Amazon GameLift

FlexMatch Developer Guide

Version

Amazon GameLift: FlexMatch Developer Guide

Table of Contents

What is Amazon GameLift FlexMatch?

GameLift FlexMatch is a customizable matchmaking service for multiplayer games. With FlexMatch, you can build a custom set of rules that defines what a multiplayer match looks like for your game, and determines how to evaluate and select compatible players for each match. You can also customize key aspects of the matchmaking process to fit your game, including fine-tuning the matching algorithm.

FlexMatch is available both as a GameLift game hosting solution (including Realtime Servers) and as a standalone matchmaking service. You can implement FlexMatch as a standalone feature with games that use peer-to-peer architecture, or host game servers on-premises or on other cloud compute solutions (including GameLift FleetIQ). This guide provides detailed information on how to build a matchmaking system for any of these scenarios.

FlexMatch gives you the flexibility to set matchmaking priorities depending on your game requirements. For example, you can do the following:

- Find a balance between match speed and quality. Set match rules to quickly find matches that are good enough, or have players wait a little longer to find the best possible match for an optimum player experience.
- Make matches based on well-matched players or well-matched teams. Create a match where all players have similar characteristics, such as skill or experience. Alternatively, form matches where the combined characteristics of each team are similar, even if the characteristics of individual players are more varied.
- Prioritize how player latency factors into a match. Set a hard limit on latency for all players in a match, make sure that everyone in a match experiences similar latency, or do both.

Ready to start working with FlexMatch?
For step-by-step guidance on getting your game up and running with FlexMatch, see the following topics:

- FlexMatch integration with GameLift hosting (p. 8)
- GameLift FlexMatch integration for standalone matchmaking (p. 7)

Key FlexMatch features

The following features are available with all FlexMatch scenarios, whether you use FlexMatch as a standalone service or with GameLift game hosting.

- **Customizable player matching.** Design and build matchmakers to suit all of the game modes that you offer your players. Build a set of custom rules to evaluate key player attributes (such as skill level or role) and geographic latency data to form great player matches for your game.
- **Latency-based matching.** Provide player latency data and create match rules that require players in a match to have similar response times. This feature is useful when your player matchmaking pools span multiple geographic regions.
- **Support for match sizes up to 200 players.** Create matches of up to 40 players using match rules that are customized for your game. Create matches of up to 200 players using a matching process that uses a streamlined custom matching process to keep player wait times manageable.

- **Player acceptance.** Require players to opt in to a proposed match before finalizing the match and starting a game session. Use this feature to initiate your custom acceptance workflow and report player responses to FlexMatch before placing a new game session for the match. If not all players accept a match, the proposed match fails and players who did accept automatically return to the matchmaking pool.

  Player parties support. Generate matches for groups of players who want to play together on the same team. Use FlexMatch to find additional players to fill out the match as needed.
- **Expandable matching rules.** Gradually relax the match requirements after a certain amount of time has passed without finding a successful match. Rule expansion lets you decide where and when to relax the initial match rules, so that players can get into playable games more quickly.
- **Match backfill.** Fill the empty player slots in an existing game session with well-matched new players. Customize when and how to request new players, and use the same custom match rules to find additional players.

FlexMatch with GameLift hosting

For games that are hosted with GameLift, FlexMatch offers the following additional features. These are available when using GameLift to host custom game servers, or when using Realtime Servers. Games that are hosted on Amazon Elastic Compute Cloud (Amazon EC2) resources with GameLift FleetIQ must implement FlexMatch as a standalone feature.

- **Game session placement.** When a match is successfully made, FlexMatch automatically requests a new game session placement from GameLift. The data generated during matchmaking, including player IDs and team assignments, is provided to the game server so that it can use that information to start the game session for the match. FlexMatch then passes back game session connection information so that game clients can join the game. To minimize the latency experienced by players in a match, game session placement with GameLift can also use regional player latency data, if provided.
- **Automatic match backfill.** With this feature enabled, FlexMatch automatically sends a match backfill request when a new game session starts with unfilled player slots. Your matchmaking system starts the game session placement process with a minimum number of players, and then quickly fills the remaining slots. You cannot use automatic backfill to replace players who drop out of a matched game session.

Pricing for GameLift FlexMatch

GameLift charges for instances by duration of use and for bandwidth by quantity of data transferred. If you host your games on GameLift servers, FlexMatch usage is included in the fees for GameLift. If you host your games on another server solution, FlexMatch usage is charged separately. For a complete list of charges and prices for GameLift, see Amazon GameLift Pricing.

For information on calculating the cost of hosting your games or matchmaking with GameLift, see Generating GameLift pricing estimates, which describes how to use the AWS Pricing Calculator.

How Amazon GameLift FlexMatch works

This topic provides an overview of the GameLift FlexMatch service, including the core components of a FlexMatch system and how they interact.

You can use FlexMatch with games that use GameLift managed hosting or with games that use another hosting solution. Games that are hosted on GameLift servers, including Realtime Servers, use the

integrated GameLift service to automatically locate available game servers and start game sessions for the matches. Games that use FlexMatch as a standalone service, including GameLift FleetIQ, must coordinate with the existing hosting system to assign hosting resources and start game sessions for the matches.

For detailed guidance on setting up FlexMatch for your games, see Getting started with FlexMatch (p. 7).

Matchmaking components

A FlexMatch matchmaking system includes some or all of the following components.

GameLift components

These are GameLift resources that control how the FlexMatch service performs matchmaking for your game. They are created and maintained using GameLift tools, including the console and the AWS CLI or, alternatively, programmatically using the AWS SDK for GameLift.

- **FlexMatch matchmaking configuration (also called a matchmaker)** – A matchmaker is a set of configuration values that customizes the matchmaking process for your game. A game can have multiple matchmakers, each configured for different game modes or experiences as needed. When your game sends a matchmaking request to FlexMatch, it specifies which matchmaker to use.
- **FlexMatch matchmaking rule set** – A rule set contains all the information that is needed to evaluate players for a potential matches and approve or reject. The rule set defines a match's team structure, declares the player attributes that are used for evaluation, and provides rules that describe the criteria for an acceptable match. Rules can apply to individual players, teams, or the entire match. For example, a rule might require that every players in the match choose the same game map, or it might require that all teams have similar player skill average.
- **GameLift game session queue (for FlexMatch with GameLift managed hosting only)** – A game session queue locates available hosting resources and starts a new game session for the match. The queue's configuration determines where GameLift looks for available hosting resources and how to select the best available host for a match.

Custom components

The following components encompass functionality that's required for a complete FlexMatch system that you must implement based on the architecture of your game.

- **Player interface for matchmaking** – This interface enables players to join a match. At a minimum, it initiates a matchmaking request through the client matchmaking service component and provides player-specific data, such as skill level and latency data, as needed for the matchmaking process.

  Note
  As a best practice, communication with the FlexMatch service should be done by a backend service, not from a game client.
- **Client matchmaking service** – This service fields the player join requests from the player interface, generates matchmaking requests, and sends them to the FlexMatch service. For requests in process, it monitors matchmaking events, tracks matchmaking status, and takes action as needed. Depending on how you manage game session hosting in your game, this service may return game session connection information back to players. This component uses the AWS SDK with the GameLift API to communicate with the FlexMatch service.
- **Match placement service (for FlexMatch as a standalone service only)** – This component works with your existing game hosting system to locate available hosting resources and start new game sessions for matches. The component must get the matchmaking results and extract the information needed to start a new game session, including player IDs, attributes, and team assignments for all players in the match.

FlexMatch matchmaking process

This topic describes a basic matchmaking scenario and interactions between the various your game components and the FlexMatch service.

Request matchmaking for players

A player using your game client clicks a "Join Game" button. This action causes your client matchmaking service to send a matchmaking request to FlexMatch. The request identifies the FlexMatch matchmaker to use when fulfilling the request. The request also includes player information that your custom matchmaker requires, such as skill level, play preferences, or geographic latency data. You can make matchmaking requests for one player or multiple players.

Add requests to the matchmaking pool

When FlexMatch receives the matchmaking request, it generates a matchmaking ticket and adds it to the matchmaker's ticket pool. The ticket remains in the pool until it is matched or a maximum time limit is reached. Your client matchmaking service is periodically notified about matchmaking events, including changes in ticket status.

Build a match

Your FlexMatch matchmaker continually runs the following process on all tickets in its pool:

1. The matchmaker sorts the pool by ticket age, then begins building a potential match starting with the oldest ticket.
2. The matchmaker adds a second ticket to the potential match and evaluates the result against your custom matchmaking rules. If the potential match passes evaluation, the ticket's players are assigned to a team.
3. The matchmaker adds the next ticket in sequence and repeats the evaluation process. When all player slots have been filled, the match is ready.

Matchmaking for large matches (41 to 200 players) uses a modified version of the process described above so that it can build matches in a reasonable time frame. Instead of evaluating each ticket individually, the matchmaker divides a pre-sorted ticket pool into potential matches, and then balances each match based on a player characteristic that you've specified. For example, a matchmaker might pre-sort tickets based on similar low-latency locations, and then use post-match balancing to make sure that the teams are evenly matched by player skill.

Report matchmaking results

When an acceptable match is found, all matched tickets are updated and a successful matchmaking event is generated for each matched ticket.

- FlexMatch as a standalone service: Your game receives match results in a successful matchmaking event. Result data includes a list of all matched players and their team assignments. If your match requests contain player latency info, the results also suggest an optimal geographic location for the match.
- FlexMatch with a GameLift hosting solution: Match results are automatically passed to a GameLift queue for game session placement. The matchmaker determines which queue is used for game session placement.

Start a game session for the match

After a proposed match is successfully formed, a new game session is started. Your game servers must be able to use the matchmaking result data, including player IDs and team assignments, when setting up a game session for the match.

- FlexMatch as a standalone service: Your custom match placement service gets match result data from successful matchmaking events, and connects to your existing game session placement system to locate an available hosting resource for the match. After a hosting resource is found,

the match placement service coordinates with your existing hosting system to start a new game session and acquire connection information.

- FlexMatch with a GameLift hosting solution: The game session queue locates the best available game server for the match. Depending on how the queue is configured, it tries to place the game session with the lowest-cost resources and where players will experience low latency (if player latency data is provided). Once the game session is successfully placed, the GameLift service prompts the game server to start a new game session, passing on the matchmaking results and other optional game data.

Connect players to the match

After a game session is started, players connect to the session, claim their team assignment, and begin gameplay.

- FlexMatch as a standalone service: Your game uses the existing game session management system to provide connection information back to players.
- FlexMatch with a GameLift hosting solution: On a successful game session placement, FlexMatch updates all of the matched tickets with game session connection information and a player session ID.

Setting up FlexMatch

GameLift FlexMatch is an AWS service, and you must have an AWS account to use this service. Creating an AWS account is free. For more information on what you can do with an AWS account, see Getting Started with AWS.

