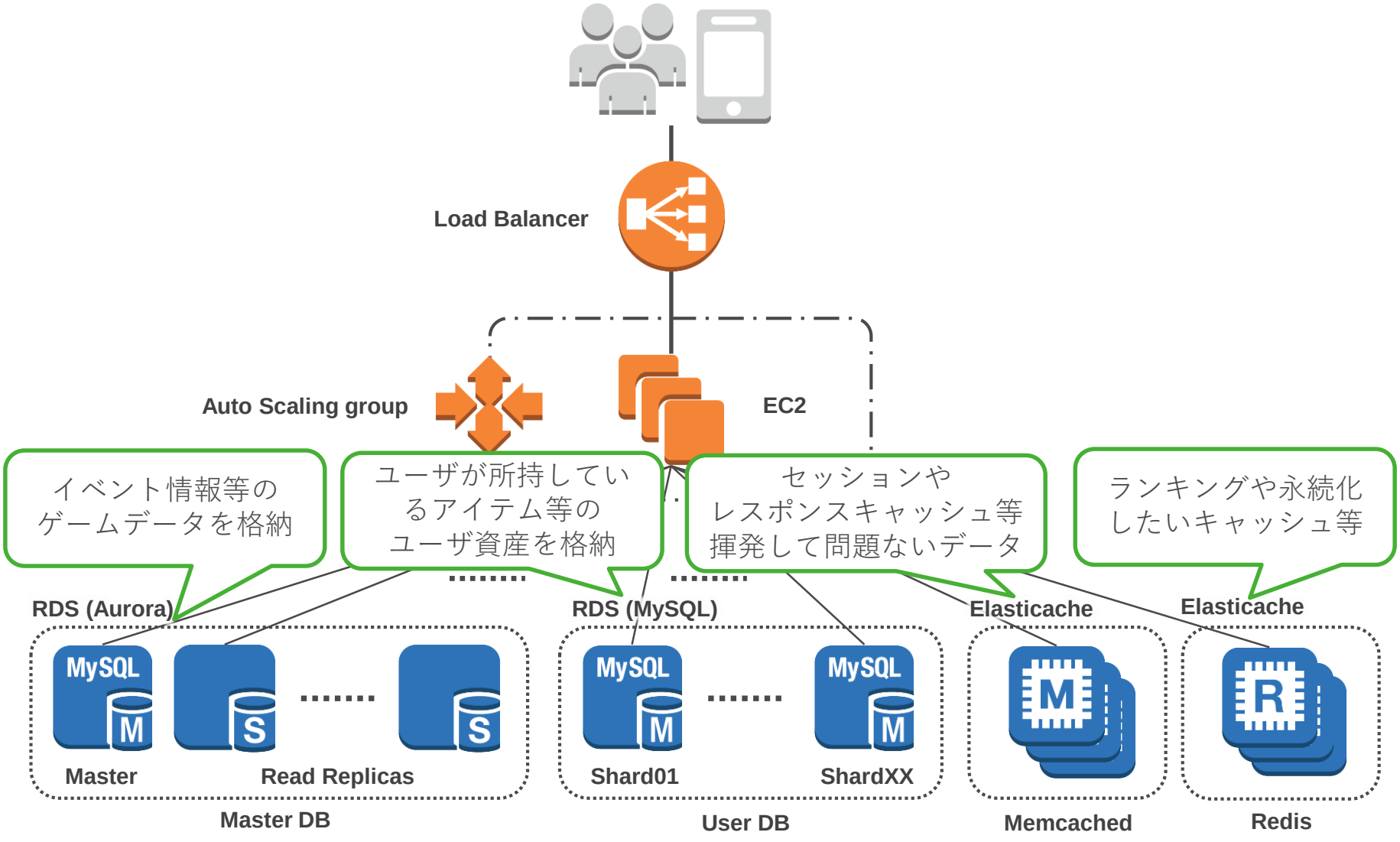
高速化は一筋縄ではいかない

インフラ構成

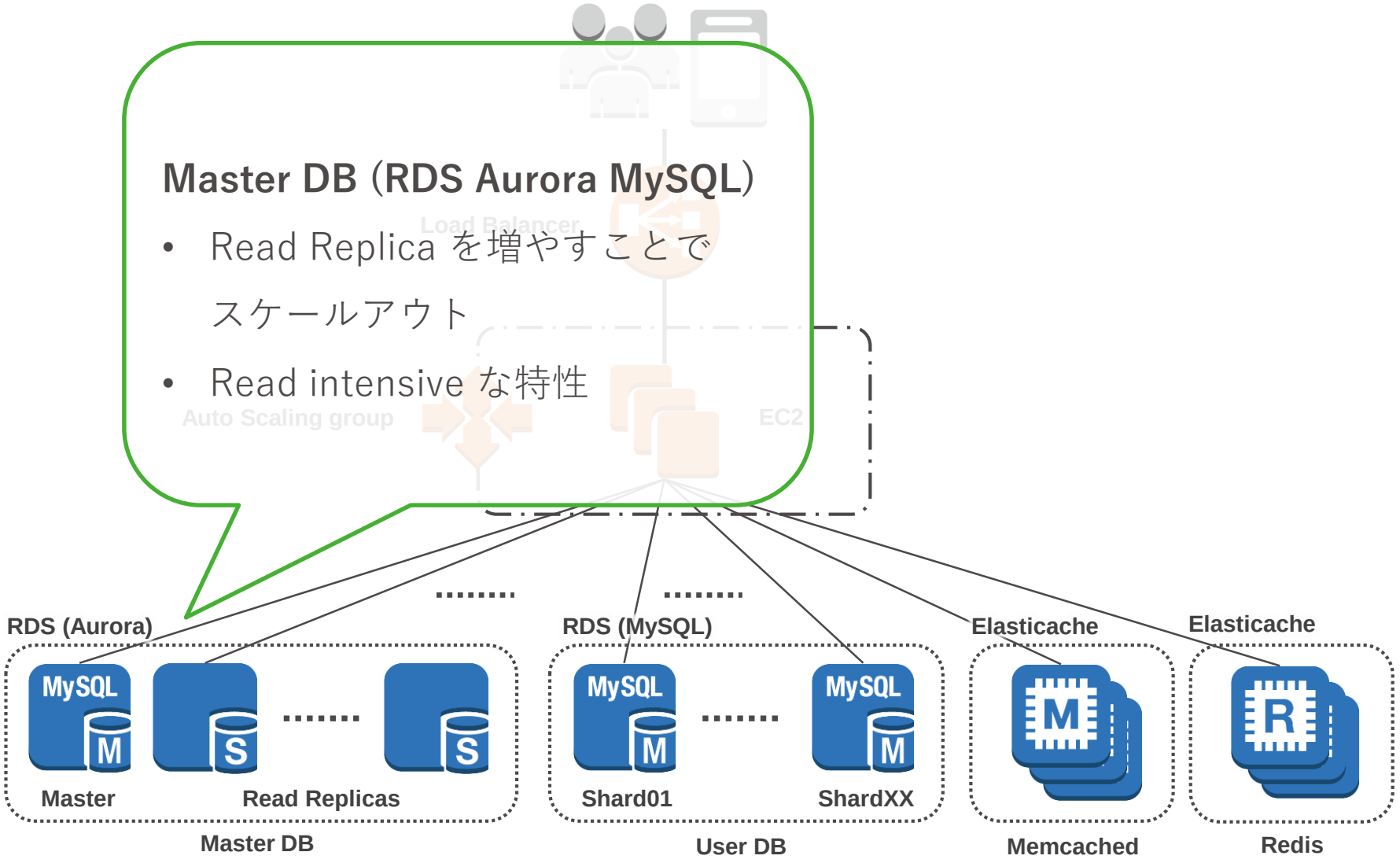
インフラ構成



インフラ構成: 各データストアの役割



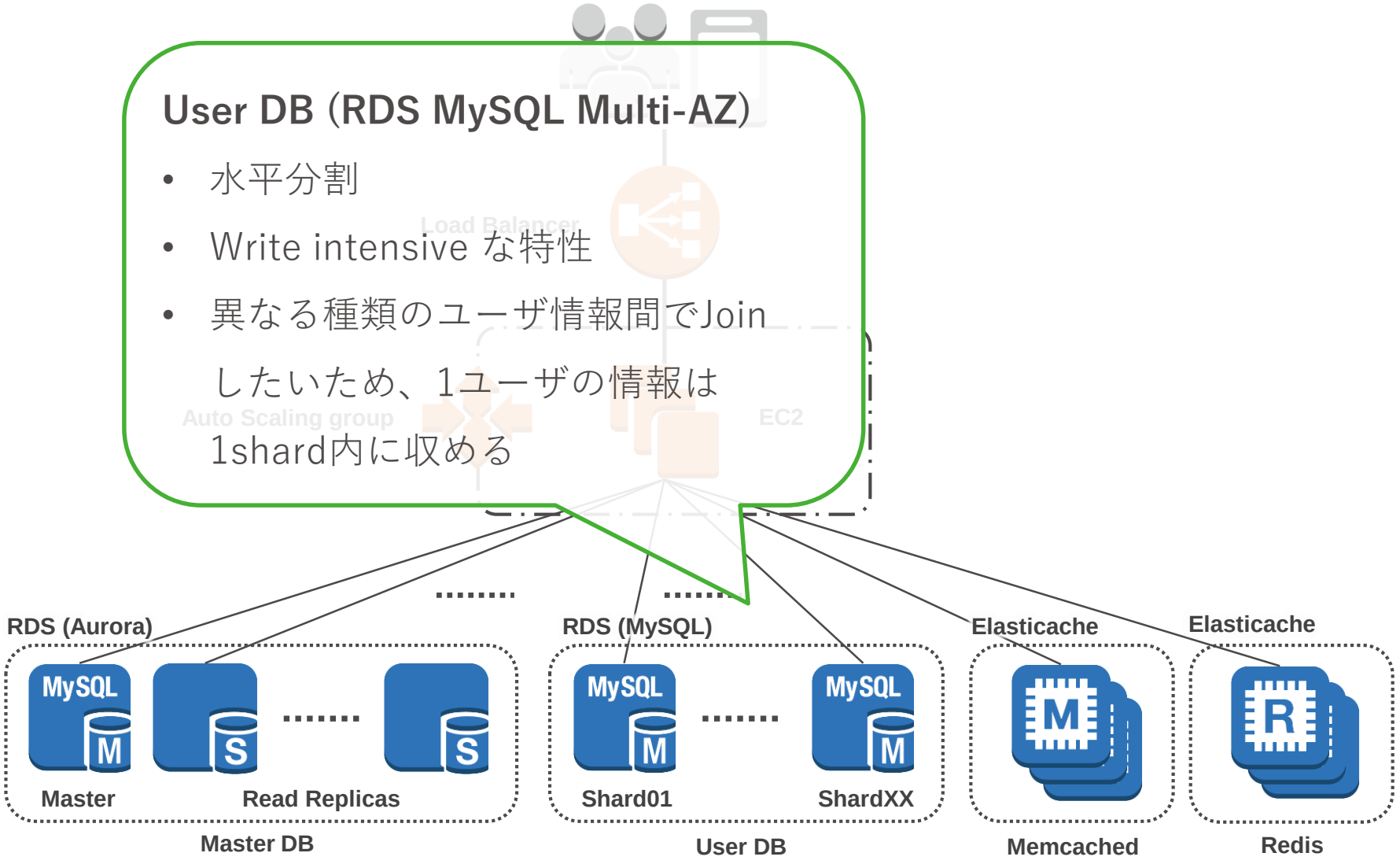
インフラ構成: スケールアウト戦略



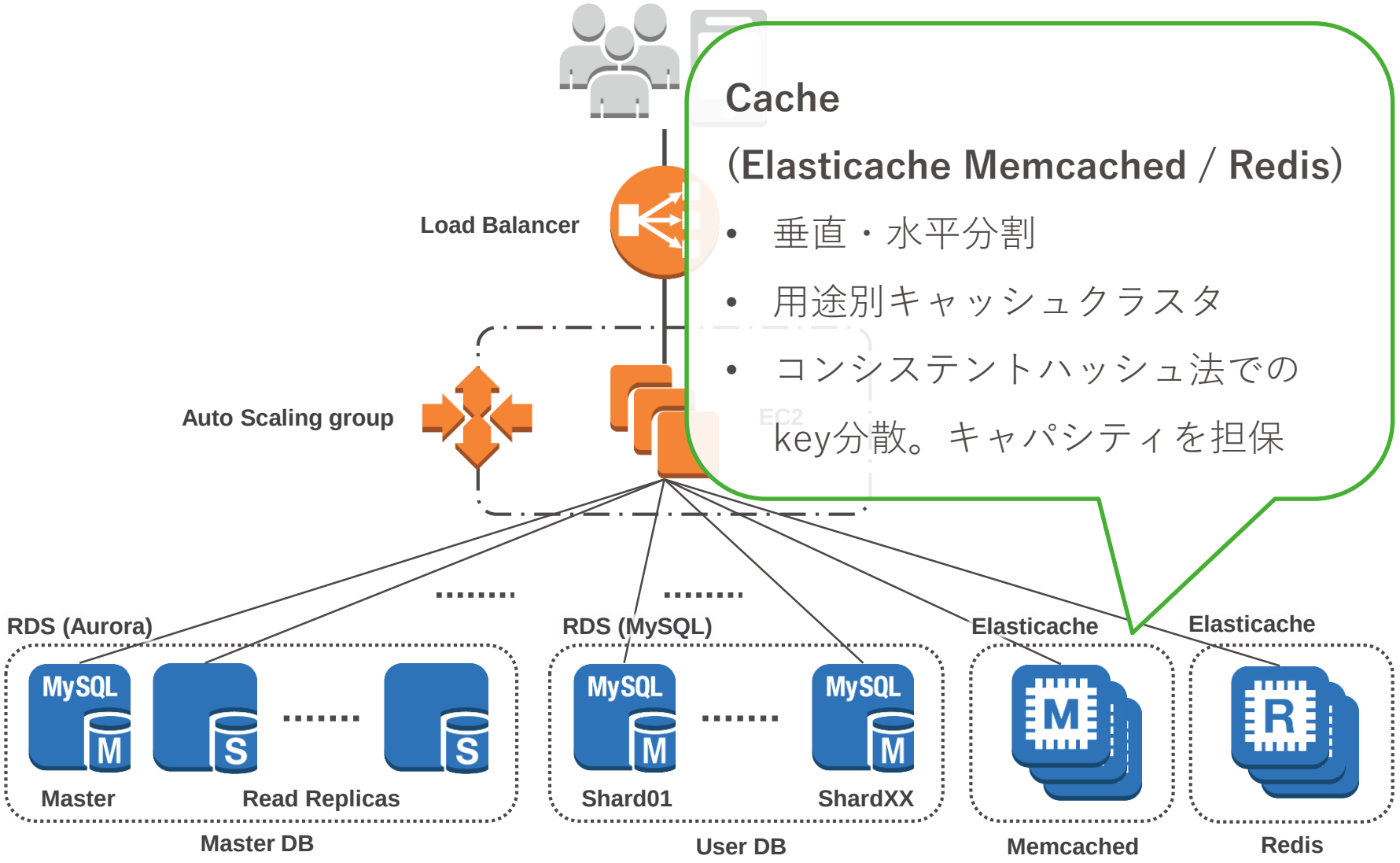
インフラ構成: スケールアウト戦略

User DB (RDS MySQL Multi-AZ)

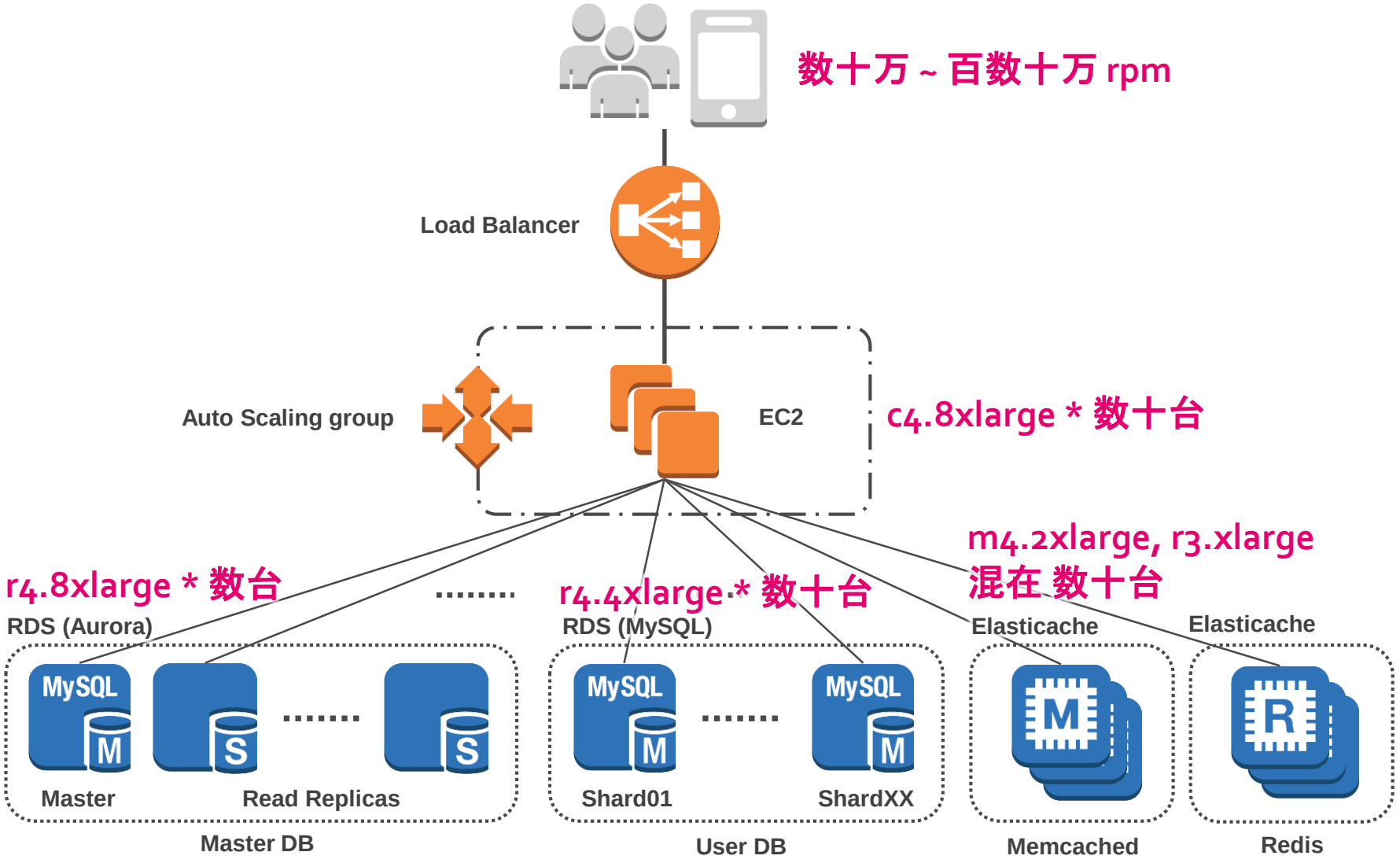
- 水平分割
- Write intensive な特性
- 異なる種類のユーザ情報間でJoin したいため、1ユーザの情報は 1shard内に収める



インフラ構成: スケールアウト戦略



インフラ構成: 規模


数十万 ~ 百数十万 rpm
Load Balancer
Auto Scaling group
EC2
c4.8xlarge * 数十台
