Google Cloud

初公開！オンライン マルチプレイヤーゲームを１人で開発しきった話

甲斐公章

株式会社バンダイナムコスタジオ

スピーカー自己紹介

甲斐 公章

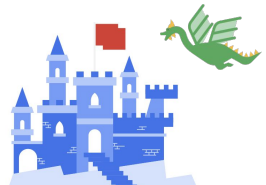
株式会社バンダイナムコスタジオ
マルチプレイヤー
ネットワークユニット・プラットフォーム開発
パート・サーバーエンジニア

2014 年

- 株式会社バンダイナムコ スタジオ入社

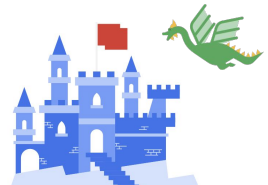
担当経歴

- モバイル向けゲームのバックエンド開発
- リアルタイム通信フレームワーク開発・運用
- サバイバルクイズシティ
- オンライン マルチプレイヤーゲームの汎用システム開発



Agenda

1. サバイバルクイズシティの紹介
2. なぜ 1 人でバックエンドの開発ができたのか
3. Gaming OSS のインフラとパフォーマンス
4. たったこれだけ？必要な 3 つの実装



サバイバルクイズシティの紹介

Google Cloud

サバイバルクイズシティ

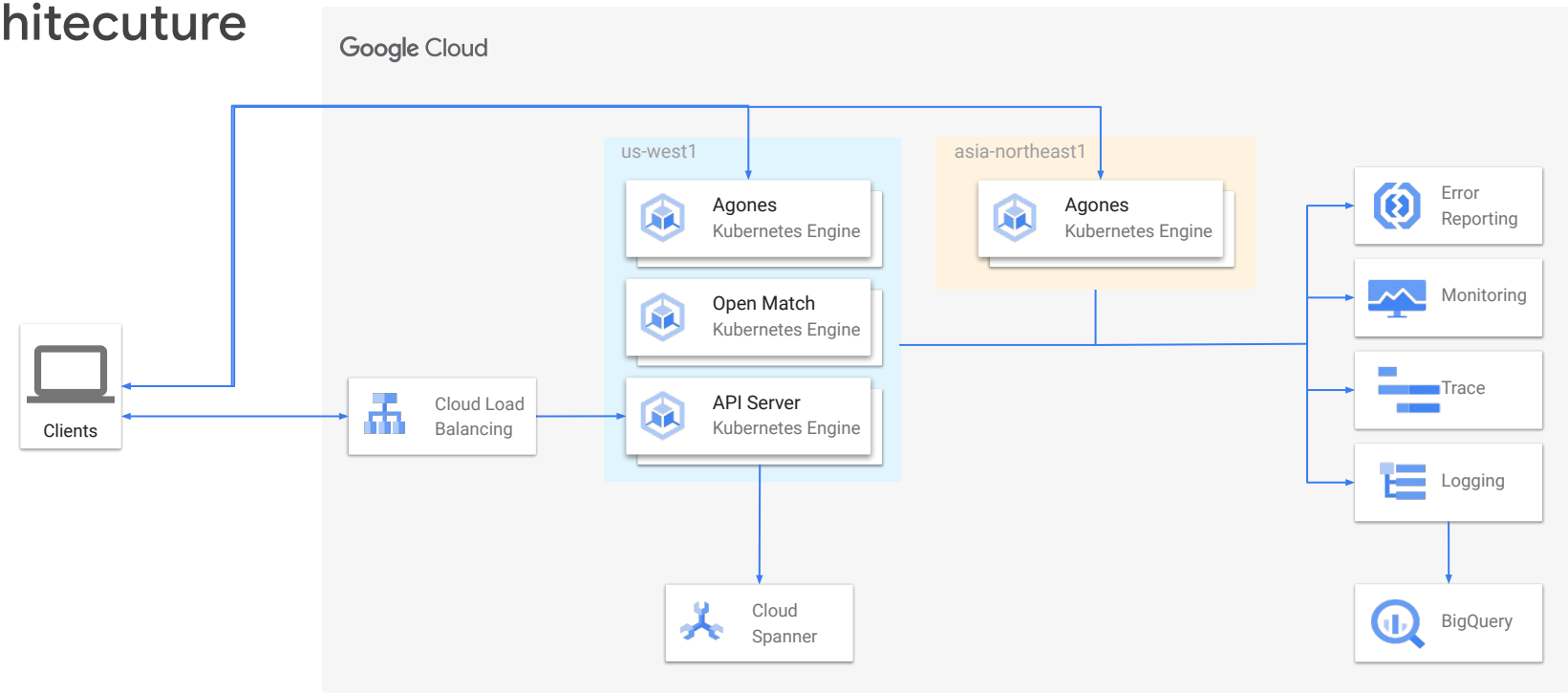
「インディーゲーム レーベル第 1 弾！」

「オンライン × 多人数」

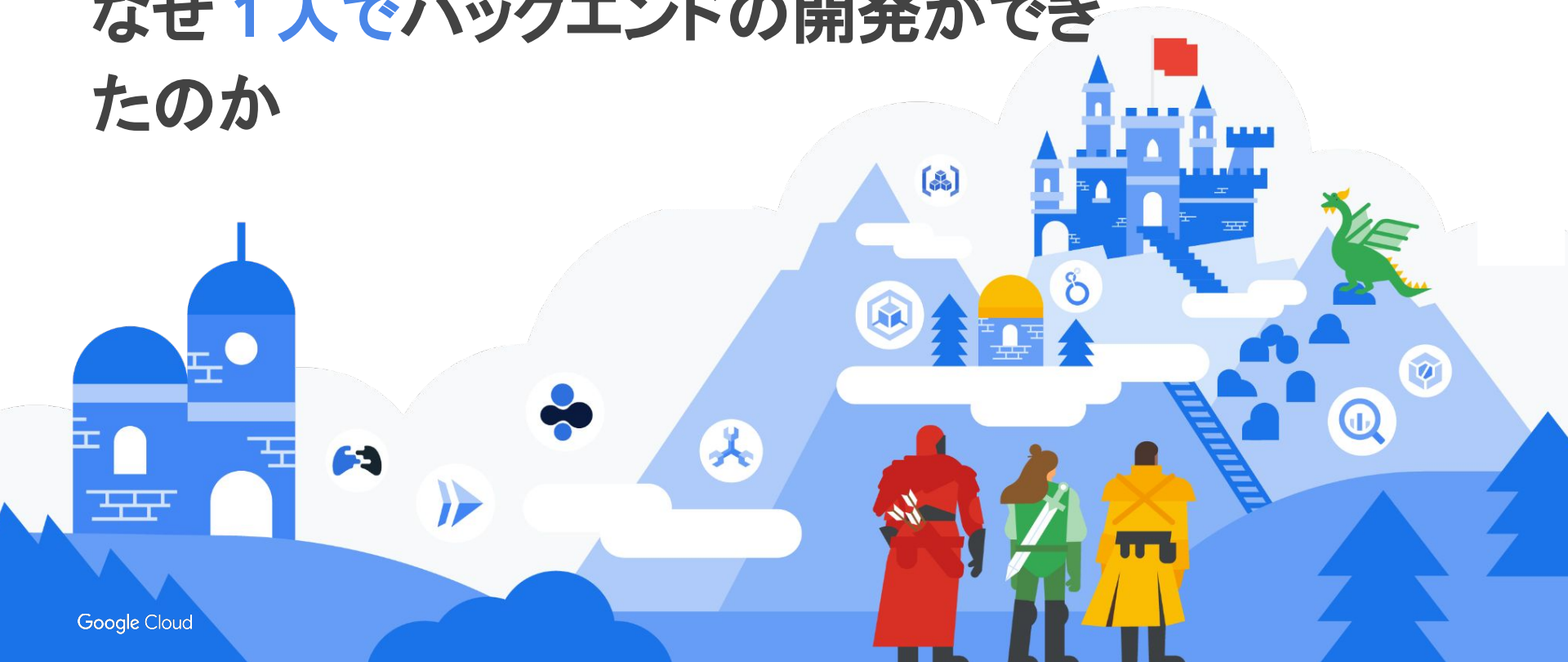
「最大 30 人で戦うクイズアクション」

2022 年 03 月 04 日リリース！

Architecuture

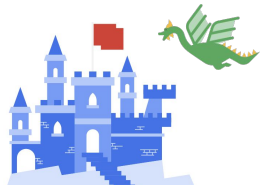


なぜ 1 人でバックエンドの開発ができたのか



そもそも
なぜ 1 人だったのか？