If you are using FlexMatch with other GameLift solutions, see the following topics:

- Setting up access for GameLift hosting and Realtime Servers
- Setting up access for hosting on Amazon EC2 with GameLift FleetIQ

To set up your account for GameLift

1. **Get an account.** Open Amazon Web Services and choose **Sign In to the Console**. Follow the prompts to either create a new account or sign in to an existing one.
2. **Set up an administrative user group.** Open the AWS Identity and Access Management (IAM) service console and follow the steps to create or update users or user groups. IAM manages access to your AWS services and resources. Everyone who accesses your FlexMatch resources, using the GameLift console or by calling GameLift APIs, must be given explicit access. For detailed instructions on using the console (or the AWS CLI or other tools) to set up user groups, see Creating IAM Users.
3. **Attach a permissions policy to your user or user group.** Access to AWS services and resources are managed by attaching an IAM policy to a user or user group. Permissions policies specify a set of AWS services and actions a user has to have access to.

   For GameLift, you must create a custom permissions policy and attach it to each user or user group. A policy is a JSON document. Use the example below to create your policy.

The following example illustrates an inline permissions policy with administrative permissions for all GameLift resources and actions. You can choose to limit access by specifying only FlexMatch-specific items.

```
{
"Version": "2012-10-17",
"Statement":
  {
    "Effect": "Allow",
    "Action": "gamelift:*",
    "Resource": "*"
  }
}
```

Getting started with FlexMatch

Use the resources in this section to help you get started with buildiing a matchmaking system with FlexMatch.

Topics

GameLift FlexMatch integration for standalone matchmaking

This topic outlines the complete integration process for implementing FlexMatch as a standalone matchmaking service. Use this process if your multiplayer game is hosted using peer-to-peer, custom-configured on-premises hardware, or other cloud compute primitives. This process is also for use with GameLift FleetIQ, which is a hosting optimization solution for games that are hosted on Amazon EC2. If you're hosting your game using GameLift managed hosting (including Realtime Servers), see FlexMatch integration with GameLift hosting (p. 8).

Before you start integration, you must have an AWS account and set up access permissions for the GameLift service. For details, see Setting up FlexMatch (p. 6). All essential tasks related to creating and managing GameLift FlexMatch matchmakers and rule sets can be done using the Amazon GameLift console, but you might also want to

1. **Create a FlexMatch matchmaking rule set.** Your custom rule set provides complete instructions for how to construct a match. In it, you define the structure and size of each team. You also provide a set of requirements that a match must meet to be valid, which FlexMatch uses to include or exclude players in a match. These requirements might apply to individual players. You can also customize the FlexMatch algorithm in the rule set, such as to build large matches with up to 200 players. See these topics:

   - Build a FlexMatch rule set (p. 15)
   - FlexMatch rule set examples (p. 25)
2. **Set up notifications for matchmaking events.** Use notifications to track FlexMatch matchmaking activity, including the status of pending match requests. This is the mechanism that's used to deliver the results of a proposed match. Since matchmaking requests are asynchronous, you need a way to track the status of requests. Using notifications is the preferred option for this. See these topics:

   - Set up FlexMatch event notifications (p. 40)
   - FlexMatch matchmaking events (p. 70)
3. **Set up a FlexMatch matchmaking configuration.** Also called a matchmaker, this component receives matchmaking requests and processes them. You configure a matchmaker by specifying a rule set, notification target, and maximum wait time. You can also enable optional features. See these topics:

   - Design a FlexMatch matchmaker (p. 10)
   - Create a matchmaking configuration (p. 12)
4. **Build a client matchmaking service.** Create or expand a game client service with functionality to build and send matchmaking requests to FlexMatch. To build matchmaking requests, this

component must have mechanisms to get the player data required by the matchmaking rule set and, optionally, regional latency information. It must also have a method for creating and assigning unique ticket IDs for each request. You might also choose to build a player acceptance workflow that requires players to opt in to a proposed match. This service must also monitor matchmaking events to get match results and initiate game session placement for successful matches. See this topic:

- Add FlexMatch to a game client (p. 43)

5. **Build a match placement service.** Create a mechanism that works with your existing game hosting system to locate available hosting resources and start new game sessions for successful matches. This component must be able to use match results information to get an available game server and start a new game session for the match. You might also want to implement a workflow to make match backfill requests, which uses matchmaking to fill open slots in matched game sessions that are already running.

FlexMatch integration with GameLift hosting

FlexMatch is available with the managed GameLift hosting for custom game servers and Realtime Servers. To add FlexMatch matchmaking to your game, complete the following tasks.

- **Set up a matchmaker.** A matchmaker receives matchmaking requests from players and processes them. It groups players based on a set of defined rules and, for each successful match, creates a new game sessions and player sessions. Follow these steps to set up a matchmaker:
  - **Create a rule set.** A rule set tells the matchmaker how to construct a valid match. It specifies team structure and specifies how to evaluate players for inclusion in a match. See these topics:
    - Build a FlexMatch rule set (p. 15)
    - FlexMatch rule set examples (p. 25)
  - **Create a game session queue.** A queue locates the best region for each match and creates a new game session in that region. Use an existing queue or create a new one for matchmaking. See this topic:
    - Create a queue
  - **Set up notifications (optional).** Since matchmaking requests are asynchronous, you need a way to track the status of requests. Notifications is the preferred option. See this topic:
    - Set up FlexMatch event notifications (p. 40)
  - **Configure a matchmaker.** Once you have a rule set, queue, and notifications target, create the configuration for your matchmaker. See these topics:
    - Design a FlexMatch matchmaker (p. 10)
    - Create a matchmaking configuration (p. 12)
- **Integrate FlexMatch into your game client service.** Add functionality to your game client service to start new game sessions with matchmaking. Requests for matchmaking specify which matchmaker to use and provide the necessary player data for the match. See this topic:
  - Add FlexMatch to a game client (p. 43)
- **Integrate FlexMatch into your game server.** Add functionality to your game server to start game sessions that are created through matchmaking. Requests for this type of game session include match-specific information, including players and team assignments. The game server needs to access and use this information when constructing a game session for the match. See this topic:
  - Add FlexMatch to a GameLift-hosted game server (p. 47)
- **Set up FlexMatch backfill (optional).** Request additional player matches to fill open player slots in existing games. You can turn on automatic backfill to have GameLift manage backfill requests. Or you can manage backfill manually by adding functionality to your game client service or game server to initiate match backfill requests. See this topic:
  - Backfill existing games with FlexMatch (p. 49)

Note
FlexMatch backfill is currently not available for games using Realtime Servers.

Buiding a GameLift FlexMatch matchmaker

A FlexMatch matchmaker process does the work of building a game match. It manages the pool of matchmaking requests received, forms teams for a match, processes and selects players to find the best possible player groups, and initiates the process of placing and starting a game session for the match. This topic describes the key aspects of a matchmaker and how to configure one customized for your game.

For a detailed description of how a FlexMatch matchmaker processes the matchmaking requests it receives, see FlexMatch matchmaking process (p. 4).

Topics

Design a FlexMatch matchmaker

This topic provides guidance on how to design a matchmaker that fits your game.

Configure a basic matchmaker

At a minimum, a matchmaker needs the following elements:

- The **rule set** determines the size and scope of teams for a match and defines a set of rules to use when evaluating players for a match. Each matchmaker is configured to use one rule set. See Build a FlexMatch rule set (p. 15) and FlexMatch rule set examples (p. 25).
- The **notification target** receives all matchmaking event notifications. You need to set up an Amazon Simple Notification Service (SNS) topic and then add the topic ID to the matchmaker. See more information on setting up notifications at Set up FlexMatch event notifications (p. 40).
- The **request timeout** determines how long matchmaking requests can remain in the request pool and be evaluated for potential matches. Once a request has timed out, it has failed to make a match and is removed from the pool.
- When using FlexMatch with GameLift managed hosting, the **game session queue** finds the best available resources to host a game session for the match, and starts a new game session. Each queue is configured with a list of AWS Regions and resource types (including Spot or On-Demand Instances) that determine where game sessions can be placed. For more information on queues, see Using multi-region queues.

Choose an AWS Region for the matchmaker

Decide where you want matchmaking activity to take place, and create your matchmaker (matchmaking configuration and rule set) in that Region. All requests for the matchmaker are sent to a ticket pool

there, where they are sorted and evaluated for viable matches. Once matches are made, players can then be directed to game sessions in any location that is supported your hosting solution.

When choosing AWS Regions for your matchmakers, consider how location might affect their performance and how to optimize the match experience for intended players. We recommend the following best practices:

- Place a matchmaker in an AWS region that is close to your players—and your client service that sends FlexMatch matchmaking requests. This approach decreases the latency effect on your matchmaking request workflow and makes it more efficient.
- If your game reaches a global audience, consider creating matchmakers in multiple regions and routing match requests to the matchmaker that is closest to the player. In addition to boosting efficiency, this causes ticket pools to form with players who are geographically near each other, which improves the matchmaker's ability to match players based on latency requirements.
- When using FlexMatch with GameLift managed hosting, place your matchmaker and the game session queue that it uses in the same AWS Region. This helps to minimize communication latency between the matchmaker and queue.

The FlexMatch resources MatchmakingConfiguration and MatchmakingRuleSet can be placed in the following GameLift-supported AWS Regions: US East (N. Virginia), US West (Oregon), EU Central (Frankfurt), EU West (Ireland), Asia Pacific Southeast (Sydney), Asia Pacific Northeast (Seoul and Tokyo), and China (Beijing and Ningxia Regions).

Add optional elements

In addition to these minimum requirements, you can configure your matchmaker with the following additional options. If you are using FlexMatch with a GameLift hosting solution, many features are built in. If you're using FlexMatch as a standalone matchmaking service, you might want to build these features into your system.

Player Acceptance

You can configure a matchmaker to require that all players who are selected for a match must accept participation. If your system requires acceptance, all players must be given the option to accept or reject a proposed match. A match must receive acceptances from all players in the proposed match before it can be completed. If any player rejects or fails to accept a match, the proposed match is discarded and the tickets are handled as follows. Tickets where all players in the ticket accepted the match are returned to the matchmaking pool for continued processing. Tickets where at least one player rejected the match or failed to respond are put into a failure status and are no longer processed. Player acceptance requires a time limit; all players must accept a proposed match within the time limit for the match to continue.

Backfill Mode

Use FlexMatch backfill to keep your game sessions filled with well-matched new players throughout the life span of the game session. When handling backfill requests, FlexMatch uses the same matchmaker as was used to match the original players. You can customize how backfill tickets are prioritized with tickets for new matches, putting backfill tickets to either the front or end of the line. This means that, as new players enter the matchmaking pool, they are more or less likely to be placed in an existing game than in a newly formed game.