m4.2xlarge, r3.xlarge
混在 数十台
r4.8xlarge * 数台
r4.4xlarge * 数十台
RDS (Aurora)
RDS (MySQL)
Elasticache
Elasticache
MySQL
M
S
Master
Read Replicas
Shard01
ShardXX
R
Master DB
User DB
Memcached
Redis

高速化の実践

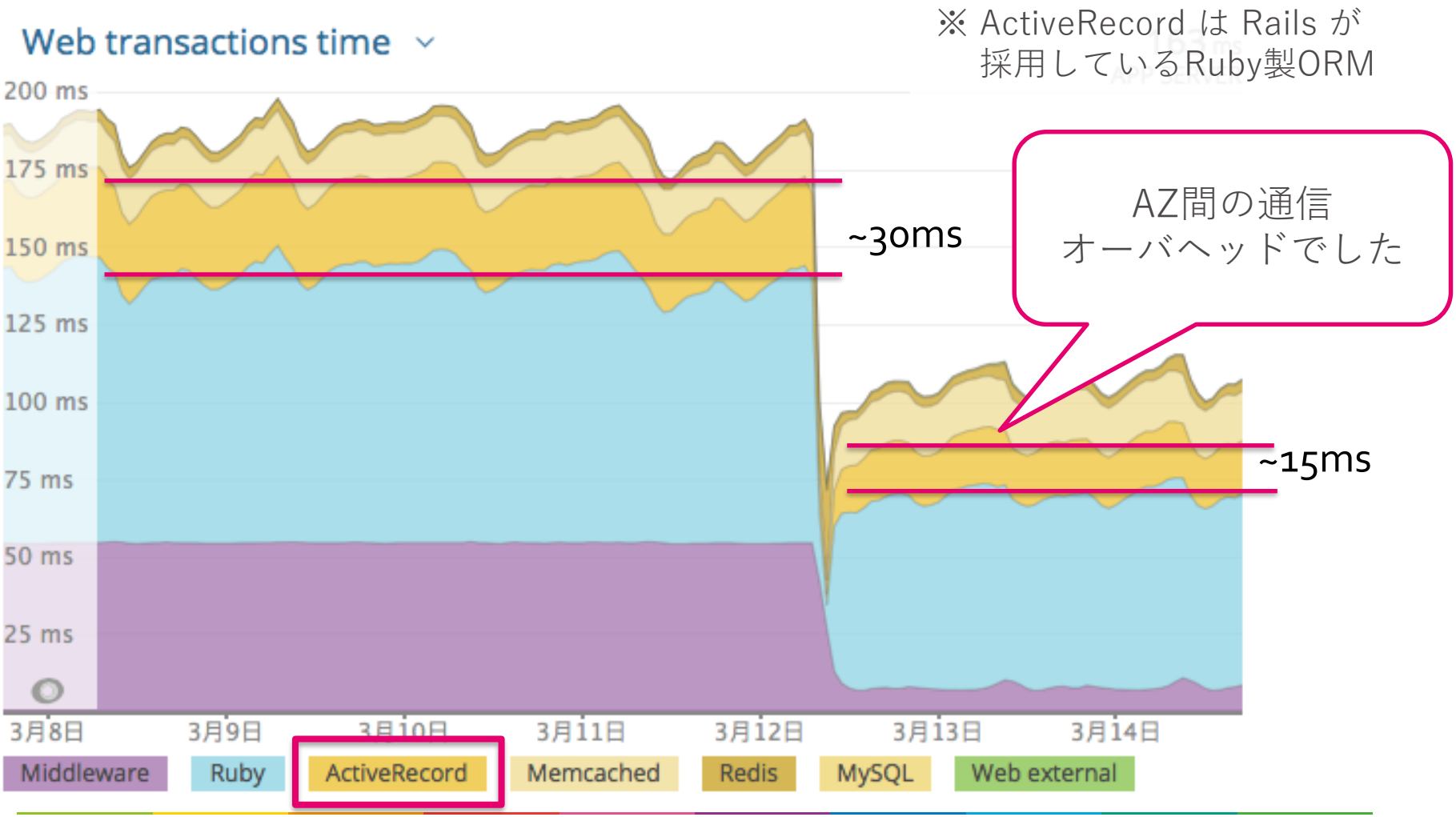
今回の高速化のターゲット(再掲)

- 運用開始して数年経つモバイルゲームのサーバサイドアプリケーション
  - Webフレームワークとして Ruby on Rails を採用
  - AWS上で動作 (EC2 + RDS + Elasticache等を利用)

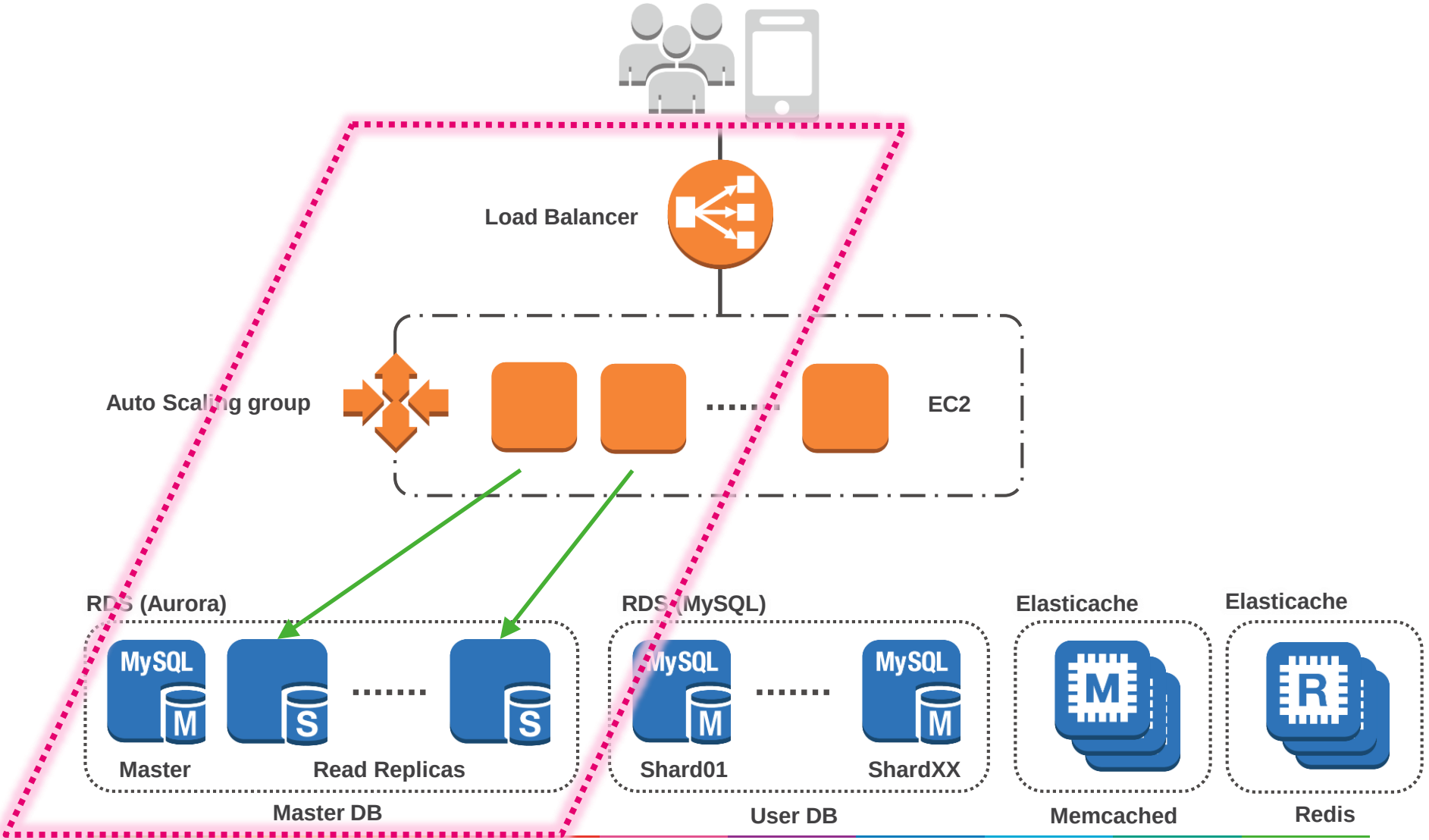
- 一般的な高速化はやりつくされている
  - N+1クエリの撲滅
  - DBからの大量レコード取得回避
  - DBへの適切なindex設計 ...etc