- オンライン マルチプレイヤー ゲーム のバックエンドに精通しているエンジニアが居なかった
- インディータイトルという事で、予算が少なかった



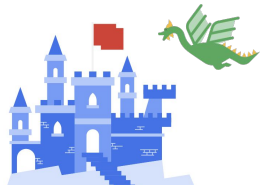
なぜ 1 人でバックエンドの開発ができたのか

Gaming OSS の採用

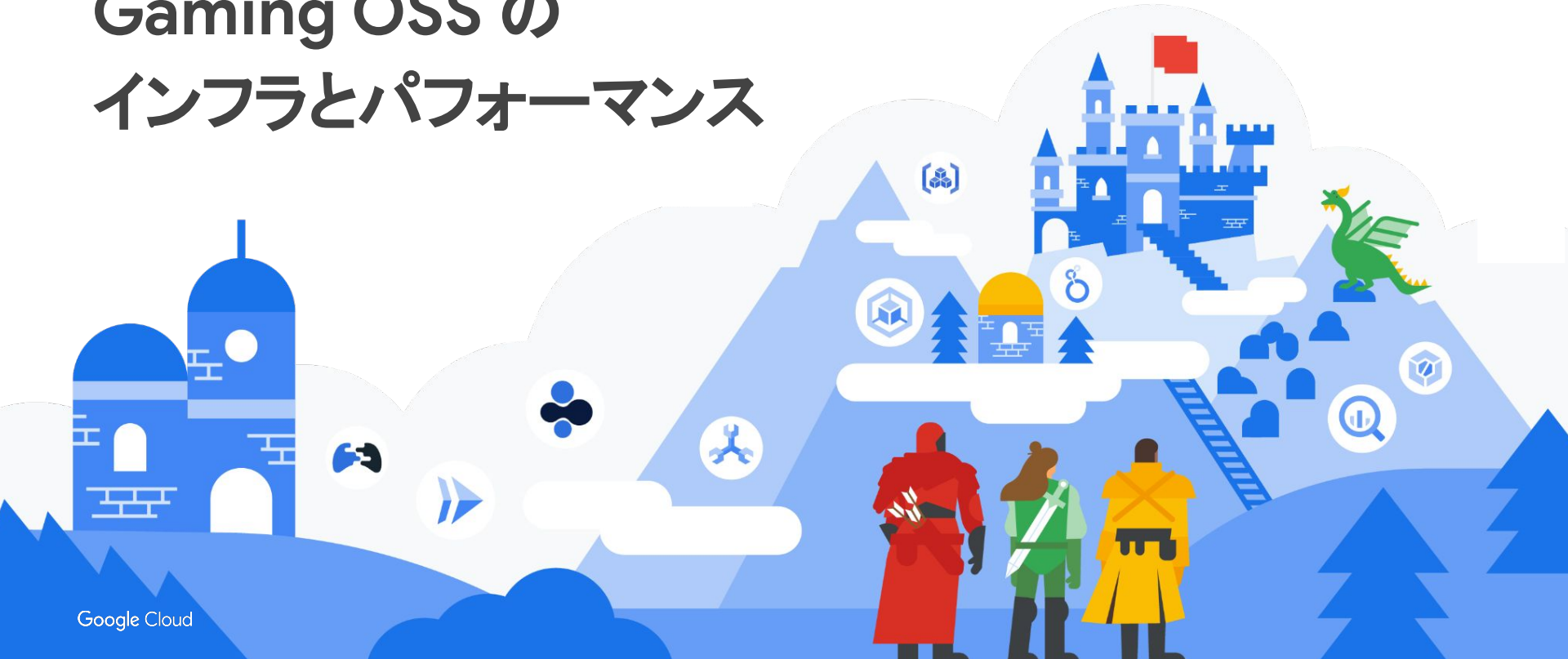
Agones、Open Match を採用したことにより、インフラ構築の大部分を省略することが可能になり、アプリケーション開発に集中することができた。

Gaming OSS の Ecosystem

Gaming OSS は Kubernetes、gRPC を採用した、Micro Service となっており、アプリケーションの一部として取り込むことが容易だった。