Manual backfill is available whether your game uses FlexMatch with managed GameLift hosting or with other hosting solutions. Manual backfill gives you the flexibility to decide when to trigger a backfill request. For example, you may want to add new players only during certain phases of your game or only when certain conditions exist.

Automatic backfill is available only for games that use managed GameLift hosting. With this feature enabled, if a game session starts with open player slots, GameLift begins automatically generating

backfill requests for it. This feature allows you to set up matchmaking so that new games are started with a minimum number of players and then quickly filled as new players enter the matchmaking pool. You can turn off automatic backfill at any time during the game session life span.

Game Properties

For games that use FlexMatch with GameLift managed hosting, you can provide additional information to be passed to a game server whenever a new game session is requested. This can be a useful way to pass game mode configurations that are needed to start a game session for the type of matches being created. All game sessions for matches that are created by a matchmaker receive the same set of game properties. You can vary game property information by creating different matchmaking configurations.

Reserved Player Slots

You can designate that certain player slots in each match be reserved and filled at a later time. This is done by configuring the "additional player count" property of a matchmaking configuration.

Custom Event Data

Use this property to include a set of custom information in all matchmaking-related events for the matchmaker. This feature can be useful for tracking certain activity unique to your game, including tracking performance of your matchmakers.

Create a matchmaking configuration

To set up a GameLift FlexMatch matchmaker to process matchmaking requests, create a matchmaking configuration. Use either the GameLift console or the AWS Command Line Interface (AWS CLI). For more information about creating a matchmaker, see Design a FlexMatch matchmaker (p. 10).

Create a matchmaker for GameLift hosting

Before creating a matchmaking configuration, create a rule set (p. 23) and a GameLift game session queue to use with the matchmaker.

Console

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/home.
2. Switch to the AWS Region where you want to create your matchmaker. For a list of Regions that support FlexMatch matchmaking configurations, see Choose an AWS Region for the matchmaker (p. 10).
3. In the navigation pane, choose **FlexMatch**, **Matchmaking configurations**.
4. On the **Matchmaking configurations** page, choose **Create configuration**.
5. On the **Define configuration details** page, under **Matchmaking configuration details**, do the following:

    a. For **Name**, enter a matchmaker name that can help you identify it in a list and in metrics. The matchmaker name must be unique within the Region. Matchmaking requests identify which matchmaker to use by its name and Region.

    b. (Optional) For **Description**, add a description to help identify the matchmaker.

    c. For **Rule set**, choose a rule set from the list to use with the matchmaker. The list contains all rule sets that you've created in the current Region.

    d. For **FlexMatch mode**, choose **Managed** for GameLift managed hosting. This mode prompts FlexMatch to pass successful matches to the specified game session queue.

    e. For **AWS Region**, choose the Region where you configured the game session queue that you want to use with the matchmaker.

f. For **Queue**, choose the game session queue that you want to use with the matchmaker.

6. Choose **Next**.
7. On the **Configure settings** page, under **Matchmaking settings**, do the following:

   a. For **Request timeout**, set the maximum amount of time, in seconds, for the matchmaker to complete a match for each request. Matchmaking requests that exceed this time are rejected.

   b. For **Backfill mode**, choose a mode for handling match backfills. To turn on the automatic backfill feature, choose **Automatic**. Or, if you're managing backfill requests in your game server or game client, or if you opt not to backfill your games, choose **Manual**.

   c. (Optional) For **Additional player count**, set the number of player slots to keep open in a match. FlexMatch can fill these slots with players in the future.

   d. (Optional) Under **Match acceptance options**, for **Acceptance required**, if you want to require each player in a proposed match to actively accept participation in the match, select **Required**. If you select this option, then for **Acceptance timeout**, set how long, in seconds, you want the matchmaker to wait for player acceptances before canceling the match.

8. (Optional) Under **Event notification settings**, do the following:

   a. (Optional) For **SNS topic**, choose an Amazon Simple Notification Service (Amazon SNS) topic for receiving matchmaking event notifications. If you haven't yet set up an SNS topic, you can choose this later by editing the matchmaking configuration. For more information, see Set up FlexMatch event notifications (p. 40).

   b. (Optional) For **Custom event data**, enter any custom data that you want to associate with this matchmaker in event messaging. FlexMatch includes this data in every event associated with the matchmaker.

9. (Optional) Expand **Additional game data**, and then do the following:

   a. (Optional) For **Game session data**, enter any additional game-related information that you want FlexMatch to deliver to new game sessions started with matches made using this matchmaking configuration.

   b. (Optional) For **Game properties**, add key-value pair properties that contain information about a new game session.

10. (Optional) Under **Tags**, add tags to help you manage and track your AWS resources.
11. Choose **Next**.
12. On the **Review and create** page, review your choices, and then choose **Create**. Upon successful creation, the matchmaker is ready to accept matchmaking requests.

AWS CLI

To create a matchmaking configuration with the AWS CLI, open a command line window and use the create-matchmaking-configuration command to define a new matchmaker.

This example command creates a new matchmaking configuration that requires player acceptance and enables automatic backfill. It also reserves two player slots for FlexMatch to add players later, and it provides some game session data.

```
aws gamelift create-matchmaking-configuration \
    --name "SampleMatchamker123" \
    --description "The sample test matchmaker with acceptance" \
    --flex-match-mode WITH_QUEUE \
    --game-session-queue-arns "arn:aws:gamelift:us-
west-2:111122223333:gamesessionqueue/MyGameSessionQueue" \
    --rule-set-name "MyRuleSet" \
    --request-timeout-seconds 120 \
    --acceptance-required \
```

```
    --acceptance-timeout-seconds 30 \
    --backfill-mode AUTOMATIC \
    --notification-target "arn:aws:sns:us-west-2:111122223333:My_Matchmaking_SNS_Topic"
\
    --additional-player-count 2 \
    --game-session-data "key=map,value=winter444"
```

If the matchmaking configuration creation request is successful, GameLift returns a MatchmakingConfiguration object with the settings that you requested for the matchmaker. The new matchmaker is ready to accept matchmaking requests.

Create a matchmaker for standalone FlexMatch

Before creating a matchmaking configuration, create a rule set (p. 23) to use with the matchmaker.

Console

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/home.
2. Switch to the AWS Region where you want to create your matchmaker. For a list of Regions that support FlexMatch matchmaking configurations, see Choose an AWS Region for the matchmaker (p. 10).
3. In the navigation pane, choose **FlexMatch**, **Matchmaking configurations**.
4. On the **Matchmaking configurations** page, choose **Create configuration**.
5. On the **Define configuration details** page, under **Matchmaking configuration details**, do the following:
   a. For **Name**, enter a matchmaker name that can help you identify it in a list and in metrics. The matchmaker name must be unique within the Region. Matchmaking requests identify which matchmaker to use by its name and Region.
   b. (Optional) For **Description**, add a description to help identify the matchmaker.
   c. For **Rule set**, choose a rule set from the list to use with the matchmaker. The list contains all rule sets that you've created in the current Region.
   d. For **FlexMatch mode**, choose **Standalone**. This indicates that you have a custom mechanism for starting new game sessions on a hosting solution outside of GameLift.
6. Choose **Next**.
7. On the **Configure settings** page, under **Matchmaking settings**, do the following:
   a. For **Request timeout**, set the maximum amount of time, in seconds, for the matchmaker to complete a match for each request. Matchmaking requests that exceed this time are rejected.
   b. (Optional) Under **Match acceptance options**, for **Acceptance required**, if you want to require each player in a proposed match to actively accept participation in the match, select **Required**. If you select this option, then for **Acceptance timeout**, set how long, in seconds, you want the matchmaker to wait for player acceptances before canceling the match.
8. (Optional) Under **Event notification settings**, do the following:
   a. (Optional) For **SNS topic**, choose an Amazon SNS topic for receiving matchmaking event notifications. If you haven't yet set up an SNS topic, you can choose this later by editing the matchmaking configuration. For more information, see Set up FlexMatch event notifications (p. 40).
   b. (Optional) For **Custom event data**, enter any custom data that you want to associate with this matchmaker in event messaging. FlexMatch includes this data in every event associated with the matchmaker.
9. (Optional) Under **Tags**, add tags to help you manage and track your AWS resources.

10. Choose **Next**.
11. On the **Review and create** page, review your choices, and then choose **Create**. Upon successful creation, the matchmaker is ready to accept matchmaking requests.

AWS CLI

To create a matchmaking configuration with the AWS CLI, open a command line window and use the create-matchmaking-configuration command to define a new matchmaker.

This example command creates a new matchmaking configuration for a standalone matchmaker that requires player acceptance.

```
aws gamelift create-matchmaking-configuration \
    --name "SampleMatchamker123" \
    --description "The sample test matchmaker with acceptance" \
    --flex-match-mode STANDALONE \
    --rule-set-name "MyRuleSetOne" \
    --request-timeout-seconds 120 \
    --acceptance-required \
    --acceptance-timeout-seconds 30 \
    --notification-target "arn:aws:sns:us-west-2:111122223333:My_Matchmaking_SNS_Topic"
```

If the matchmaking configuration creation request is successful, GameLift returns a MatchmakingConfiguration object with the settings that you requested for the matchmaker. The new matchmaker is ready to accept matchmaking requests.

Build a FlexMatch rule set

Every FlexMatch matchmaker must have a rule set. The rule set determines the two key elements of a match: your game's team structure and size, and how to group players together for the best possible match.

For example, a rule set might describe a match like this: Create a match with two teams of five players each, one team is the defenders and the other team the invaders. A team can have novice and experienced players, but the average skill of the two teams must be within 10 points of each other. If no match is made after 30 seconds, gradually relax the skill requirements.

The topics in this section describe how design and build a matchmaking rule set. When creating a rule set, you can use either the Amazon GameLift console or the AWS CLI.

Topics

Design a FlexMatch rule set

This topic covers the basic structure of a rule set and how to build a rule set for small matches (up to 40 players). A matchmaking rule set must do two things: lay out a match's team structure and size and tell the matchmaker how to choose players to form the best possible match.

But your matchmaking rule set can do more. For example, you can:

- Optimize the matchmaking algorithm for your game.
- Set up minimum player latency requirements to protect the quality of gameplay.
- Gradually relax team requirements and match rules over time so all active players can find an acceptable match when they want one.
- Define handling for group matchmaking requests (party aggregation).
- Process large matches (>40 players). Learn more about building large matches in Design a FlexMatch large-match rule set (p. 20).

When building a matchmaking rule set, you can do some or all of the following tasks:

- Describe the rule set (p. 16) (required)
- Customize the match algorithm (p. 16) (optional)
- Declare player attributes (p. 19)
- Define match teams (p. 19)
- Set rules for player matching (p. 20)
- Allow requirements to relax over time (p. 20)

Describe the rule set

Provide details for the rule set.