高速化①: ActiveRecordの実行時間短縮

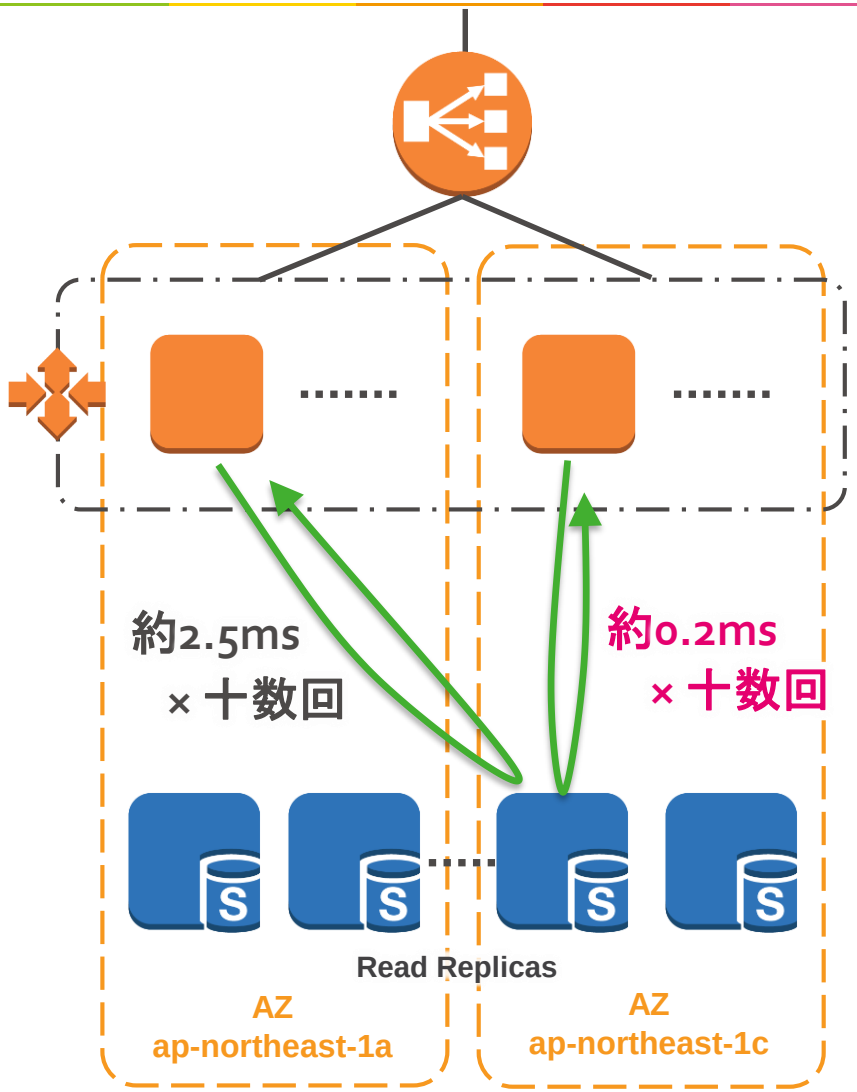
※ ActiveRecord は Rails が
採用しているRuby製ORM



EC2 – RDS 間の通信距離

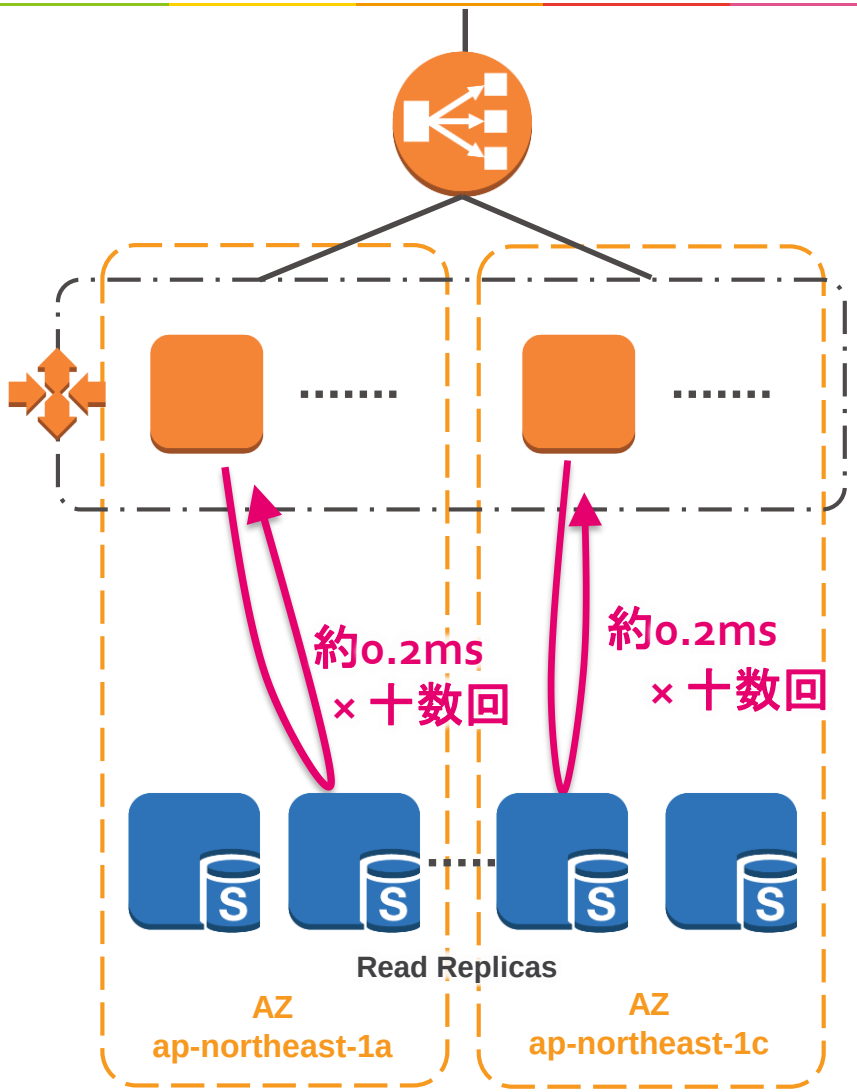


異なるAZ間の通信距離



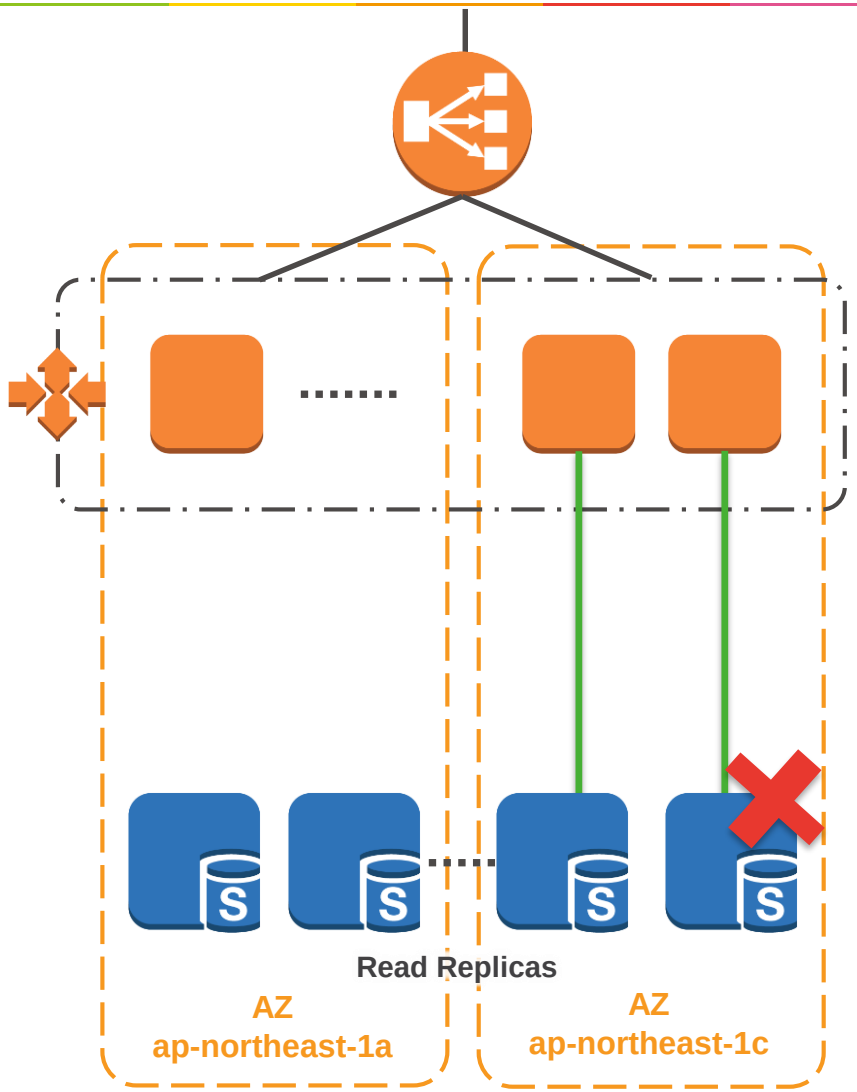
- AWSデザインパターンとして
  対障害性担保のため
  複数AZにインスタンスを配置
  することが推奨されている
- しかし、異なるAZ間のRTTは
  同一AZ内のものより数倍大きい
- ActiveRecordを使っていると
  1トランザクション内で複数の
  クエリを発行することになるため
  この差は無視できないものとなる

同一AZへの優先的なクエリ発行



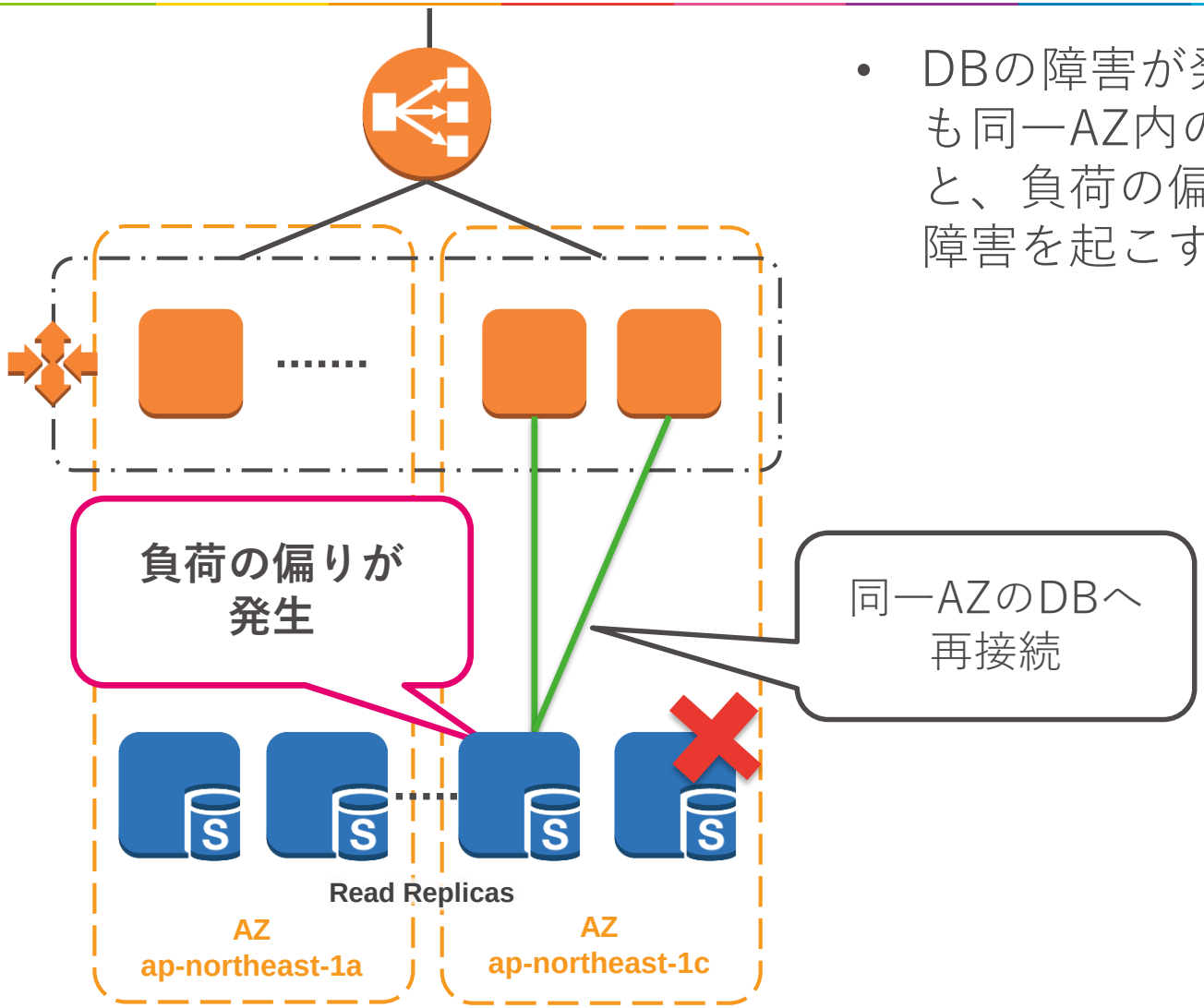
- AWSデザインパターンとして
  対障害性担保のため
  複数AZにインスタンスを配置
  することを推奨している
- しかし、異なるAZ間のRTTは
  同一AZ内のものより数倍大きい
- ActiveRecordを使っていると
  1トランザクション内で複数の
  クエリを発行することになるため
  この差は無視できないものとなる
- **可能な限り同一AZにあるDBへ
  クエリ発行を行うことで
  レスポンスタイムの高速化を実現**