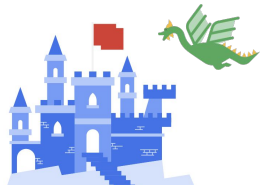
Gaming OSS の
インフラとパフォーマンス



Google Cloud

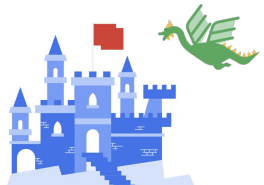
Gaming OSS の インフラとパフォーマンス

- Agones
  - Game Server のオートスケーリング戦略
  - Allocation 負荷試験
- Open Match
  - v0.10 以降のパフォーマンス改善
  - マッチング 負荷試験



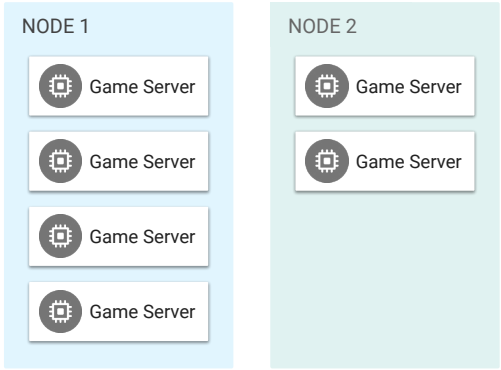
Agones Game Server のオートスケーリング戦略

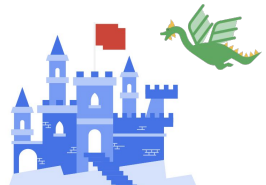
- Fleet Autoscaler + Cluster Autoscaler
- “Packed” スケジューリング



Packed スケジューリング

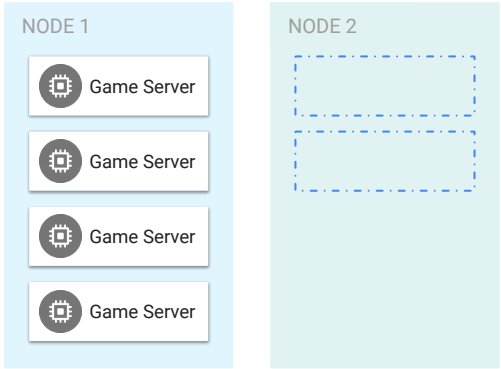
- “Packed” スケジューリングによって、Fleet Autoscaler が pod をノードに集約するようにデプロイする
- ノードのリソースギリギリでデプロイされている可能性が高いので、pod のリソースとリミットの差はなるべく小さく

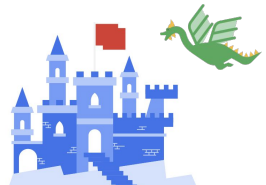




Game Server のスケールダウン

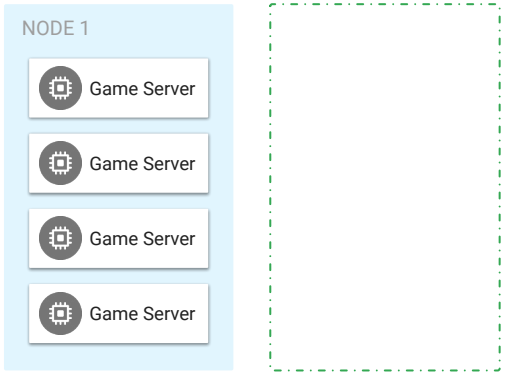
- Fleet Autoscaler が Game Server の数が少ないノードから pod を削除
- プレイ中（Allocated）の場合は、対象外

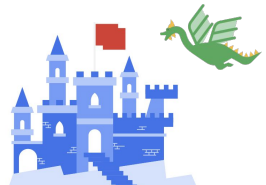




ノードのスケールダウン

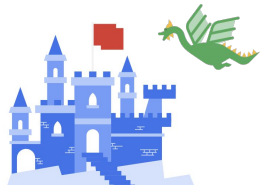
- 最後に Cluster Autoscaler によってノードが削除されます
- 他の pod と混ざると、スケーリングが複雑になるので、Game Server 専用ノードプールでの管理がお勧め
- cluster-autoscaler.kubernetes.io/safe-to-evict: false のラベルを付加してドレイン対象から除外