- *name* (optional) – A descriptive label for your own use. This value isn't associated with the rule set name that you specify when creating the rule set with GameLift.
- *ruleLanguageVersion* (required) – The version of the property expression language used to create FlexMatch rules. The value must be `1.0`.

Customize the match algorithm

You can change the default matching algorithm. The default algorithm is optimized for most games to get players into acceptable matches with minimal wait time. You can customize the algorithm and adjust matchmaking for your game.

The default FlexMatch matchmaking process is as follows:

1. FlexMatch places all open matchmaking tickets and backfill tickets in a ticket pool.
2. Tickets in the pool are randomly grouped into one or more batches for matching. As the ticket pool gets larger, FlexMatch forms additional batches to maintain optimal batch size.
3. Within each batch, FlexMatch sorts the tickets by age.
4. FlexMatch builds a match based on the oldest ticket of each batch.

To customize the match algorithm, add an `algorithm` component to your rule set schema. See FlexMatch rule set schema (p. 56) for the complete reference information.

Use the following optional customizations to impact different stages of your matchmaking process.

- Add pre-batch sorting (p. 17)
- Form batches based on batchDistance attributes

- Prioritize backfill tickets (p. 18)
- Favor older tickets with expansions (p. 18)

Add pre-batch sorting

You can configure FlexMatch to sort the ticket pool before forming batches. This type of customization is most effective with games with large tickets pools. Pre-batch sorting can help speed up the matchmaking process and increase player uniformity in defined characteristics.

Define Pre-batch sorting methods using the algorithm property `batchingPreference`. The default setting is `random`.

Options for customizing pre-batch sorting include:

- **Sort by player attributes.** Provide a list of player attributes to pre-sort the ticket pool. FlexMatch then creates batches with more uniformity in the sorted attributes. For example, if you pre-sort the ticket pool by player skill, FlexMatch batches tickets with similar skill levels together. If your rule set also contains match rules based on player skill, pre-batch sorting can improve matchmaking efficiency.

  To sort by player attributes, set `batchingPreference` to `sorted`, and set `sortByAttributes` to the list of player attributes. To use an attribute, declare the attribute in the `playerAttributes` component of the rule set.

  In the following example, FlexMatch sorts the ticket pool based on players' preferred game map and then by player skill. The resulting batches are more likely to contain similarly skilled players who want to use the same map.

```
"algorithm": {
    "batchingPreference": "sorted",
    "sortByAttributes": ["map", "player_skill"],
    "strategy": "exhaustiveSearch"
},
```

- **Sort by latency.** Prioritize based on matches with the lowest available latency or quickly creating matches with acceptable latency. This customization is useful for rule sets forming large matches (more than 40 players). The algorithm property `strategy` must be set to `balanced`, which limits the available types of rule statements. See Design a FlexMatch large-match rule set (p. 20).

  FlexMatch pre-sorts tickets based on reported latency data in one of the following ways:

  - *Get players into lowest latency Regions.* The ticket pool is pre-sorted by the Regions where players report their lowest latency values. FlexMatch then batches tickets with low latency in the same Regions, creating an overall better game play experience. It also reduces the number of tickets in each batch, so matchmaking can take longer. To use this customization, set `batchingPreference` to `fastestRegion`, as shown in the following example.

```
"algorithm": {
    "batchingPreference": "fastestRegion",
    "strategy": "balanced"
},
```

  - *Get players into acceptable latency matches quickly.* The ticket pool is pre-sorted by Regions where players report any acceptable latency value. This forms fewer batches containing more tickets that have acceptable latency in the same Regions. With more tickets in each batch, finding enough acceptable matches is easier and faster. To use this customization, set the property `batchingPreference` to `largestPopulation`, as shown in the following example.

```
"algorithm": {
```

```
    "batchingPreference": "largestPopulation",
    "strategy": "balanced"
},
```

Note
`largetPopulation` is the default setting for rule sets using the balanced strategy.

Prioritize backfill tickets

If your game implements auto-backfill or manual backfill, you can customize how FlexMatch processes matchmaking tickets based on request type (new match or backfill request). By default, FlexMatch treats both types of requests the same. You can determine if FlexMatch tries to fill backfill tickets first or if FlexMatch fills backfill ticket when new matches can't be made.

Backfill prioritization impacts how FlexMatch handles tickets after they have been batched. Only use backfill prioritization with rule sets that use the exhaustive search strategy. FlexMatch doesn't match multiple backfill tickets together.

To change prioritization for backfill tickets, set the property `backfillPriority` as follows:

- **Match backfill tickets first.** This option prompts FlexMatch to try to complete backfill tickets before creating new matches. This means that incoming players have a higher chance of joining an existing game. Set `backfillPriority` to `high`.

  If your game is using auto-backfill, use this as a best practice. Auto-backfill is most often used in games with short game session time frames and high player turnaround. Auto-backfill helps these games to quickly form minimum viable matches and get them started even as FlexMatch searches for more players to fill open slots.

```
"algorithm": {
    "backfillPriority": "high",
    "strategy": "exhaustiveSearch"
},
```

- **Match backfill tickets last.** This option prompts FlexMatch to ignore backfill tickets until it evaluates all other tickets. This means that FlexMatch backfills incoming players into existing games when it can't match them into new games. Set `backfillPriority` to `low`.

  This option is useful when you want to use backfill as a last-chance option to get players into a game, such as when there are too few players to form a new match. Don't de-prioritize backfill tickets with auto-backfill.

```
"algorithm": {
    "backfillPriority": "low",
    "strategy": "exhaustiveSearch"
},
```

Favor older tickets with expansions

You can customize how FlexMatch applies expansion rules, which relax match criteria when matches are difficult to complete. GameLift applies expansion rules when tickets in a partially completed match reach a certain age. The creation timestamps of the tickets determine when GameLift applies the rules; by default, FlexMatch tracks the timestamp of the most recently matched ticket.

To change when FlexMatch applies expansion rules, set the property `expansionAgeSelection` as follows:

- **Expand based on newest tickets.** This option applies expansion rules based on the newest ticket added to the potential match. Each time FlexMatch matches a new ticket, the time clock is reset. With this option, resulting matches tend to be higher quality but take longer to match; match requests might time out before completing if they take too long to match. Set `expansionAgeSelection` to `newest`. `newest` is default.
- **Expand based on oldest tickets.** This option applies expansion rules based on the oldest ticket in the potential match. With this option, FlexMatch applies expansions faster, which improves wait times for the earliest matched players, but lowers the match quality for all players. Set `expansionAgeSelection` to `oldest`.

```
"algorithm": {
    "expansionAgeSelection": "oldest",
    "strategy": "exhaustiveSearch"
},
```

Declare player attributes

In this section, list individual player attributes to include in matchmaking requests. There are two reasons you might declare player attributes in a rule set:

- When the rule set contains rules that rely on player attributes.
- When you want to pass a player attribute to the game session through the match request. For example, you might want to pass player character choices to the game session before each player connects.

When declaring a player attribute, include the following information:

- *name* (required) – This value must be unique to the rule set.
- *type* (required) – The data type of the attribute value. Valid data types are number, string, or string map.
- *default* (optional) – Enter a default value to use if a matchmaking request doesn't provide an attribute value. If no default is declared and a request doesn't include a value, FlexMatch can't fulfill the request.

Define match teams

Describe the structure and size of the teams for a match. Each match must have at least one team, and you can define as many teams as you want. Your teams can have the same number of players or be asymmetric. For example, you might define a single-player monster team and a hunters team with 10 players.

FlexMatch processes match requests as either small match or large match, based on how the rule set defines team sizes. Potential matches of up to 40 players are small matches, matches with more than 40 players are large matches. To determine a rule set's potential match size, add up the *maxPlayer* settings for all teams defined in the rule set.

- *name* (required) – Assign each team a unique name. You use this name in rules and expansions, and FlexMatch references for the matchmaking data in a game session.
- *maxPlayers* (required) – Specify the maximum number of players to assign to the team.
- *minPlayers* (required) – Specify the minimum number of players to assign to the team.
- *quantity* (optional) – Specify the number of team to make with this definition. When FlexMatch creates a match, it gives these teams the provided name with an appended number. For example `Red-Team1`, `Red-Team2`, and `Red-Team3`.

FlexMatch attempts to fill teams to the maximum player size but does create teams with fewer players. If you want all teams in the match to be equally sized, you can create a rule for that. See the FlexMatch rule set examples (p. 25) topic for an example of an `EqualTeamSizes` rule.

Set rules for player matching

Create a set of rule statements that evaluate players for acceptance in to a match. Rules might set requirements that apply to individual players, teams, or an entire match. When GameLift processes a match request, it starts with the oldest player in the pool of available players and builds a match around that player. For detailed help on creating FlexMatch rules, see FlexMatch rule types (p. 63).

- *name* (required) – A meaningful name that uniquely identifies the rule within a rule set. Rule names are also referenced in event logs and metrics that track activity related to this rule.
- *description* (optional) – Use this element to attach a free-form text description.
- *type* (required) – The type element identifies the operation to use when processing the rule. Each rule type requires a set of additional properties. See a list of valid rule types and properties in FlexMatch rules language (p. 56).
- Rule type property (may be required) – Depending on the type of rule defined, you may need to set certain rule properties. Learn more about properties and how to use the FlexMatch property expression language in FlexMatch rules language (p. 56).

Allow requirements to relax over time

Expansions allow you to relax rule criteria over time when FlexMatch can't find a match. This feature ensures that FlexMatch makes a best available when it can't make a perfect match. By relaxing your rules with an expansion, you gradually expand the pool of players that are an acceptable match.

Expansions start when the age of the newest ticket in the incomplete match matches an expansion wait time. When FlexMatch adds a new ticket to the match, the expansion wait time clock may be reset. You can customize how expansions start in the `algorithm` section of the rule set.

Here's an example of an expansion that gradually increases the minimum skill level required for the match. The rule set uses a distance rule statement, named *SkillDelta* to require that all players in a match be within 5 skill levels of each other. If no new matches are made for fifteen seconds, this expansion looks for a skill level difference of 10, and then ten seconds later looks for a difference of 20.

```
"expansions": [{
        "target": "rules[SkillDelta].maxDistance",
        "steps": [{
            "waitTimeSeconds": 15,
            "value": 10
        }, {
            "waitTimeSeconds": 25,
            "value": 20
        }]
    }]
```

With matchmakers that have automatic backfill enabled, don't relax your player count requirements too quickly. It takes a few seconds for the new game session to start up and begin automatic backfill. A better approach is to start your expansion after automatic backfill tends to kicks in for your games. Expansion timing varies depending on your team composition, so do testing to find the best expansion strategy for your game.

Design a FlexMatch large-match rule set

If your rule set creates matches that allow 41 to 200 players, you need to make some adjustments to your rule set configuration. These adjustments optimize the match algorithm so that it can build viable

large matches while also keeping player wait times short. As a result, large match rule sets replace time-consuming custom rules with standard solutions that are optimized for common matchmaking priorities.