同一AZへの優先的なクエリ発行(縮退時)



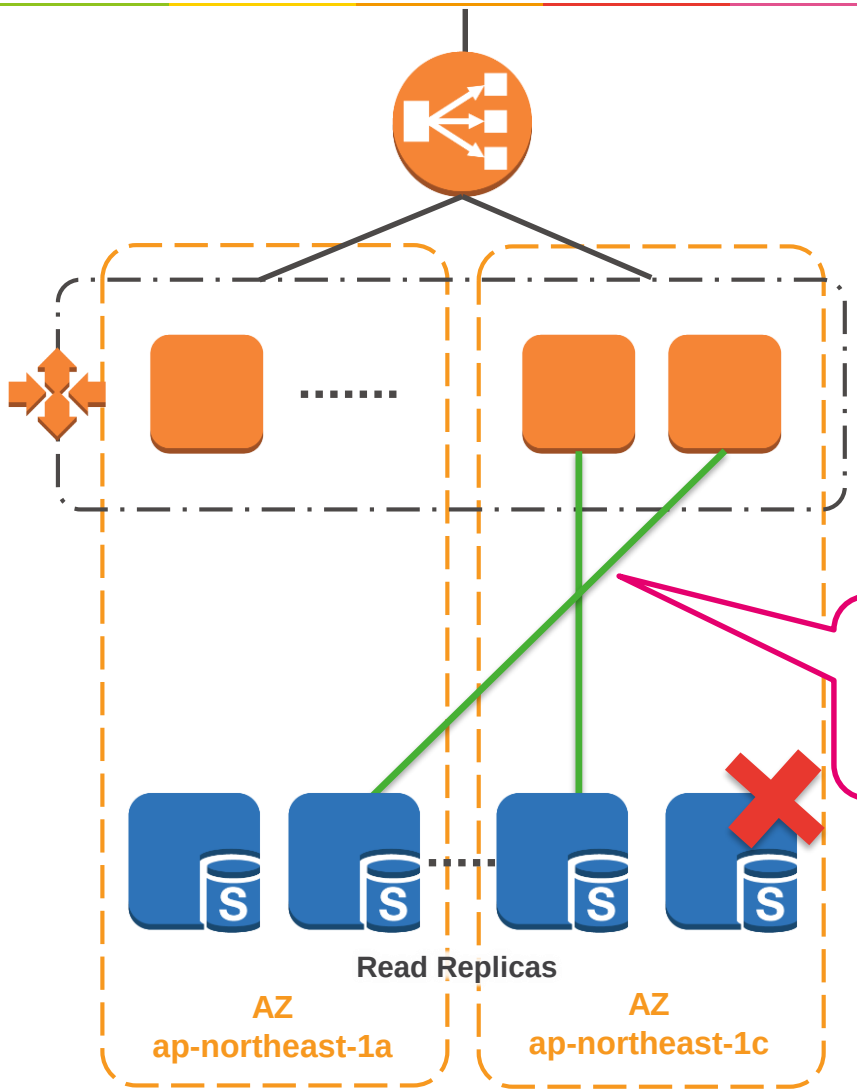
- DBの障害が発生した際の再接続先も同一AZ内のインスタンスにすると、負荷の偏りが生じ、連鎖的な障害を起こす可能性がある

同一AZへの優先的なクエリ発行(縮退時)



- DBの障害が発生した際の再接続先も同一AZ内のインスタンスにすると、負荷の偏りが生じ、連鎖的な障害を起こす可能性がある

同一AZへの優先的なクエリ発行(縮退時)

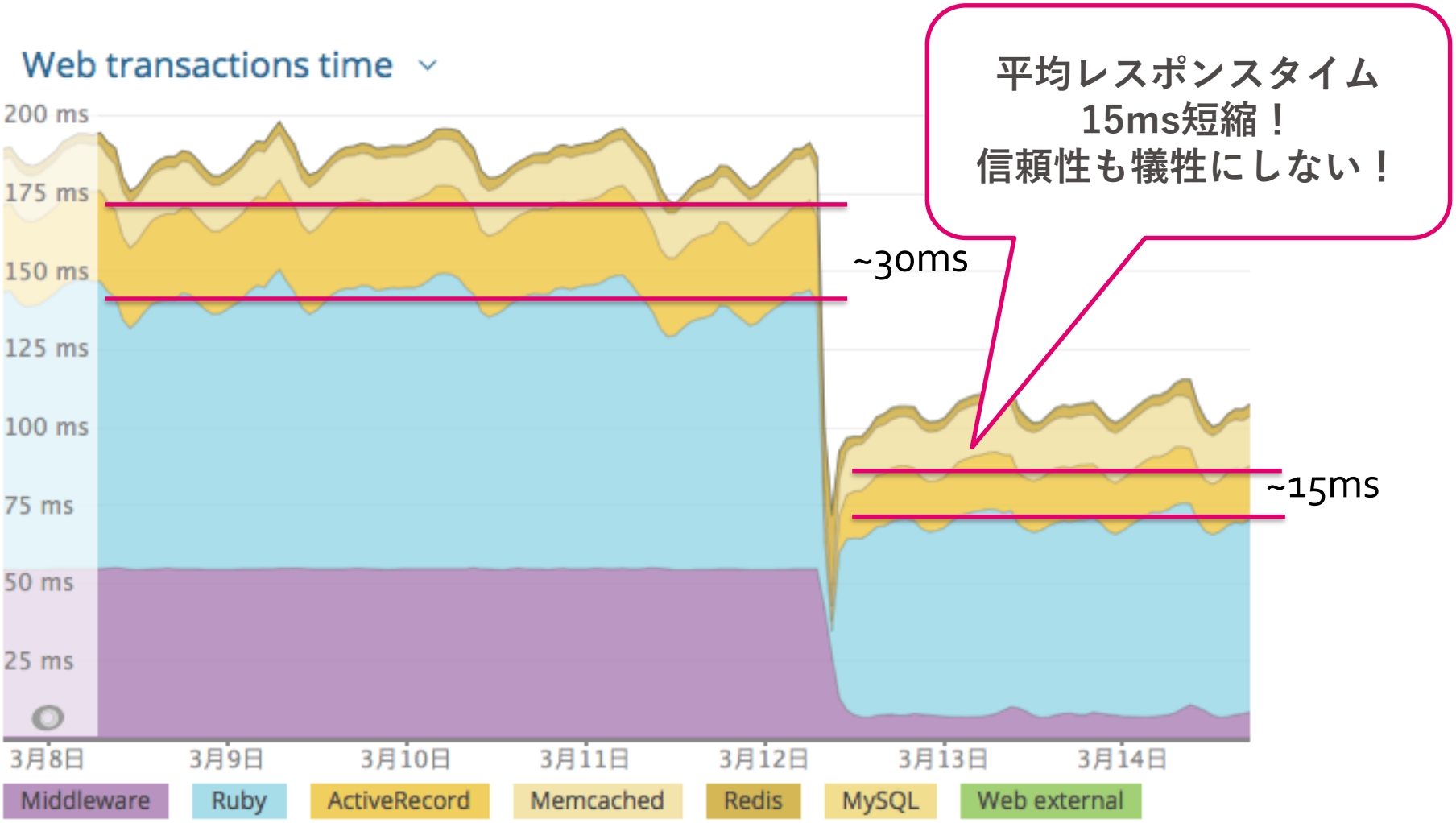


- DBの障害が発生した際の再接続先も同一AZ内のインスタンスにすると、負荷の偏りが生じ、連鎖的な障害を起こす可能性がある
- **縮退時は同一AZへの優先的な接続をやめ、ランダムでインスタンスを選ぶことで負荷分散をする**