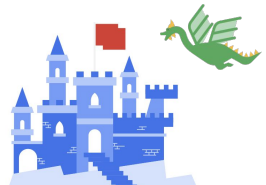
Agones
Allocation 負荷試験

- v1.11.0
- 秒間にどれだけ Game Server を割り当てられるか測定



Agones
負荷試験結果

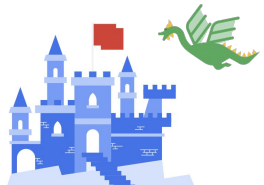
- 100rps で一部の allocation がエラーになった
  - GKE の 秒間の POD 操作上限が原因
- サバイバル クイズシティでは、1 試合 30 人で 30 分程ゲームをプレイするので問題なしと判断
- 既に回避策の事例がネット上にある



Open Match v0.10 以降のパフォーマンス改善

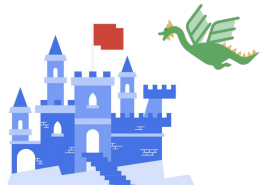
- （2 年前ほどの話ですが）マッチングのパフォーマンスが良くなかった。
- チケットのマッチング条件の登録と検索を Redis から in memory のサービスに移行
- 劇的に改善しているので、過去に断念された方は、再チャレンジする価値あり

※Proposal: Use query for caching and filtering #1123



Open Match
マッチング 負荷試験

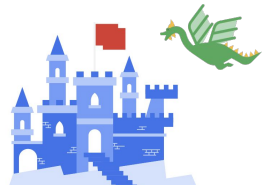
- v1.0.0
- 実際のサバイバルクイズシティのマッチング条件を使用を使用して、E2E で測定（マッチング登録、待機、確定）



Google Cloud

Open Match
負荷試験の結果

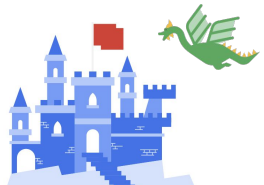
- 2000rps 程でも捌けたので、問題なしと判断
- 別タイトル（1v1）で実施した際には、5000 rps でも問題はありませんでした
- マッチング条件（1v1、30 人など）によって負荷が変わるので、条件毎に測定することをお勧めします



たったこれだけ？必要な 3 つの実装

たったこれだけ？
必要な 3 つの実装

- サバイバルクイズシティのマッチング条件
- マッチング条件の設定方法
- 3 つの実装（Open Match）
  - マッチングロジック（Match Function）
  - マッチングの評価（Evaluator）
  - Agones と Open Match の連携（Director）



マッチング条件

- 30 人マッチング
- プレイヤーのランク帯ごとにマッチング
- 待機時間に応じた条件の緩和
- 待機時間が長いユーザーが優先的にマッチング
- リージョン毎（日本、北米）
- 15 人以上が一定期間待機していた場合は、30 人に満たなくてもマッチング

シティ神

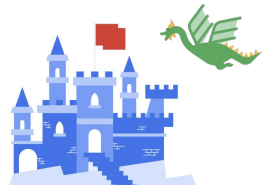
シティ王

シティ名人

シティ玄人

シティ常連

シティ通行人

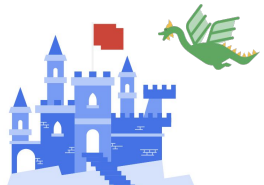


マッチング条件の設定方法

- open match の match profile を使用して、マッチング条件に合わせた pool を設定する
- 複数の profile を同時にリクエスト
- profile の数だけ並列に処理が走り、登録されている pool 数に応じて負荷が増加するので、なるべくシンプルに


```
{"profile":{"Pools":[{"Rank":"GOD","Elapsed":0},{"Rank":"KING","Elapsed":20},{"Rank":"Master","Elapsed":40}]}}
{"profile":{"Pools":[{"Rank":"KING","Elapsed":0},{"Rank":"GOD","Elapsed":20}]}}
{"profile":{"Pools":[{"Rank":"KING","Elapsed":0},{"Rank":"MASTER","Elapsed":20}]}}
{"profile":{"Pools":[{"Rank":"MASTER","Elapsed":0},{"Rank":"KING","Elapsed":20},{"Rank":"GOD","Elapsed":40}]}}
```