Here's how to determine if you need to optimize your rule set for large matches:

1. For each team defined in your rule set, get the value of *maxPlayer*,
2. Add up all the *maxPlayer* values. If the total exceeds 40, you've got a large match rule set.

To optimize your rule set for large matches, make the adjustments described as follows. See the schema for a large match rule set in Rule set schema for large matches (p. 58) and rule set examples in Example 7: Create a large match (p. 34).

Customize match algorithm for large matches

Add an algorithm component to the rule set, if one doesn't already exist. Set the following properties.

- `strategy` (required) – Set the `strategy` property to “balanced”. This setting triggers FlexMatch to do additional post-match checks to find the optimal team balance based on a specified player attribute, which is defined in the `balancedAttribute` property. The balanced strategy replaces the need for custom rules to build evenly matched teams.
- `balancedAttribute` (required) – Identify a player attribute to use when balancing the teams in a match. This attribute must have a numerical data type (double or integer). For example, if you choose to balance on player skill, FlexMatch tries to assign players so that all teams have aggregate skill levels that are as evenly matched as possible. The balancing attribute must be declared in the rule set's player attributes.
- `batchingPreference` (optional) – Choose how much emphasis you want to put on forming the lowest latency matches possible for your players. This setting affects how match tickets are sorted prior to building matches. Options include:
  - Largest population. FlexMatch allows matches using all tickets in the pool that have acceptable latency values in at least one Region in common. As a result, the potential ticket pool tends to be large, which makes it easier to fill matches more quickly. Players might be placed in games with acceptable, but not always optimal, latency. If the `batchingPreference` property isn't set, this is the default behavior when `strategy` is set to "balanced".
  - Fastest region. FlexMatch pre-sorts all tickets in the pool based on where they report the lowest latency values. As a result, matches tend to be formed with players that report low latency in the same Regions. At the same time, the potential ticket pool for each match is smaller, which can increase the time needed to fill a match. In addition, because a higher priority is placed on latency, players in matches may vary more widely with regard to the balancing attribute.

The following example configures the match algorithm to behave as follows: (1) Pre-sort the ticket pool to group tickets by Region where they have acceptable latency values; (2) Form batches of sorted tickets for matching; (3) Create matches with tickets in a batch and balance the teams to even out the average player skill.

```
"algorithm": {
    "strategy": "balanced",
    "balancedAttribute": "player_skill",
    "batchingPreference": "largestPopulation"
},
```

Declare player attributes

Make sure that you declare the player attribute that is used as a balancing attribute in the rule set algorithm. This attribute should be included for each player in a matchmaking request. You can provide a default value for the player attribute, but attribute balancing works best when player-specific values are provided.

Define teams

The process of defining team size and structure is the same as with small matches, but the way FlexMatch fills the teams is different. This affects how matches are likely to look like when only partially filled. You may want to adjust your minimum team sizes in response.

FlexMatch uses the following rules when assigning a player to a team. First: look for teams that haven't yet reached their minimum player requirement. Second: of those teams, find the one with the most open slots.

For matches that define multiple equally sized teams, players are added sequentially to each team until full. As a result, teams in a match always have a nearly equal number of players, even when the match is not full. There is currently no way to force equally sized teams in large matches. For matches with asymmetrically sized teams, the process is a bit more complex. In this scenario, players are initially assigned to the largest teams that have the most open slots. As the number of open slots become more evenly distributed across all teams, players are slotted into the smaller teams.

For example, let's say you have a rule set with three teams. The Red and Blue teams are both set to `maxPlayers=10`, `minPlayers=5`. The Green team is set to `maxPlayers=3`, `minPlayers=2`. Here's the fill sequence:

1. No team has reached `minPlayers`. Red and Blue teams have 10 open slots, while Green has 3. The first 10 players are assigned (5 each) to the Red and Blue teams. Both teams have now reached `minPlayers`.
2. Green team has not yet reached `minPlayers`. The next 2 players are assigned to the Green team. The Green team has now reached `minPlayers`.
3. With all teams at `minPlayers`, additional players are now assigned based on the number of open slots. The Red and Blue teams each have 5 open slots, while the Green team has 1. The next 8 players are assigned (4 each) to the Red and Blue teams. All teams now have 1 open slot.
4. The remaining 3 player slots are assigned (1 each) to teams in no particular order.

Set rules for large matches

Matchmaking for large matches relies primarily on the balancing strategy and latency batching optimizations. Most custom rules are not available. However, you can incorporate the following types of rules:

- Rule that sets a hard limit on player latency. Use the `latency` rule type with the property `maxLatency`. See Latency rule (p. 66) reference. Here's an example that sets maximum player latency to 200 milliseconds:

```
"rules": [{
       "name": "player-latency",
       "type": "latency",
       "maxLatency": 200
   }],
```

- Rule to batch players based on closeness in a specified player attribute. This is different than defining a balancing attribute as part of the large-match algorithm, which focuses on building evenly matched teams. This rule batches matchmaking tickets based on similarity in the specified attribute values, such as beginner or expert skill, which tends to lead to matches players who are closely aligned on the specified attribute. Use the `batchDistance` rule type, identify a numerically-based attribute, and specify the widest range to allow. See Batch distance rule (p. 63) reference. Here's an example that calls for a match's players to be within one skill level of each other:

```
"rules": [{
```

```
        "name": "batch-skill",
        "type": "batchDistance",
        "batchAttribute": "skill",
        "maxDistance": 1
```

Relax large match requirements

As with small matches, you can use expansions to relax match requirements over time when no valid matches are possible. With large matches, you have the option to relax either the latency rules or the team player counts.

If you're using automatic match backfill for large matches, avoid relaxing your team player counts too quickly. FlexMatch starts generating backfill requests only after a game session starts, which may not happen for several seconds after a match is created. During that time, FlexMatch can create multiple partially filled new game sessions, especially when the player count rules are lowered. As a result, you end up with more game sessions than you need and players spread too thinly across them. Best practice is to give the first step in your player count expansion a longer wait time, long enough for your game session to start. Since backfill requests are given higher priority with large matches, incoming players will be slotted into existing games before new game are started. You may need to experiment to find the ideal wait time for your game.

Here's an example that gradually lowers the Yellow team's player count, with a longer initial wait time. Keep in mind that wait times in rule set expansions are absolute, not compounded. So the first expansion occurs at five seconds, and the second expansion occurs five seconds later, at ten seconds.

```
"expansions": [{
        "target": "teams[Yellow].minPlayers",
        "steps": [{
            "waitTimeSeconds": 5,
            "value": 8
        }, {
            "waitTimeSeconds": 10,
            "value": 5
        }]
    }]
```

Create matchmaking rule sets

Before you create a matchmaking rule set for your GameLift FlexMatch matchmaker, we recommend checking the rule set syntax (p. 56). After you create a rule set using the GameLift console or the AWS Command Line Interface (AWS CLI), you can't change it.

Note that there is a service quota for the maximum number of rule sets that you can have in an AWS Region, so it's a good idea to delete unused rule sets.

Related topics

- Design a FlexMatch rule set (p. 15)
- FlexMatch rule set examples (p. 25)
- FlexMatch rules language (p. 56)

Console

Create a rule set

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.

2. Switch to the AWS Region where you want to create your rule set. Define rule sets in the same Region as the matchmaking configuration that uses them.
3. In the navigation pane, choose **FlexMatch**, **Matchmaking rule sets**.
4. On the **Matchmaking rule sets** page, choose **Create rule set**.
5. On the **Create matchmaking rule set** page, do the following:

   a. Under **Rule set settings**, for **Name**, enter a unique descriptive name that you can use to identify it in a list or in events and metrics tables.
   b. For **Rule set**, enter your rule set in JSON. For information about designing a rule set, see Design a FlexMatch rule set (p. 15). You can also use one of the example rule sets from FlexMatch rule set examples (p. 25).
   c. Choose **Validate** to verify that the syntax of your rule set is correct. You can't edit rule sets after they're created, so it's a good idea to validate them first.
   d. (Optional) Under **Tags**, add tags to help you manage and track your AWS resources.
6. Choose **Create**. If creation is successful, you can use the rule set with a matchmaker.

AWS CLI

Create a rule set

Open a command line window and use the command create-matchmaking-rule-set.

This example command creates a simple matchmaking rule set that sets up a single team. Be sure to create the rule set in the same AWS Region as the matchmaking configurations that uses it.

```
aws gamelift create-matchmaking-rule-set \
    --name "SampleRuleSet123" \
    --rule-set-body '{"name": "aliens_vs_cowboys", "ruleLanguageVersion": "1.0",
 "teams": [{"name": "cowboys", "maxPlayers": 8, "minPlayers":  4}]}'
```

If the creation request is successful, GameLift returns a MatchmakingRuleSet object that includes the settings that you specified. A matchmaker can now use the new rule set.

Console

Delete a rule set

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. Switch to the Region that you created the rule set in.
3. In the navigation pane, choose **FlexMatch**, **Matchmaking rule sets**.
4. On the **Matchmaking rule sets** page, select the rule set that you want to delete, and then choose **Delete**.
5. In the **Delete rule set** dialog box, choose **Delete** to confirm deletion.

   Note
   If a matchmaking configuration is using the rule set, GameLift displays an error message (**Can't delete rule set**). If this occurs, change the matchmaking configuration to use a different rule set, then try again. To find out which matchmaking configurations are using a rule set, choose the name of a rule set to view its details page.

AWS CLI

Delete a rule set

Open a command line window and use the command delete-matchmaking-rule-set to delete a matchmaking rule set.

If a matchmaking configuration is using the rule set, GameLift returns an error message. If this occurs, change the matchmaking configuration to use a different rule set, then try again. To get a list of which matchmaking configurations are using a rule set, use the command describe-matchmaking-configurations and specify the rule set name.

This example command checks for the matchmaking rule set's usage and then deletes the rule set.

```
aws gamelift describe-matchmaking-configurations \
    --rule-set-name "SampleRuleSet123" \
    --limit 10

aws gamelift delete-matchmaking-rule-set \
    --name  "SampleRuleSet123"
```

FlexMatch rule set examples

FlexMatch rule sets can cover a variety of matchmaking scenarios. The following examples conform to the FlexMatch configuration structure and property expression language. Copy these rule sets in their entirety or choose components as needed.

For more information on using FlexMatch rules and rule sets, see the following topics:

- Build a FlexMatch rule set (p. 15)
- Design a FlexMatch rule set (p. 15)
- FlexMatch rule set schema (p. 56)
- FlexMatch rules language (p. 56)

Note
When evaluating a matchmaking ticket that includes multiple players, all players in the request must meet the match requirements.

Example 1: Create two teams with evenly matched players

This example illustrates how to set up two equally matched teams of players with the following instructions.