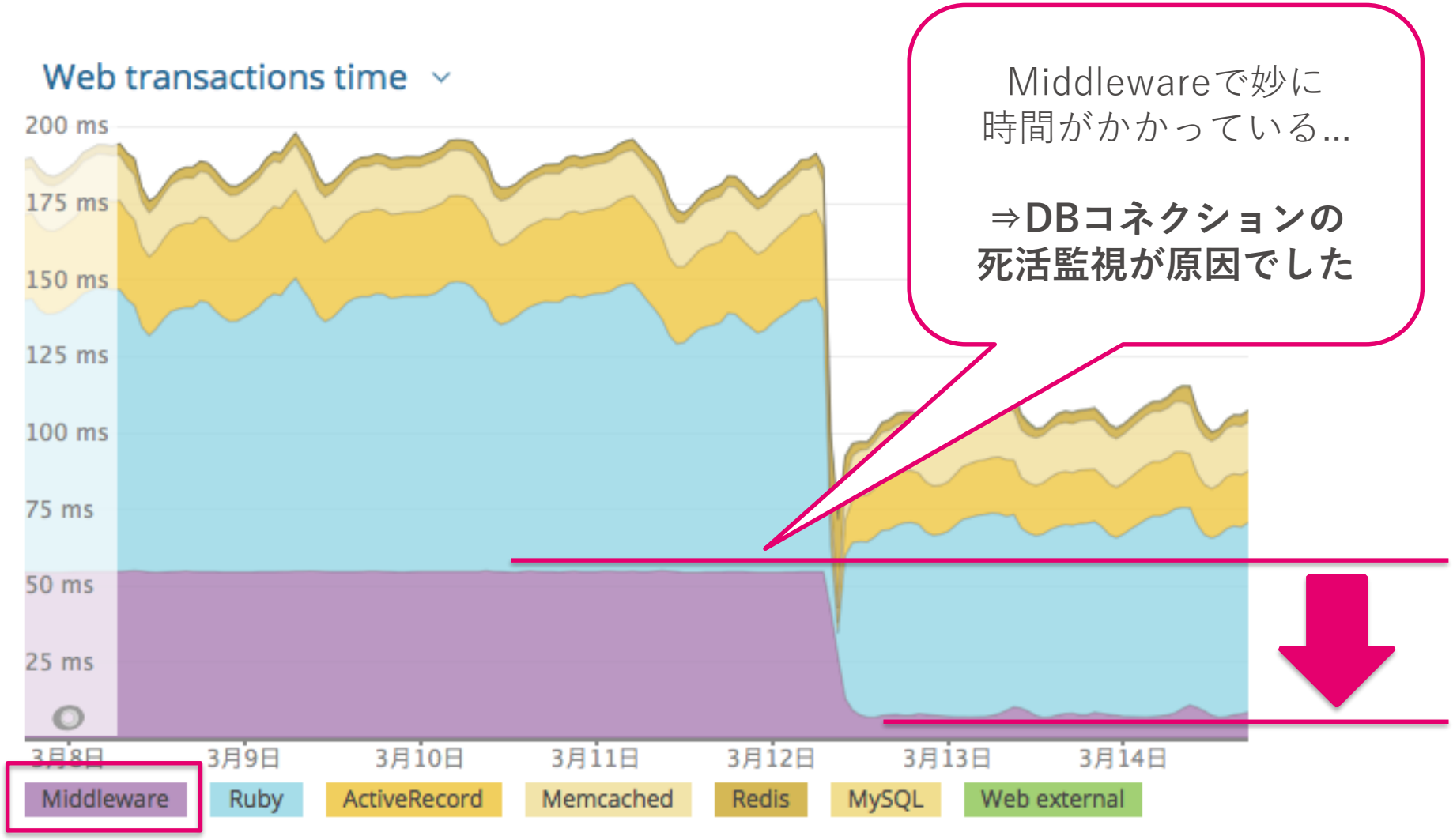
縮退時は
負荷分散を優先

高速化だけでなくシステムの信頼性を維持することも忘れてはならない

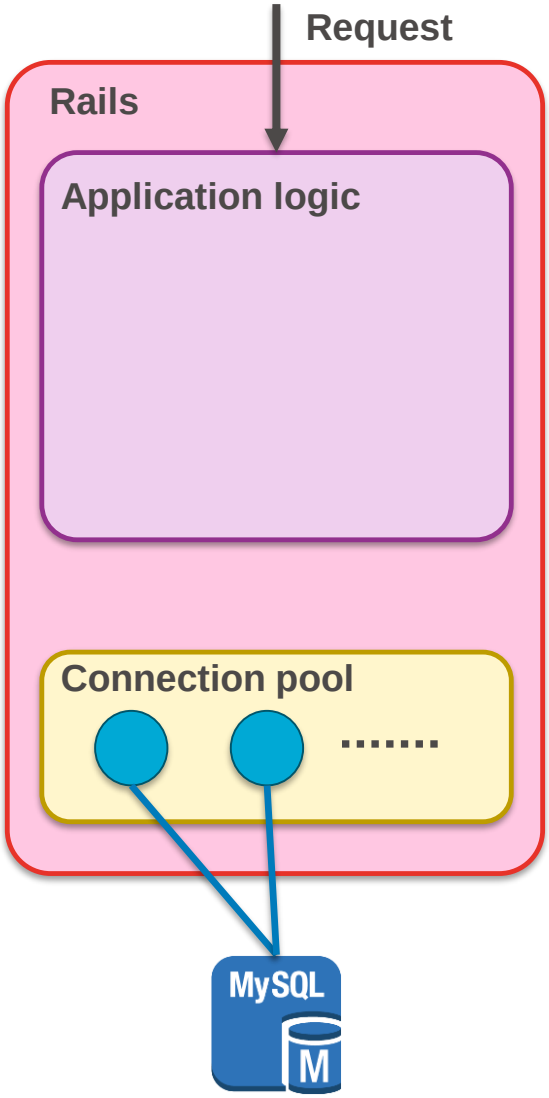
同一AZへの優先的なクエリ発行の結果



高速化②: Middleware部分の実行時間短縮

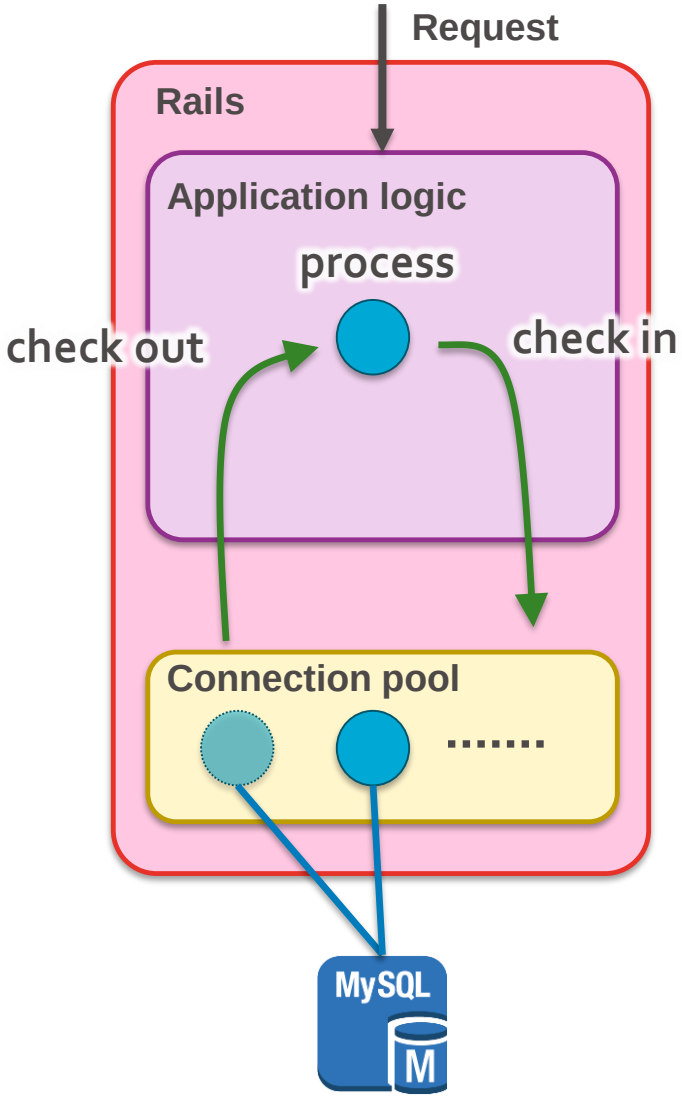


RailsにおけるDBのコネクション管理



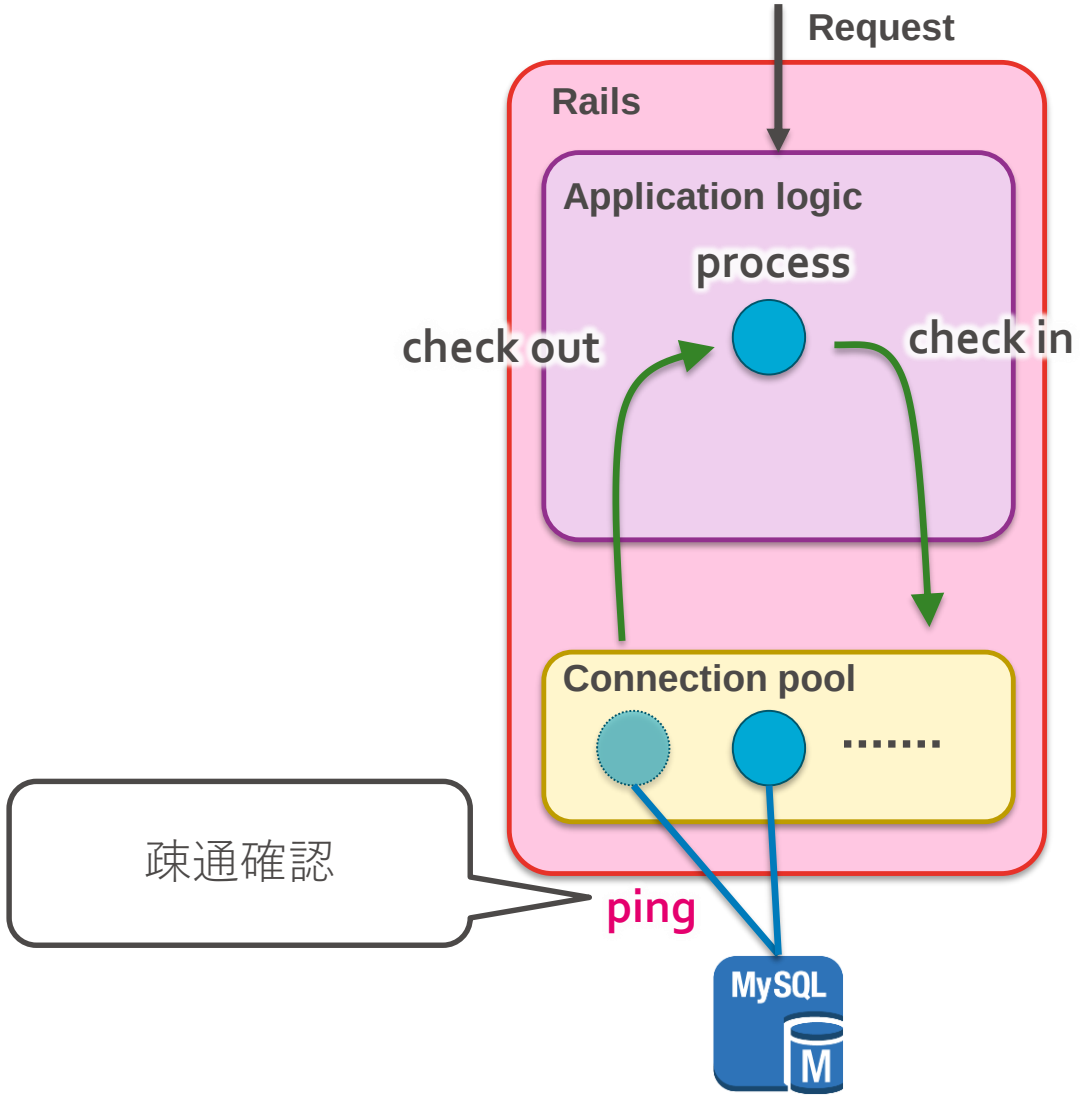
- Rails ではDBのコネクションをプーリングしており、必要に応じてプールからコネクションが返却される

RailsにおけるDBのコネクション管理



- Rails ではDBのコネクションをプーリングしており、必要に応じてプールからコネクションが返却される

RailsにおけるDBのコネクション管理

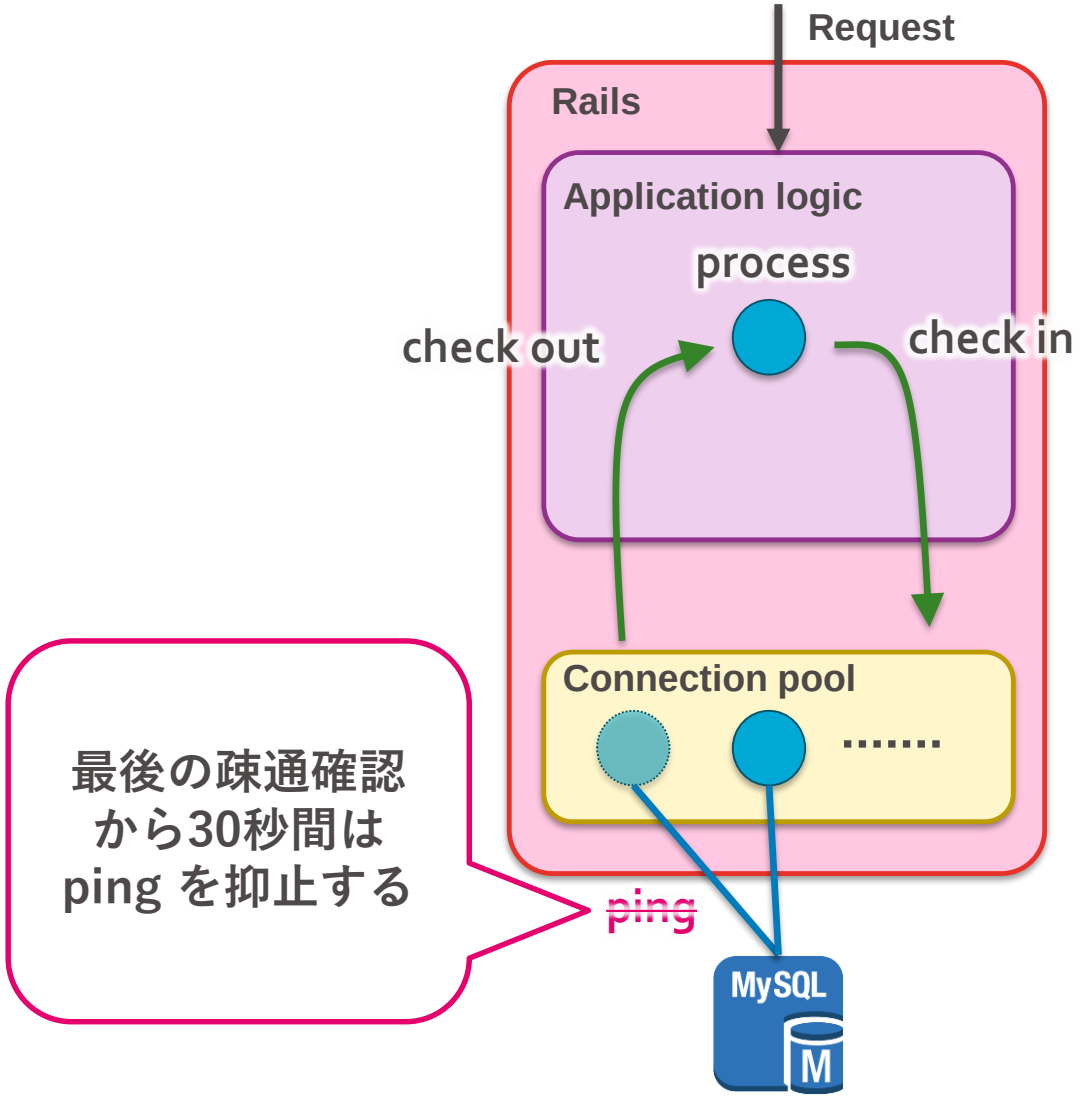


- Rails ではDBのコネクションをプーリングしており、必要に応じてプールからコネクションが返却される
- プールからは生きたコネクションを返却するため、事前に ping が疎通することを確認している
- コネクションが切れる要因
  - Failover
  - WaitTimeout