※ 簡略化しているので、実際のprofile とは異なります

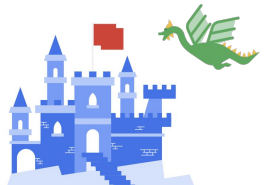


マッチングロジック（Match Function）example code

- Match Profile に登録されている pool に基づいて、チケットが集められる
- 待機時間順でソート
- 集められたチケットを 30 人ずつ、マッチに振り分け

```
func makeMatches(profileName string, tickets []*pb.Ticket) ([]*pb.Match, error) {

    sort.Slice(tickets, func(i, j int) bool {
        return tickets[i].CreateTime.AsTime().Before(tickets[j].CreateTime.AsTime())
    })

    var matches []*pb.Match
    var collectCount = 30
    matchTickets := make([]*pb.Ticket, 0, collectCount)

    for _, ticket := range tickets {
        matchTickets = append(matchTickets, ticket)

        if len(matches) >= collectCount {
            matches = append(matches, &pb.Match{
                MatchId:       uuid.NewString(),
                MatchProfile:  profileName,
                MatchFunction: mmfName,
                Tickets:       matchTickets,
            })

            matchTickets = make([]*pb.Ticket, 0, collectCount)
        }
    }

    // 次のスライド参照
    matches = append(matches, makeMinMatch(profileName, matchTickets))

    return matches, nil
}
```

※ これは疑似コードです

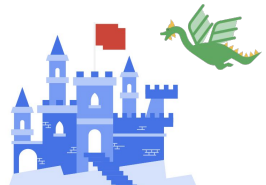


最低 15 人マッチング example code

- 一定期間待機しているユーザーが 15 人以上いる場合は、少ない人数でもマッチング

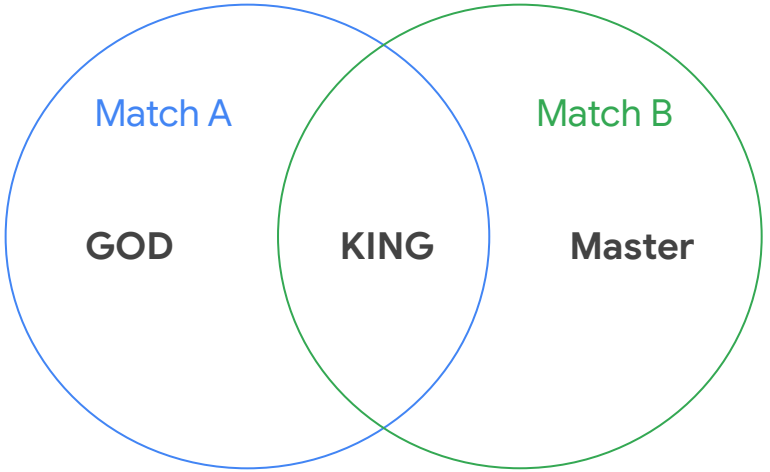
```
func makeMinMatch(profileName string, tickets []*pb.Ticket) *pb.Match {
   var min = 15
   var waitingDuration = time.Minute * 5

   if len(tickets) < min {
       return nil
   }

   var waitingCount = 0
   for _, t := range tickets {
       if time.Since(t.GetCreateTime().AsTime()) > waitingDuration {
           waitingCount++
       }
   }

   if waitingCount < min {
       return nil
   }

   return &pb.Match{
       MatchId:       uuid.NewString(),
       MatchProfile:  profileName,
       MatchFunction: fmt.Sprintf("%s-%d", mmfName, waitingCount),
       Tickets:       tickets,
   }
}
```

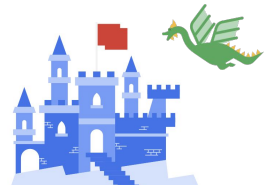
※ これは疑似コードです



マッチングの評価（Evaluator）

- Match Profile は並列に処理される
- 条件範囲が重複した場合、複数のマッチにチケットが登録される可能性がある
- e.g.
  - {rank : GOD, rank : KING}
    {rank : KING, rank : MASTER}
  - KING ランクのユーザーは、二つのグループに含まれているので、重複する可能性がある