- Create two teams of players.
  - Include between four and eight players in each team.
  - Final teams must have the same number of players.
- Include a player's skill level (if not provided, default to 10).
- Choose players based on whether their skill level is similar to other players. Ensure that both teams have an average player skill within 10 points of each other.
- If the match is not filled quickly, relax the player skill requirement to complete a match in reasonable time.
  - After 5 seconds, expand the search to allow teams with average player skills within 50 points.
  - After 15 seconds, expand the search to allow teams with average player skills within 100 points.

Notes on using this rule set:

- This example allows for teams to be any size between four and eight players (although they must be the same size). For teams with a range of valid sizes, the matchmaker makes a best-effort attempt to match the maximum number of allowed players.
- The `FairTeamSkill` rule ensures that teams are evenly matched based on player skill. To evaluate this rule for each new prospective player, FlexMatch tentatively adds the player to a team and calculates the averages. If rule fails, the prospective player is not added to the match.
- Since both teams have identical structures, you could opt to create just one team definition and set the team quantity to "2". In this scenario, if you named the team "aliens", then your teams would be assigned the names "aliens_1" and "aliens_2".

```
{
    "name": "aliens_vs_cowboys",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "skill",
        "type": "number",
        "default": 10
    }],
    "teams": [{
        "name": "cowboys",
        "maxPlayers": 8,
        "minPlayers": 4
    }, {
        "name": "aliens",
        "maxPlayers": 8,
        "minPlayers": 4
    }],
    "rules": [{
        "name": "FairTeamSkill",
        "description": "The average skill of players in each team is within 10 points from
 the average skill of all players in the match",
        "type": "distance",
        // get skill values for players in each team and average separately to produce list
 of two numbers
        "measurements": [ "avg(teams[*].players.attributes[skill])" ],
        // get skill values for players in each team, flatten into a single list, and
 average to produce an overall average
        "referenceValue": "avg(flatten(teams[*].players.attributes[skill]))",
        "maxDistance": 10 // minDistance would achieve the opposite result
    }, {
        "name": "EqualTeamSizes",
        "description": "Only launch a game when the number of players in each team matches,
 e.g. 4v4, 5v5, 6v6, 7v7, 8v8",
        "type": "comparison",
        "measurements": [ "count(teams[cowboys].players)" ],
        "referenceValue": "count(teams[aliens].players)",
        "operation": "=" // other operations: !=, <, <=, >, >=
    }],
    "expansions": [{
        "target": "rules[FairTeamSkill].maxDistance",
        "steps": [{
            "waitTimeSeconds": 5,
            "value": 50
        }, {
            "waitTimeSeconds": 15,
            "value": 100
        }]
    }]
}
```

Example 2: Create uneven teams (Hunters vs. Monster)

This example describes a game mode in which a group of players hunt a single monster. People choose either a hunter or a monster role. Hunters specify the minimum skill level for the monster that they want to face. The minimum size of the hunter team can be relaxed over time to complete the match. This scenario sets out the following instructions:

- Create one team of exactly five hunters.
- Create a separate team of exactly one monster.
- Include the following player attributes:
  - A player's skill level (if not provided, default to 10).
  - A player's preferred monster skill level (if not provided, default to 10).
  - Whether the player wants to be the monster (if not provided, default to 0 or false).
- Choose a player to be the monster based on the following criteria:
  - Player must request the monster role.
  - Player must meet or exceed the highest skill level preferred by the players who are already added to the hunter team.
- Choose players for the hunter team based on the following criteria:
  - Players who request a monster role cannot join the hunter team.
  - If the monster role is already filled, player must want a monster skill level that is lower than the skill of the proposed monster.
- If a match is not filled quickly, relax the hunter team's minimum size as follows:
  - After 30 seconds, allow a game to start with only four players in the hunter team.
  - After 60 seconds, allow a game to start with only three people in the hunter team.

Notes on using this rule set:

- By using two separate teams for hunters and monster, you can evaluate membership based on different sets of criteria.

```
{
    "name": "players_vs_monster_5_vs_1",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "skill",
        "type": "number",
        "default": 10
    },{
        "name": "desiredSkillOfMonster",
        "type": "number",
        "default": 10
    },{
        "name": "wantsToBeMonster",
        "type": "number",
        "default": 0
    }],
    "teams": [{
        "name": "players",
        "maxPlayers": 5,
        "minPlayers": 5
    }, {
        "name": "monster",
        "maxPlayers": 1,
        "minPlayers": 1
```

```
    }],
    "rules": [{
        "name": "MonsterSelection",
        "description": "Only users that request playing as monster are assigned to the
 monster team",
        "type": "comparison",
        "measurements": ["teams[monster].players.attributes[wantsToBeMonster]"],
        "referenceValue": 1,
        "operation": "="
    },{
        "name": "PlayerSelection",
        "description": "Do not place people who want to be monsters in the players team",
        "type": "comparison",
        "measurements": ["teams[players].players.attributes[wantsToBeMonster]"],
        "referenceValue": 0,
        "operation": "="
    },{
        "name": "MonsterSkill",
        "description": "Monsters must meet the skill requested by all players",
        "type": "comparison",
        "measurements": ["avg(teams[monster].players.attributes[skill])"],
        "referenceValue": "max(teams[players].players.attributes[desiredSkillOfMonster])",
        "operation": ">="
    }],
    "expansions": [{
        "target": "teams[players].minPlayers",
        "steps": [{
            "waitTimeSeconds": 30,
            "value": 4
        },{
            "waitTimeSeconds": 60,
            "value": 3
        }]
    }]
}
```

Example 3: Set team-level requirements and latency limits

This example illustrates how to set up player teams and apply a set of rules to each team instead of each individual player. It uses a single definition to create three equally matched teams. It also establishes a maximum latency for all players. Latency maximums can be relaxed over time to complete the match. This scenario sets out the following instructions:

- Create three teams of players.
  - Include between three and five players in each team.
  - Final teams must contain the same or nearly the same number of players (within one).
- Include the following player attributes:
  - A player's skill level (if not provided, default to 10).
  - A player's character role (if not provided, default to "peasant").
- Choose players based on whether their skill level is similar to other players in the match.
  - Ensure that each team has an average player skill within 10 points of each other.
- Limit teams to the following number of "medic" characters:
  - An entire match can have a maximum of five medics.
- Only match players who report latency of 50 milliseconds or less.
- If a match is not filled quickly, relax the player latency requirement as follows:
  - After 10 seconds, allow player latency values up to 100 ms.
  - After 20 seconds, allow player latency values up to 150 ms.

Notes on using this rule set:

- The rule set ensures that teams are evenly matched based on player skill. To evaluate the `FairTeamSkill` rule, FlexMatch tentatively adds the prospective player to a team and calculates the average skill of players in the team. It then compares it against the average skill of players in both teams. If rule fails, the prospective player is not added to the match.
- The team- and match-level requirements (total number of medics) are achieved through a collection rule. This rule type takes a list of character attributes for all players and checks against the maximum counts. Use `flatten` to create a list for all players in all teams.
- When evaluating based on latency, note the following:
  - Latency data is provided in the matchmaking request as part of the Player object. It is not a player attribute, so it does not need to be listed as one.
  - The matchmaker evaluates latency by region. Any region with a latency higher than the maximum is ignored. To be accepted for a match, a player must have at least one region with a latency below the maximum.
  - If a matchmaking request omits latency data one or more players, the request is rejected for all matches.

```
{
    "name": "three_team_game",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "skill",
        "type": "number",
        "default": 10
    },{
        "name": "character",
        "type": "string_list",
        "default": [ "peasant" ]
    }],
    "teams": [{
        "name": "trio",
        "minPlayers": 3,
        "maxPlayers": 5,
        "quantity": 3
    }],
    "rules": [{
        "name": "FairTeamSkill",
        "description": "The average skill of players in each team is within 10 points from
 the average skill of players in the match",
        "type": "distance",
        // get players for each team, and average separately to produce list of 3
        "measurements": [ "avg(teams[*].players.attributes[skill])" ],
        // get players for each team, flatten into a single list, and average to produce
 overall average
        "referenceValue": "avg(flatten(teams[*].players.attributes[skill]))",
        "maxDistance": 10 // minDistance would achieve the opposite result
    }, {
        "name": "CloseTeamSizes",
        "description": "Only launch a game when the team sizes are within 1 of each other.
 e.g. 3 v 3 v 4 is okay, but not 3 v 5 v 5",
        "type": "distance",
        "measurements": [ "max(count(teams[*].players))"],
        "referenceValue": "min(count(teams[*].players))",
        "maxDistance": 1
    }, {
        "name": "OverallMedicLimit",
        "description": "Don't allow more than 5 medics in the game",
        "type": "collection",
        // This is similar to above, but the flatten flattens everything into a single
```

```
        // list of characters in the game.
        "measurements": [ "flatten(teams[*].players.attributes[character])"],
        "operation": "contains",
        "referenceValue": "medic",
        "maxCount": 5
    }, {
        "name": "FastConnection",
        "description": "Prefer matches with fast player connections first",
        "type": "latency",
        "maxLatency": 50
    }],
    "expansions": [{
        "target": "rules[FastConnection].maxLatency",
        "steps": [{
            "waitTimeSeconds": 10,
            "value": 100
        }, {
            "waitTimeSeconds": 20,
            "value": 150
        }]
    }]
}
```

Example 4: Use explicit sorting to find best matches

This example sets up a simple match with two teams of three players. It illustrates how to use explicit sorting rules to help find the best possible matches as quickly as possible. These rules sort all active matchmaking tickets to create the best matches based on certain key requirements. This scenario is implemented with the following instructions:

- Create two teams of players.
- Include exactly three players in each team.
- Include the following player attributes:
  - Experience level (if not provided, default to 50).
  - Preferred game modes (can list multiple values) (if not provided, default to “coop” and “deathmatch”).
  - Preferred game maps, including map name and preference weighting (if not provided, default to `"defaultMap"` with a weight of 100).
- Set up presorting:
  - Sort players based on their preference for the same game map as the anchor player. Players can have multiple favorite game maps, so this example uses a preference value.
  - Sort players based on how closely their experience level matches the anchor player. With this sort, all players in all teams will have experience levels that are as close as possible.
- All players across all teams must have selected at least one game mode in common.
- All players across all teams must have selected at least one game map in common.