⇒どれもレアケース

⇒ **全リクエスト疎通確認する必要は無い**

RailsにおけるDBのコネクション管理



- Rails ではDBのコネクション
  をプーリングしており、
  必要に応じてプールから
  コネクションが返却される

- プールからは生きたコネク
  ションを返却するため、
  事前に ping が疎通する
  ことを確認している

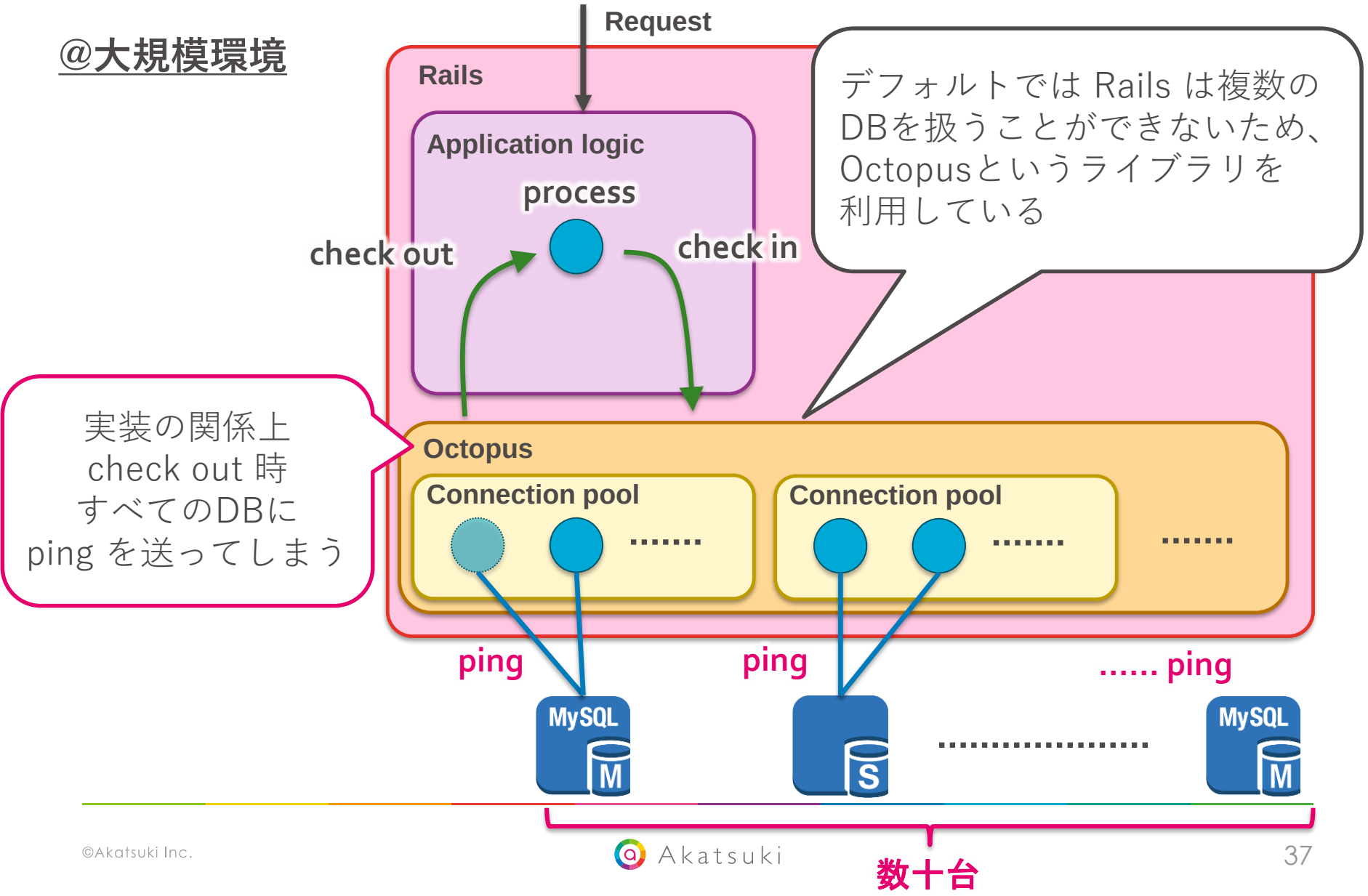
- コネクションが切れる要因
  - Failover
  - WaitTimeout

⇒どれもレアケース

⇒ **全リクエスト疎通確認する**
必要は無い

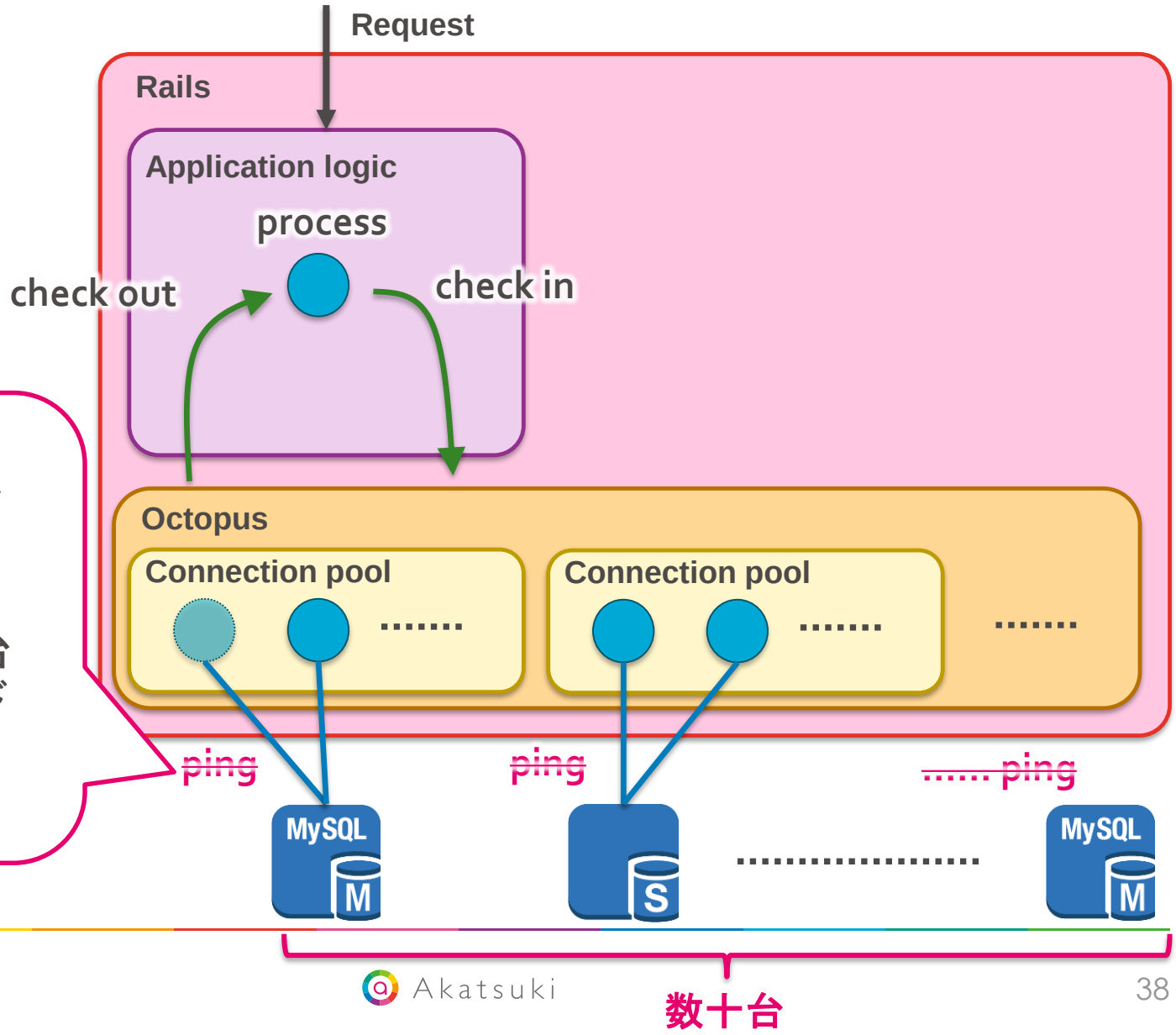
RailsにおけるDBのコネクション管理

@大規模環境



RailsにおけるDBのコネクション管理

@大規模環境

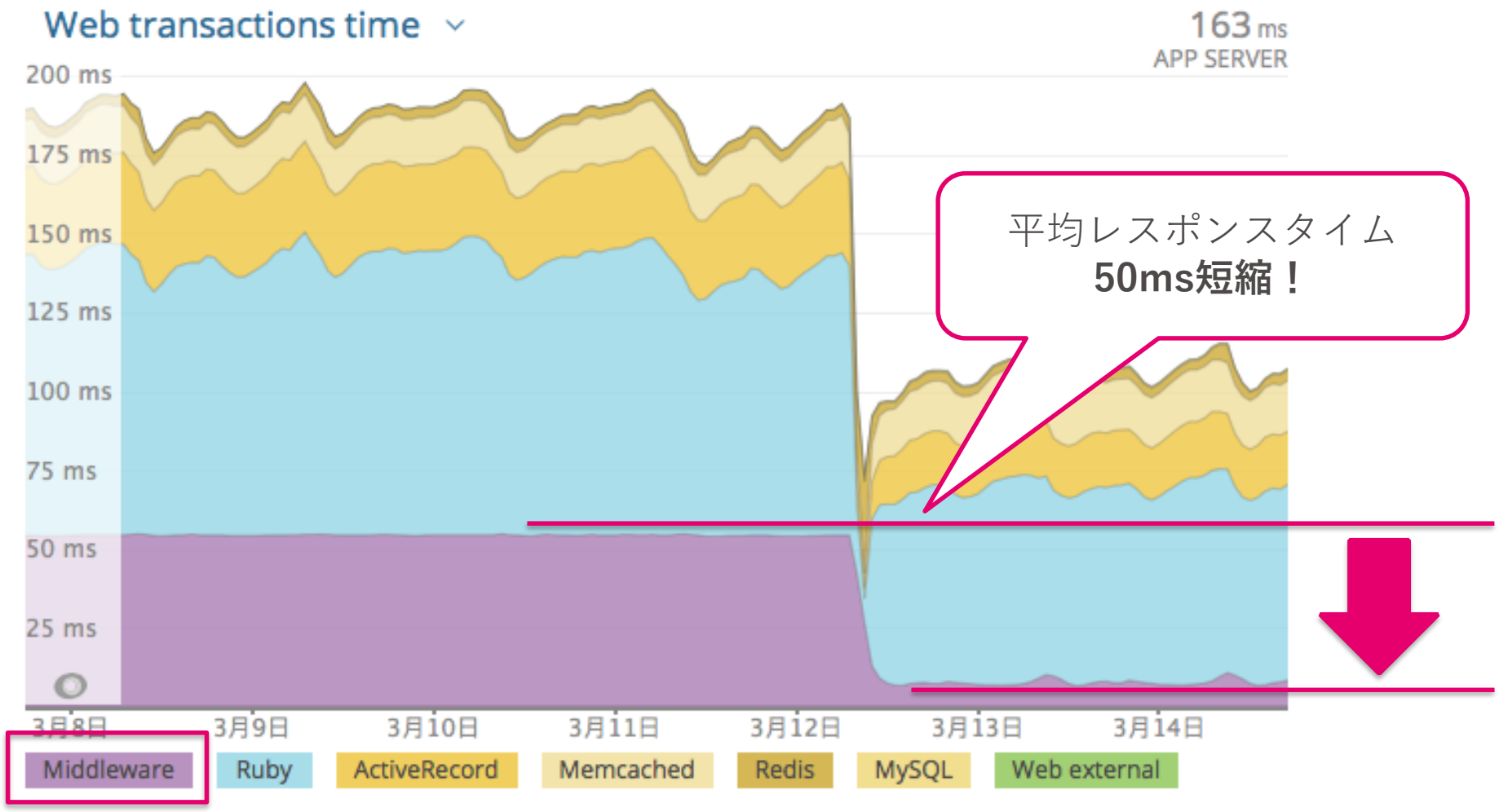


今回の対策で
不要な ping が
抑止され

数ms × 数十台のオーバヘッド

の削減に成功

DBの死活監視抑制の結果



ちなみに....

Do not ping MySQL on connection checkout #27651

Open kirs wants to merge 1 commit into `rails:master` from `kirs:kirs-ping-less`

Conversation 22 Commits 1 Checks 0 Files changed 9

kirs commented on 12 Jan 2017 Member

Now on every single connection checkout we ping MySQL with `mysql_ping`. This is nothing for an app with a couple app servers, but in case of Shopify we have over 500 servers, each running 4 containers with 12 unicorn processes. It means that on the first request after the deploy, we checkout 500 * 4 * 12 = **24000** connections. This creates a peak of 24K QPS on the master MySQL database and makes our data ops sad.

Why is this bad? Because these pings are not free. On every request we (usually) checkout two connections: 1) the shard, and 2) the master shard. This is two network roundtrips of roughly 1ms each. In addition, these calls are not free on the database either. Each call takes 100µs, whereas a small, indexed query takes 500µs. Given that we have such a staggering number of them, especially on the master shard which is checked out on connections to all shards, this adds up to an absurd amount of CPU time spent in Rails pinging connections and on the database processing those pings.

Why is the ping there? To health-check the connection and reconnect if it's not healthy. This is problematic though, because it's racy. If you perform the ping, and reconnect, by the time you hit the first query your connection might have been lost. For the majority of production, this is a no-op ping. It's legitimate in cases where the connection has gone stale (even that is rare). If we blindly remove the ping, in those cases we'd lose the entire request. This is not unreasonable, but this patch will ensure that even requests from stale or closed connections will succeed by retrying the first query after a connection checkout.