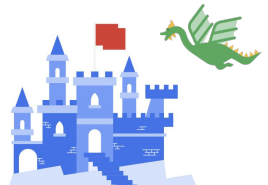


マッチングの評価（Evaluator）example code

- 排他処理の基準
  - マッチング チケット ID
  - User ID
- 既に別のマッチで使用されていた ID が含まれているマッチはリジェクト

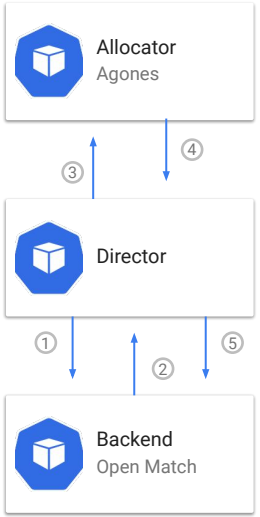
```
func EvaluateProposal(ctx context.Context, proposals []*opb.Match) ([]string, error) {
   usedTicketIds := map[string]string{}
   usedUserIds := map[string]string{}

   for _, p := range proposals {
       for _, t := range p.Tickets {

           if _, ok := usedTicketIds[t.Id]; ok {
               break
           }

           userId, err := extension.GetUserIDFromEx(t.Extensions)
           if err != nil {
               break
           }

           if _, ok := usedUserIds[userId]; ok {
               break
           }

           // ~~~~~~~~~~~~~~~~~~~~~~~~~ 省略 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

※ これは疑似コードです

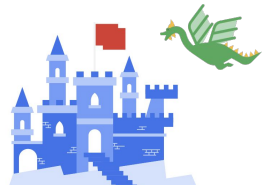


Agones と Open Match の連携（Director）

- Open Match に Director というコンセプトがある
- 処理の流れ



1. Open Match にマッチングのリクエスト
2. マッチングしたマッチ情報の返却
3. Agones に Game Serverの割り当て要求
   （Game Server に meta data を設定）
4. Game Server の IP、Port が返却
5. マッチに登録されているチケットに IP、Port を設定して、マッチ情報確定

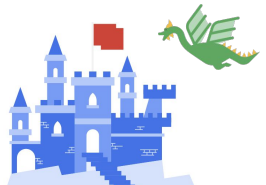


処理の流れ
example code

- FetchMathes API でマッチングをリクエスト
- stream.Recv でマッチ情報を集める

```
func Do(ctx context.Context) {
    req := profiler.CreateRequest()
    stream, err := openMatchClient.FetchMatches(ctx, req)
    if err != nil {
        return
    }

    var matches []*opb.Match
    for {
        resp, err := stream.Recv()

        if err == io.EOF {
            break
        }
        if err != nil {
            break
        }
        matches = append(matches, resp.Match)
    }

    for _, m := range matches {
        go handleMatch(m)
    }
}
```

※ これは疑似コードです



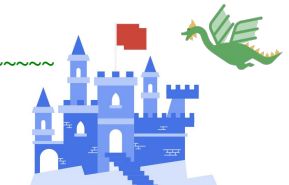
処理の流れ
example code

- Agones Allocation API で Game Server の割り当てを要求
- 要求時に meta 情報を Game Server の Label に設定
    - Kubernetes の label の文字セットしか使用できない
- マルチリージョンは Agones の Multi Cluster Setting は使用せず、Google Cloud の Multi Cluster Services を使用しました。

※マルチクラスターサービス

```
func () handleMatch(match *opb.Match) {
   req := &apb.AllocationRequest{
       Namespace: "default",
       MultiClusterSetting: &apb.MultiClusterSetting{
           Enabled: false,
       },
       MetaPatch: &apb.MetaPatch{
           Labels: map[string]string{
               allocator.GameMode:       "SURVIVAL",
               allocator.GameServerMode: "RANK_MATCH",
               allocator.MatchID:        uuid.NewString(),
           },
       },
   }
   region, err := extension.GetRegionFromEx(match.Extensions)
   if err != nil {
     return
   }
   var gs *apb.AllocationResponse
   var err error
   if region == "JP" {
       gs, err = allocationClientJP.AllocateGameServer(req)
   } else {
       gs, err = allocationClientUS.AllocateGameServer(req)
   }

   if err != nil {
       return
   }
   // ~~~~~~~~~~~~~~~~~~~~~~~~~~ 省略 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