Notes on using this rule set:

- The game map sort uses an absolute sort that compares the mapPreference attribute value. Because it is first in the rule set, this sort is performed first.
- The experience sort uses a distance sort to compare a prospective player's skill level with the anchor player's skill.
- Sorts are performed in the order they are listed in a rule set. In this scenario, players are sorted by game map preference, and then by experience level.

```
{
    "name": "multi_map_game",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "experience",
        "type": "number",
        "default": 50
    }, {
        "name": "gameMode",
        "type": "string_list",
        "default": [ "deathmatch", "coop" ]
    }, {
        "name": "mapPreference",
        "type": "string_number_map",
        "default": { "defaultMap": 100 }
    }, {
        "name": "acceptableMaps",
        "type": "string_list",
        "default": [ "defaultMap" ]
    }],
    "teams": [{
        "name": "red",
        "maxPlayers": 3,
        "minPlayers": 3
    }, {
        "name": "blue",
        "maxPlayers": 3,
        "minPlayers": 3
    }],
    "rules": [{
        // We placed this rule first since we want to prioritize players preferring the
same map
        "name": "MapPreference",
        "description": "Favor grouping players that have the highest map preference aligned
with the anchor's favorite",
        // This rule is just for sorting potential matches.  We sort by the absolute value
of a field.
        "type": "absoluteSort",
        // Highest values go first
        "sortDirection": "descending",
        // Sort is based on the mapPreference attribute.
        "sortAttribute": "mapPreference",
        // We find the key in the anchor's mapPreference attribute that has the highest
value.
        // That's the key that we use for all players when sorting.
        "mapKey": "maxValue"
    }, {
        // This rule is second because any tie-breakers should be ordered by similar
experience values
        "name": "ExperienceAffinity",
        "description": "Favor players with similar experience",
        // This rule is just for sorting potential matches.  We sort by the distance from
the anchor.
        "type": "distanceSort",
        // Lowest distance goes first
        "sortDirection": "ascending",
        "sortAttribute": "experience"
    }, {
        "name": "SharedMode",
        "description": "The players must have at least one game mode in common",
        "type": "collection",
        "operation": "intersection",
        "measurements": [ "flatten(teams[*].players.attributes[gameMode])"],
        "minCount": 1
    }, {
```

```
        "name": "MapOverlap",
        "description": "The players must have at least one map in common",
        "type": "collection",
        "operation": "intersection",
        "measurements": [ "flatten(teams[*].players.attributes[acceptableMaps])"],
        "minCount": 1
    }]
}
```

Example 5: Find intersections across multiple player attributes

This example illustrates how to use a collection rule to find intersections in two or more player attributes. When working with collections, you can use the `intersection` operation for a single attribute, and the `reference_intersection_count` operation for multiple attributes.

To illustrate this approach, this example evaluates players in a match based on their character preferences. The example game is a "free-for-all" style in which all players in a match are opponents. Each player is asked to (1) choose a character for themselves, and (2) choose characters they want to play against. We need a rule that ensures that every player in a match is using a character that is on all other players' preferred opponents list.

The example rule set describes a match with the following characteristics:

- Team structure: One team of five players
- Player attributes:
  - *myCharacter*: The player's chosen character.
  - *preferredOpponents*: List of characters that the player wants to play against.
- Match rules: A potential match is acceptable if each character in use is on every player's preferred opponents list.

To implement the match rule, this example uses a collection rule with the following property values:

- Operation – Uses `reference_intersection_count` operation to evaluate how each string list in the measurement value intersects with the string list in the reference value.
- Measurement – Uses the `flatten` property expression to create a list of string lists, with each string list containing one player's *myCharacter* attribute value.
- Reference value – Uses the `set_intersection` property expression to create a string list of all *preferredOpponents* attribute values that are common to every player in the match.
- Restrictions – `minCount` is set to 1 to ensure that each player's chosen character (a string list in the measurement) matches at least one of the preferred opponents common to all players. (a string in the reference value).
- Expansion – If a match is not filled within 15 seconds, relax the minimum intersection requirement.

The process flow for this rule is as follows:

1. A player is added to the prospective match. The reference value (a string list) is recalculated to include intersections with the new player's preferred opponents list. The measurement value (a list of string lists) is recalculated to add the new player's chosen character as a new string list.
2. Amazon GameLift verifies that each string list in the measurement value (the players' chosen characters) intersects with at least one string in the reference value (the players' preferred opponents). Since in this example each string list in the measurement contains only one value, the intersection is either 0 or 1.
3. If any string list in the measurement does not intersect with the reference value string list, the rule fails and the new player is removed from the prospective match.

4. If a match is not filled within 15 seconds, drop the opponent match requirement to fill the remaining player slots in the match.

```
{
    "name": "preferred_characters",
    "ruleLanguageVersion": "1.0",

    "playerAttributes": [{
        "name": "myCharacter",
        "type": "string_list"
    }, {
        "name": "preferredOpponents",
        "type": "string_list"
    }],

    "teams": [{
        "name": "red",
        "minPlayers": 5,
        "maxPlayers": 5
    }],

    "rules": [{
        "description": "Make sure that all players in the match are using a character that
 is on all other players' preferred opponents list.",
        "name": "OpponentMatch",
        "type": "collection",
        "operation": "reference_intersection_count",
        "measurements": ["flatten(teams[*].players.attributes[myCharacter])"],
        "referenceValue":
 "set_intersection(flatten(teams[*].players.attributes[preferredOpponents]))",
        "minCount":1
    }],
    "expansions": [{
        "target": "rules[OpponentMatch].minCount",
        "steps": [{
            "waitTimeSeconds": 15,
            "value": 0
        }]
    }]
}
```

Example 6: Compare attributes across all players

This example illustrates how to compare player attributes across a group of players.

The example rule set describes a match with the following characteristics:

- Team structure: Two single-player teams
- Player attributes:
  - *gameMode*: Type of game chosen by the player (if not provided, default to "turn-based").
  - *gameMap*: Game world chosen by the player (if not provided, default to 1).
  - *character*: Character chosen by the player (no default value means that players must specify a character).
- Match rules: Matched players must meet the following requirements:
  - Players must choose the same game mode.
  - Players must choose the same game map.
  - Players much choose different characters.

Notes on using this rule set:

- To implement the match rule, this example uses comparison rules to check all players' attribute values. For game mode and map, the rule verifies that the values are the same. For character, the rule verifies that the values are different.
- This example uses one player definition with a quantity property to create both player teams. The team are assigned the following names: "player_1" and "player_2".

```
{
    "name": "",
    "ruleLanguageVersion": "1.0",

    "playerAttributes": [{
        "name": "gameMode",
        "type": "string",
        "default": "turn-based"
    }, {
        "name": "gameMap",
        "type": "number",
        "default": 1
    }, {
        "name": "character",
        "type": "number"
    }],

    "teams": [{
        "name": "player",
        "minPlayers": 1,
        "maxPlayers": 1,
        "quantity": 2
    }],

    "rules": [{
        "name": "SameGameMode",
        "description": "Only match players when they choose the same game type",
        "type": "comparison",
        "operation": "=",
        "measurements": ["flatten(teams[*].players.attributes[gameMode])"]
    }, {
        "name": "SameGameMap",
        "description": "Only match players when they're in the same map",
        "type": "comparison",
        "operation": "=",
        "measurements": ["flatten(teams[*].players.attributes[gameMap])"]
    }, {
        "name": "DifferentCharacter",
        "description": "Only match players when they're using different characters",
        "type": "comparison",
        "operation": "!=",
        "measurements": ["flatten(teams[*].players.attributes[character])"]
    }]
}
```

Example 7: Create a large match

This example illustrates how to set up a rule set for matches that can exceed 40 players. When a rule set describes teams with a total maxPlayer count greater than 40, it is processed as a large match. Learn more in Design a FlexMatch large-match rule set (p. 20).

The example rule set creates a match using the following instructions:

- Create one team with up to 200 players, with a minimum requirement of 175 players.
- Balancing criteria: Select players based on similar skill level. All players must report their skill level to be matched.
- Batching preference: Group players by similar balancing criteria when creating matches.
- Latency rules: Set the maximum acceptable player latency of 150 milliseconds.
- If the match is not filled quickly, relax the requirements to complete a match in reasonable time.
  - After 10 seconds, accept a team with 150 players.
  - After 12 seconds, raise the maximum acceptable latency to 200 milliseconds.
  - After 15 seconds, accept a team with 100 players.

Notes on using this rule set:

- Because the algorithm uses the "largest population" batching preference, players are first sorted based on the balancing criteria. As a result, matches tend to be fuller and contain players that are more similar in skill. All players meet acceptable latency requirements, but they may not get the best possible latency for their location.
- The algorithm strategy used in this rule set, "largest population", is the default setting. To use the default, you can opt to omit the setting.
- If you've enabled match backfill, do not relax player count requirements too quickly, or you may end up with too many partially filled game sessions. Learn more in Relax large match requirements (p. 23).

```
{
    "name": "free-for-all",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "skill",
        "type": "number"
    }],
    "algorithm": {
        "balancedAttribute": "skill",
        "strategy": "balanced",
        "batchingPreference": "largestPopulation"
    },
    "teams": [{
        "name": "Marauders",
        "maxPlayers": 200,
        "minPlayers": 175
    }],
    "rules": [{
        "name": "low-latency",
        "description": "Sets maximum acceptable latency",
        "type": "latency",
        "maxLatency": 150
    }],
    "expansions": [{
        "target": "rules[low-latency].maxLatency",
        "steps": [{
            "waitTimeSeconds": 12,
            "value": 200
        }],
        "target": "teams[Marauders].minPlayers",
        "steps": [{
            "waitTimeSeconds": 10,
            "value": 150
        }, {
            "waitTimeSeconds": 15,
```

```
            "value": 100
        }]
    }]
}
```

Example 8: Create a multi-team large match

This example illustrates how to set up a rule set for matches with multiple teams that can exceed 40 players. This example illustrates how to create multiple identical teams with one definition and how asymmetrically sized teams are filled during match creation.

The example rule set creates a match using the following instructions:

- Create ten identical "hunter" teams with up to 15 players, and one "monster" team with exactly 5 players.
- Balancing criteria: Select players based on number of monster kills. If players don't report their kill count, use a default value of 5.
- Batching preference: Group players based on the regions where they report the fastest player latency.
- Latency rule: Sets a maximum acceptable player latency of 200 milliseconds.
- If the match is not filled quickly, relax the requirements to complete a match in reasonable time.
  - After 15 seconds, accept teams with 10 players.
  - After 20 seconds, accept teams with 8 players.