この問題は Rails upstream でも議論されていて、SpotifyやGrouponでも課題になっているようです https://github.com/rails/rails/pull/27651

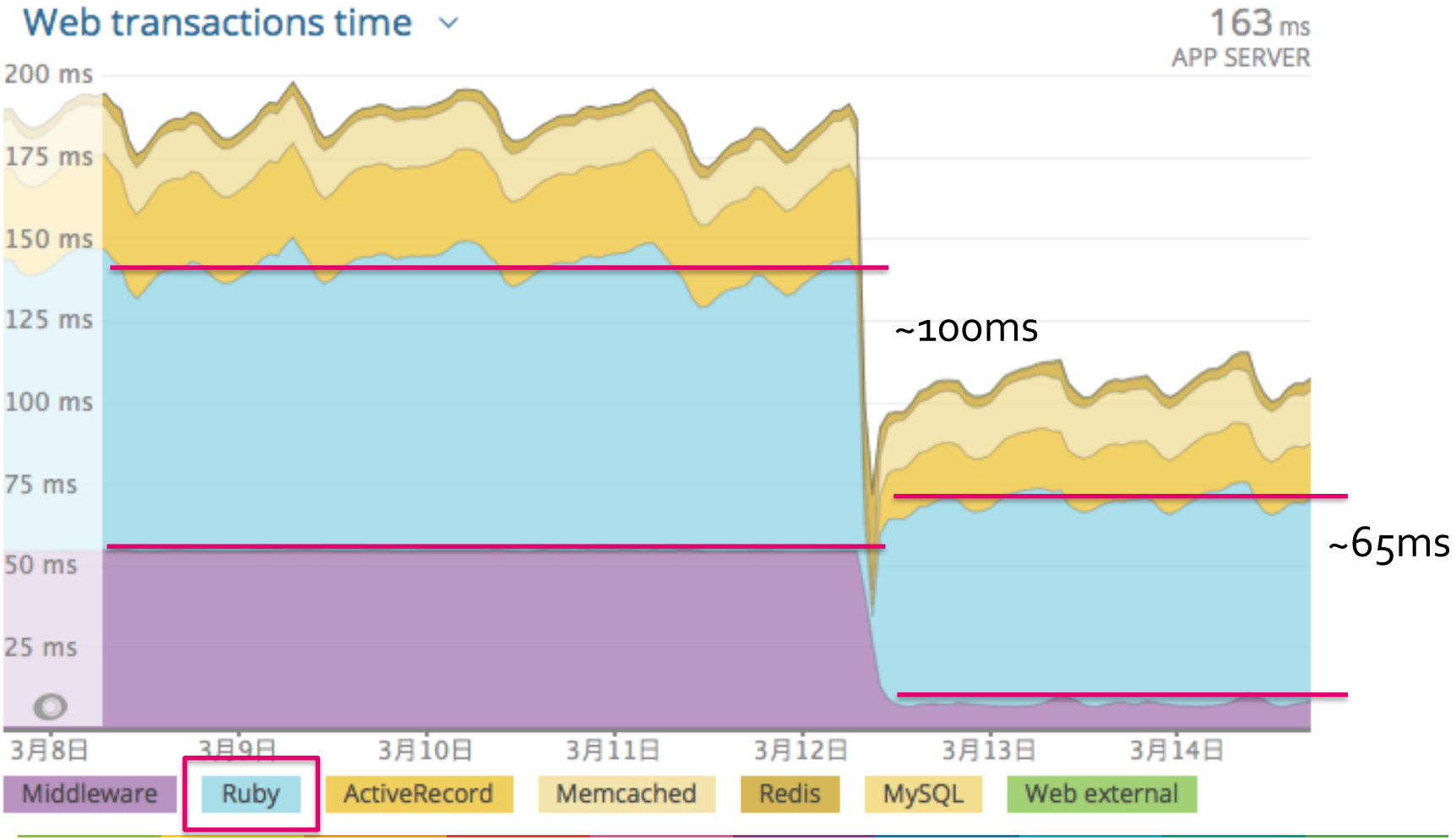
ここまでの高速化の効果

変更内容	効果
同一AZへの優先的なクエリ発行	15ms
DBへの過剰な死活監視抑止	50ms
計	**65ms**

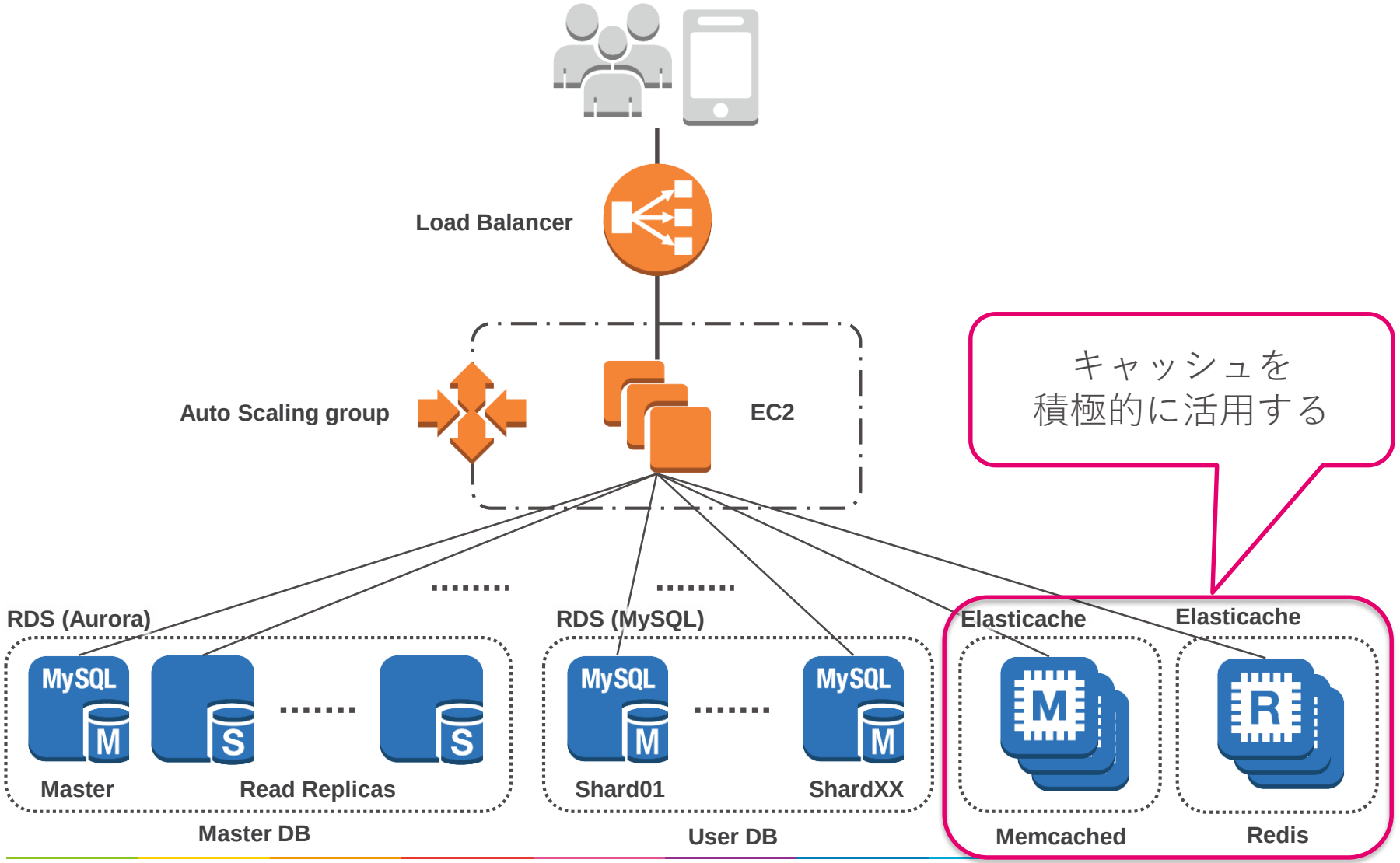
100ms削減までの残りは…

⇒**地道なアプリケーションの最適化**

アプリケーション最適化



アプリケーションの最適化



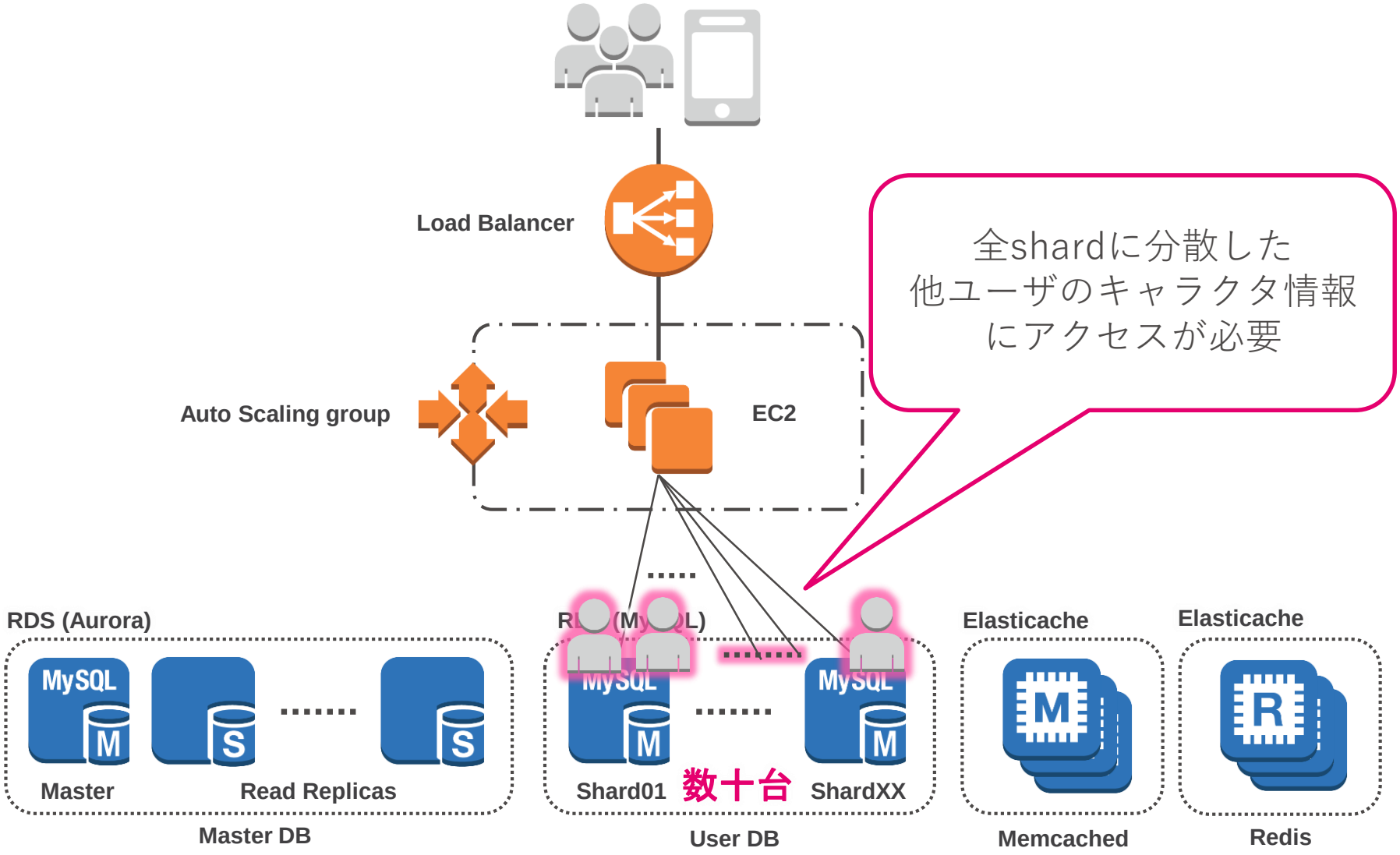
高速化事例: フレンド機能の高速化

- フレンド機能とは
  - 他ユーザの一部キャラクタを自分のキャラクタのようにイベントに連れ出せる機能
  - モバイルゲームではよく見られる

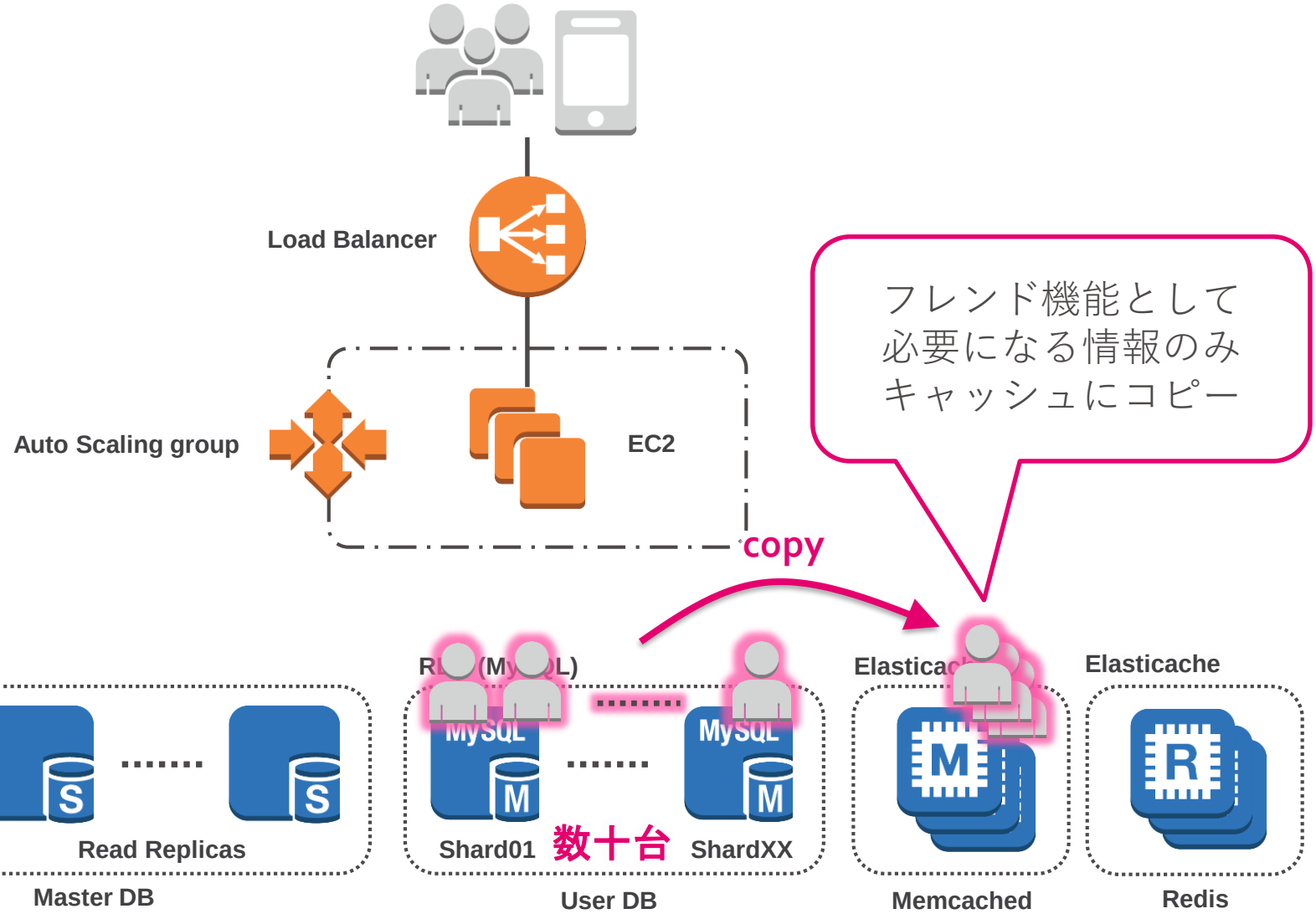
- 技術的な課題
  - キャラクタデータを保持しているUserDBは水平分割されている
  - キャラクタ1体あたりのデータ量が多い
  - 候補としてリストアップするフレンドが多い

⇒ **キャッシュを活用**

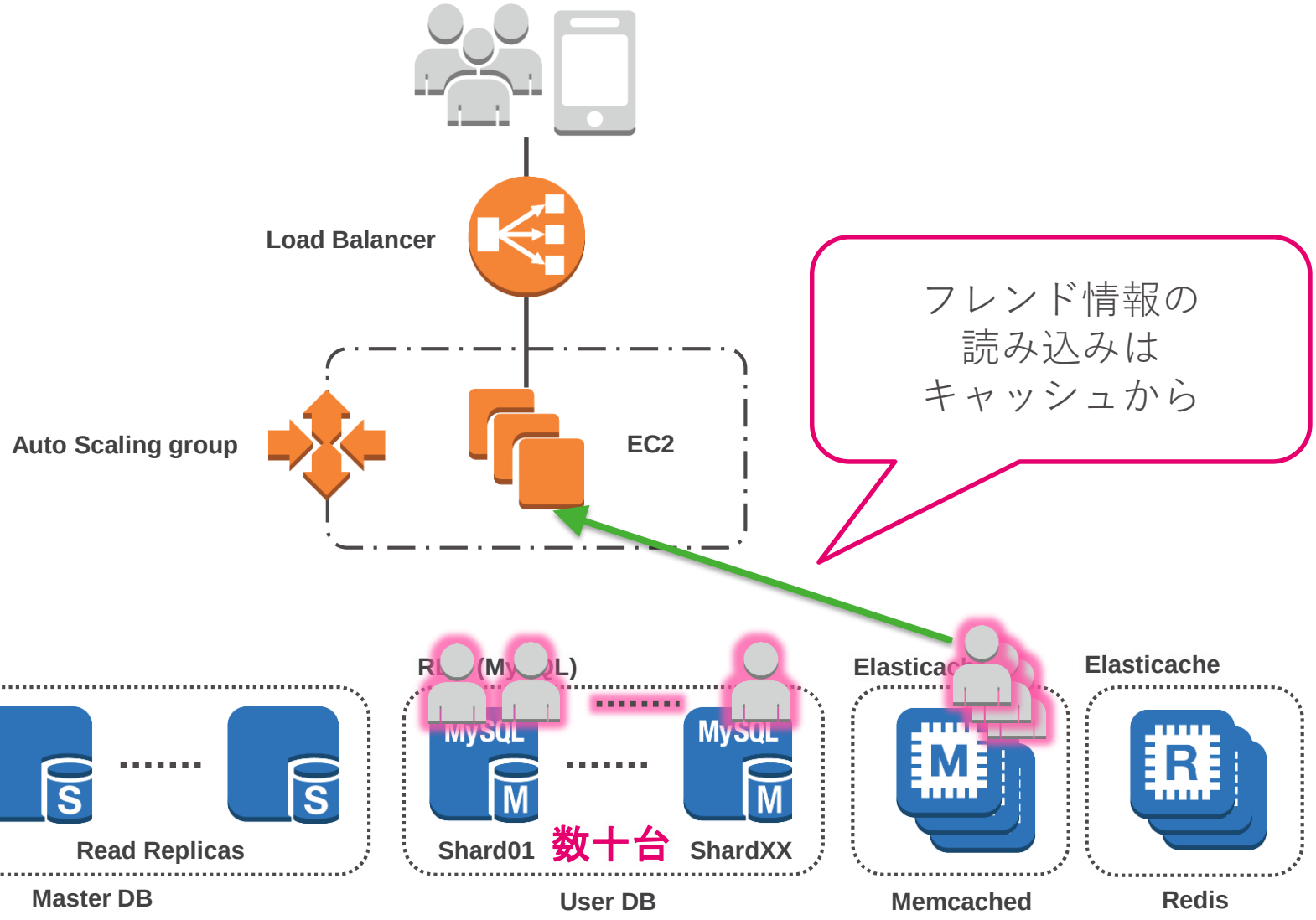
フレンド機能の高速化


Load Balancer
Auto Scaling group
EC2
全shardに分散した
他ユーザのキャラクタ情報
にアクセスが必要
RDS (Aurora)
MySQL
M
S
S
Master
Read Replicas
Master DB
MySQL
M
MySQL
M
Shard01
数十台
ShardXX
User DB
Elasticache
M
Memcached
Elasticache
R
Redis

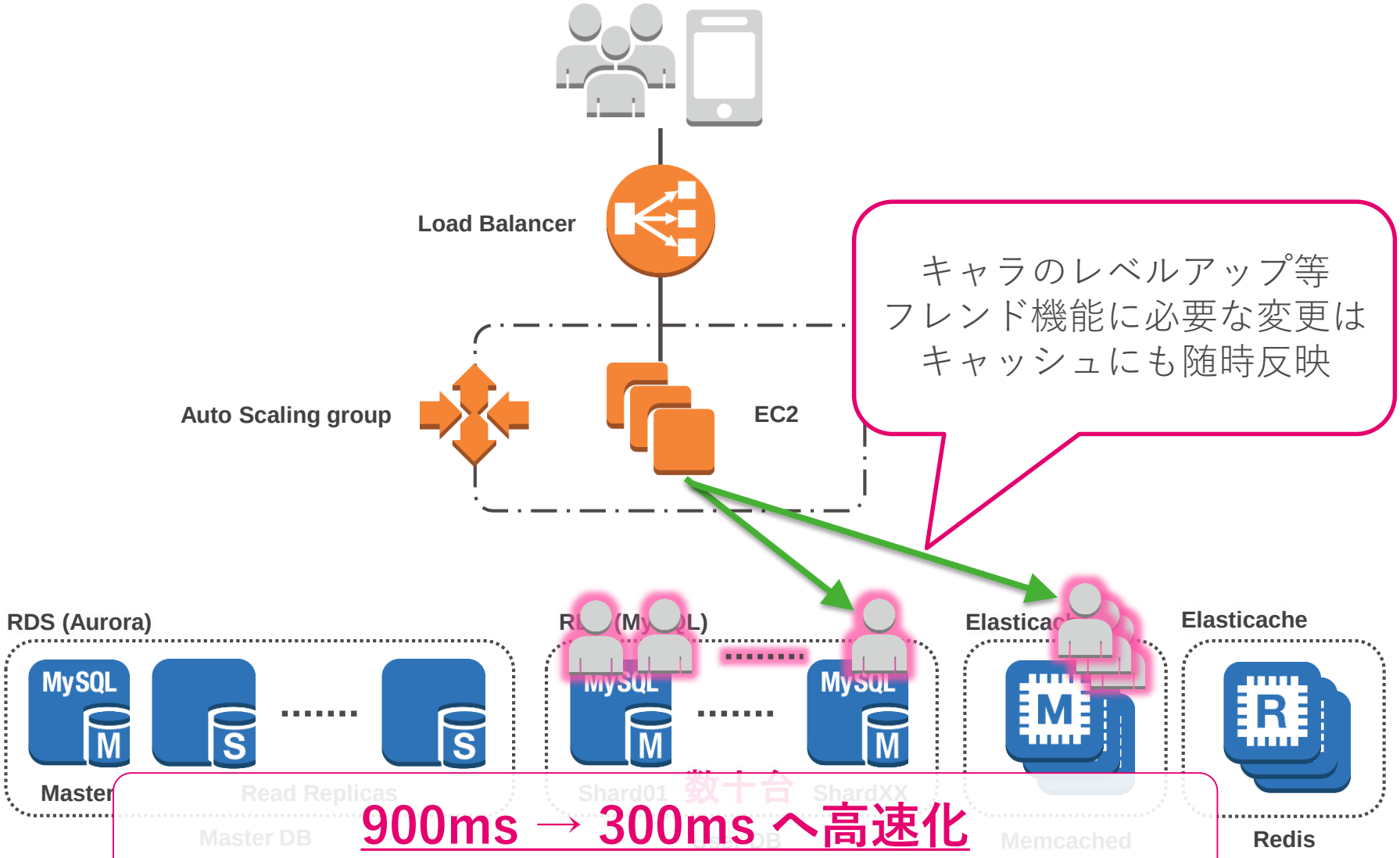
フレンド機能の高速化


Load Balancer
Auto Scaling group
EC2
フレンド機能として
必要になる情報のみ
キャッシュにコピー
copy
RDS (Aurora)
MySQL
M
S
S
Master
Read Replicas
Master DB
MySQL
M
MySQL
M
Shard01
数十台
ShardXX
User DB
M
Memcached
Elasticache
R
Redis

フレンド機能の高速化


Load Balancer