※ これは疑似コードです

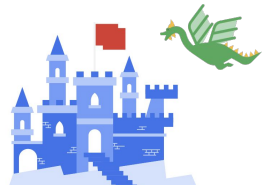


処理の流れ
example code

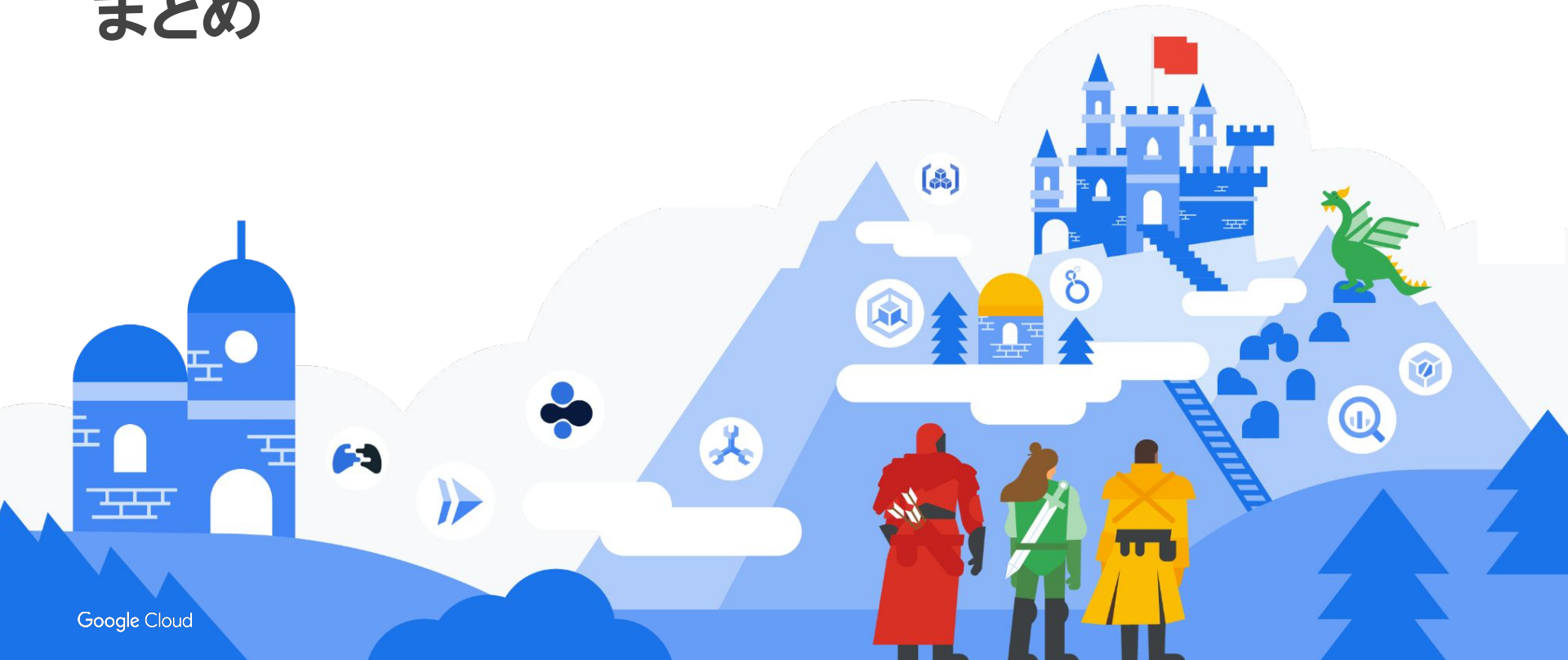
- 割り当てられた、Game Server の IP、Port を マッチに登録されているチケットに Assign し、AssignTickets API をリクエスト
- クライアントに接続先情報が返るので、それを元に Game Server に直接接続

```
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~ 省略 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    var ids []string
    for _, t := range match.Tickets {
        ids = append(ids, t.GetId())
    }


    req := &opb.AssignTicketsRequest{
        Assignments: []*opb.AssignmentGroup{
            {
                TicketIds: ids,
                Assignment: &opb.Assignment{
                    Connection: fmt.Sprintf("%s:%d", gs.Address, gs.Ports[0].Port),
                },
            },
        },
    }

    res, err := openMatchClient.AssignTickets(context.Background(), req)
    if err != nil {
        return
    }

}
```

※ これは疑似コードです



まとめ



まとめ

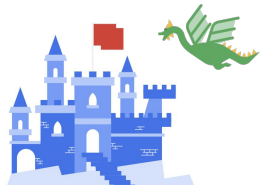
01 | Don’t reinvent the wheel ！

（実装が必要なのは、Match Function、Evaulator、Director のたった 3 つ！）

02 | “Given enough eyeballs, all bugs are shallow ”

- Linus Torvalds

（使用者が増えれば、バグも減り、より使いやすい OSS に！）



Google Cloud

Thank you