Auto Scaling group
EC2
フレンド情報の
読み込みは
キャッシュから
RDS (Aurora)
MySQL
M
S
S
Master
Read Replicas
Master DB
MySQL
M
MySQL
M
Shard01
数十台
ShardXX
User DB
M
Memcached
Elasticache
R
Redis

フレンド機能の高速化


Load Balancer
Auto Scaling group
EC2
キャラのレベルアップ等
フレンド機能に必要な変更は
キャッシュにも随時反映
RDS (Aurora)
MySQL
M
S
S
Master
Read Replicas
Master DB
Shard01
数十台
ShardXX
MySQL
M
MySQL
M
Elasticache
M
Memcached
Elasticache
R
Redis
900ms → 300ms へ高速化

高速化の結果

変更内容	効果
同一AZへの優先的なクエリ発行	15ms
DBへの過剰な死活監視抑止	50ms
アプリケーションの最適化	35ms
計	**100ms**

約100ms の平均レスポンスタイム短縮に成功！

EC2の台数が約2/3へ ⇒ 年間数千万円の削減
リージョン間の通信費も約2/3に ⇒ 年間数百万円の削減※

より良いUXを提供しながらコスト削減にも成功

これこそが高速化の効果

※ AWSエンタープライズサポートのおかげです

まとめ

AWS上で大規模にスケールアウトした Ruby on Rails アプリケーションの高速化事例について紹介しました

- 対障害性を犠牲にせず同一AZ内の通信を優先的に行う
- コネクションの死活監視といったミドルウェアレベルでの挙動最適化
- インフラの特性を活かしたアプリケーションの最適化

このような最適化を継続していくことで Ruby on Rails の高い生産性・柔軟性を活かしながらも大規模かつ高品質なサービスを、コストを抑えながら提供することができます

備考: 使用しているソフトウェア・サービスの詳細

種別	
Webフレームワーク	Ruby on Rails 4.2.10
言語	Ruby 2.3.5
アプリケーションサーバ	Unicorn 5.3.1
RDB Sharding ライブラリ	Octopus 0.8.4
RDBMS	Aurora MySQL 5.6.10a MySQL 5.6.35 (RDS)
キャッシュ	Memcached 1.4.34 (Elasticache) Redis 2.8.24 (Elasticache)
APM	NewRelic

Akatsuki

ご清聴ありがとうございました