Notes on using this rule set:

- This rule set defines teams that can potentially hold up to 155 players, which makes it a large match. (10 x 15 hunters + 5 monsters = 155)
- Because the algorithm uses the "fastest region" batching preference, players tend to be placed in regions where they report faster latency and not in regions where they report high (but acceptable) latency. At the same time, matches are likely to have fewer players, and the balancing criteria (number of monster skills) may vary more widely.
- When an expansion is defined for a multi-team definition (quantity > 1), the expansion applies to all teams created with that definition. So by relaxing the hunter team minimum players setting, all ten hunter teams are affected equally.
- Since this rule set is optimized to minimize player latency, the latency rule acts as a catch-all to exclude players who have no acceptable connection options. We don't need to relax this requirement.
- Here's how FlexMatch fills matches for this rule set before any expansions take effect:
  - No teams have reached minPlayers count yet. Hunter teams have 15 open slots, while Monster team has 5 open slots.
    - The first 100 players are assigned (10 each) to the ten hunter teams.
    - The next 22 players are assigned sequentially (2 each) to hunter teams and monster team.
  - Hunter teams have reached minPlayers count of 12 players each. Monster team has 2 players and has not reached minPlayers count.
    - The next three players are assigned to the monster team.
  - All teams have reached minPlayers count. Hunter teams each have three open slots. Monster team is full.
    - The final 30 players are assigned sequentially to the hunter teams, ensuring that all hunter teams have nearly the same size (plus or minus one player).
- If you've enabled backfill for matches created with this rule set, do not relax player count requirements too quickly, or you may end up with too many partially filled game sessions. Learn more in Relax large match requirements (p. 23).

```
{
    "name": "monster-hunters",
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": "monster-kills",
        "type": "number",
        "default": 5
    }],
    "algorithm": {
        "balancedAttribute": "monster-kills",
        "strategy": "balanced",
        "batchingPreference": "fastestRegion"
    },
    "teams": [{
        "name": "Monsters",
        "maxPlayers": 5,
        "minPlayers": 5
    }, {
        "name": "Hunters",
        "maxPlayers": 15,
        "minPlayers": 12,
        "quantity": 10
    }],
    "rules": [{
        "name": "latency-catchall",
        "description": "Sets maximum acceptable latency",
        "type": "latency",
        "maxLatency": 150
    }],
    "expansions": [{
        "target": "teams[Hunters].minPlayers",
        "steps": [{
            "waitTimeSeconds": 15,
            "value": 10
        }, {
            "waitTimeSeconds": 20,
            "value": 8
        }]
    }]
}
```

Example 9: Create a large match with players with similar attributes

This example illustrates how to set up a rule set for matches with two teams using `batchDistance`. In the example:

- The `SimilarLeague` rule ensures all players in a match have a `league` within 2 of other players.
- The `SimilarSkill` rule ensures all players in a match have a `skill` within 10 of other players. If a player has been waiting 10 seconds, the distance is expanded to 20. If a player has been waiting 20 seconds, the distance is expanded to 40.
- The `SameMap` rule ensures all players in a match have requested the same `map`.
- The `SameMode` rule ensures all players in a match have requested the same `mode`.

```
{
    "ruleLanguageVersion": "1.0",
    "teams": [{
        "name": "red",
        "minPlayers": 100,
```

```
        "maxPlayers": 100
    }, {
        "name": "blue",
        "minPlayers": 100,
        "maxPlayers": 100
    }],
    "algorithm": {
        "strategy":"balanced",
        "balancedAttribute": "skill",
        "batchingPreference":"fastestRegion"
    },
    "playerAttributes": [{
        "name": "league",
        "type": "number"
    },{
        "name": "skill",
        "type": "number"
    },{
        "name": "map",
        "type": "string"
    },{
        "name": "mode",
        "type": "string"
    }],
    "rules": [{
        "name": "SimilarLeague",
        "type": "batchDistance",
        "batchAttribute": "league",
        "maxDistance": 2
    }, {
        "name": "SimilarSkill",
        "type": "batchDistance",
        "batchAttribute": "skill",
        "maxDistance": 10
    }, {
        "name": "SameMap",
        "type": "batchDistance",
        "batchAttribute": "map"
    }, {
        "name": "SameMode",
        "type": "batchDistance",
        "batchAttribute": "mode"
    }],
    "expansions": [{
        "target": "rules[SimilarSkill].maxDistance",
        "steps": [{
            "waitTimeSeconds": 10,
            "value": 20
        }, {
            "waitTimeSeconds": 20,
            "value": 40
        }]
    }]
}
```

Example 10: Use a compound rule to create a match with players with similar attributes or similar selections

This example illustrates how to set up a rule set for matches with two teams using `compound`. In the example:

- The `SimilarLeagueDistance` rule ensures all players in a match have a `league` within 2 of other players.

- The `SimilarSkillDistance` rule ensures all players in a match have a `skill` within 10 of other players. If a player has been waiting 10 seconds, the distance is expanded to 20. If a player has been waiting 20 seconds, the distance is expanded to 40.
- The `SameMapComparison` rule ensures all players in a match have requested the same `map`.
- The `SameModeComparison` rule ensures all players in a match have requested the same `mode`.
- The `CompoundRuleMatchmaker` rule ensures a match if at least one of the following conditions is true:
  - Players in a match have requested the same `map` and the same `mode`.
  - Players in a match have comparable `skill` and `league` attributes.

```
{
    "ruleLanguageVersion": "1.0",
    "teams": [{
        "name": "red",
        "minPlayers": 10,
        "maxPlayers": 20
    }, {
        "name": "blue",
        "minPlayers": 10,
        "maxPlayers": 20
    }],
    "algorithm": {
        "strategy":"balanced",
        "balancedAttribute": "skill",
        "batchingPreference":"fastestRegion"
    },
    "playerAttributes": [{
        "name": "league",
        "type": "number"
    },{
        "name": "skill",
        "type": "number"
    },{
        "name": "map",
        "type": "string"
    },{
        "name": "mode",
        "type": "string"
    }],
    "rules": [{
        "name": "SimilarLeagueDistance",
        "type": "distance",
        "measurements": ["max(flatten(teams[*].players.attributes[league]))"],
        "referenceValue": "min(flatten(teams[*].players.attributes[league]))",
        "maxDistance": 2
    }, {
        "name": "SimilarSkillDistance",
        "type": "distance",
        "measurements": ["max(flatten(teams[*].players.attributes[skill]))"],
        "referenceValue": "min(flatten(teams[*].players.attributes[skill]))",
        "maxDistance": 10
    }, {
        "name": "SameMapComparison",
        "type": "comparison",
        "operation": "=",
        "measurements": ["flatten(teams[*].players.attributes[map])"]
    }, {
        "name": "SameModeComparison",
        "type": "comparison",
        "operation": "=",
        "measurements": ["flatten(teams[*].players.attributes[mode])"]
```

```
    }, {
        "name": "CompoundRuleMatchmaker",
        "type": "compound",
        "statement": "or(and(SameMapComparison, SameModeComparison),
 and(SimilarSkillDistance, SimilarLeagueDistance))"
    }],
    "expansions": [{
        "target": "rules[SimilarSkillDistance].maxDistance",
        "steps": [{
            "waitTimeSeconds": 10,
            "value": 20
        }, {
            "waitTimeSeconds": 20,
            "value": 40
        }]
    }]
}
```

Set up FlexMatch event notifications

If you're using GameLift FlexMatch matchmaking in your game, you need a way to track the status of individual matchmaking requests. Implementing event notifications is a fast and efficient method for tracking matchmaking events. All games in production, or in pre-production with high-volume matchmaking activity, should use event notifications.

There are two options for setting up event notifications. You can use Amazon CloudWatch Events, which has a suite of tools available for managing events and taking action on them. Or, you can set up your own Amazon Simple Notification Service (Amazon SNS) topics and configure your matchmaker to send matchmaking event notifications directly to the topics.

For a list of the FlexMatch events that GameLift emits, see FlexMatch matchmaking events (p. 70).

Set up CloudWatch Events

GameLift automatically posts all matchmaking events to CloudWatch Events. With CloudWatch Events, you can set up rules to have matchmaking events routed to a range of targets, including SNS topics and other AWS services for processing. For example, you might set a rule to route the event "PotentialMatchCreated" to an AWS Lambda function that handles player acceptances. For more information about how to use CloudWatch Events, including a collection of tutorials, see Getting Started with Amazon CloudWatch Events in the *Amazon CloudWatch Events User Guide*.

If you plan to use CloudWatch Events, when configuring your matchmakers, you can keep the notification target field empty, or reference an SNS topic if you want to use both options.

You can access GameLift matchmaking events in CloudWatch Events in the CloudWatch console. For more information, see Sign in to the Amazon CloudWatch Console. CloudWatch Events identifies each matchmaking event by the service (GameLift), the matchmaking name, and the matchmaking ticket.

Set up an Amazon SNS topic

You can have GameLift publish all events that a FlexMatch matchmaker generates to an Amazon SNS topic.

To create an SNS topic for GameLift event notifications

1. Open the Amazon SNS console.

2. In the navigation pane, choose **Topics**.
3. On the **Topics** page, choose **Create topic**.
4. Create a topic in the console. For more information, see To create a topic using the AWS Management Console in the *Amazon Simple Notification Service Developer Guide*.
5. On the **Details** page for your topic, choose **Edit**.
6. On the **Edit** page for your topic, expand **Access policy - *optional***, and then add the bold syntax from the following AWS Identity and Access Management (IAM) policy statement to the end of your existing policy. (The entire policy is shown here for clarity.) Be sure to use the Amazon Resource Name (ARN) details for your own SNS topic and GameLift matchmaking configuration.

```
{
  "Version": "2008-10-17",
  "Id": "__default_policy_ID",
  "Statement": [
    {
      "Sid": "__default_statement_ID",
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "SNS:GetTopicAttributes",
        "SNS:SetTopicAttributes",
        "SNS:AddPermission",
        "SNS:RemovePermission",
        "SNS:DeleteTopic",
        "SNS:Subscribe",
        "SNS:ListSubscriptionsByTopic",
        "SNS:Publish"
      ],
      "Resource": "arn:aws:sns:your_region:your_account:your_topic_name",
      "Condition": {
        "StringEquals": {
          "AWS:SourceAccount": "your_account"
        }
      }
    },
    {
      "Sid": "__console_pub_0",
      "Effect": "Allow",
      "Principal": {
        "Service": "gamelift.amazonaws.com"
      },
      "Action": "SNS:Publish",
      "Resource": "arn:aws:sns:your_region:your_account:your_topic_name",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn":
 "arn:aws:gamelift:your_region:your_account:matchmakingconfiguration/
your_matchmaking_configuration_name"
        }
      }
    }
  ]
}
```

7. Choose **Save changes**.

Configure a topic subscription to invoke a Lambda function

You can invoke a Lambda function using event notifications published to your Amazon SNS topic. When configuring the matchmaker, be sure to set the notification target to your SNS topic's ARN.

The following AWS CloudFormation template configures a subscription to an SNS topic named `MyFlexMatchEventTopic` to invoke a Lambda function named `FlexMatchEventHandlerLambdaFunction`. The template creates an IAM permissions policy that allows GameLift to write to the SNS topic. Finally, it adds permissions for the SNS topic to invoke the Lambda function.

```
FlexMatchEventTopic:
  Type: "AWS::SNS::Topic"
  Properties:
    KmsMasterKeyId: alias/aws/sns #Enables server-side encryption on the topic using an AWS
 managed key
    Subscription:
      - Endpoint: !GetAtt FlexMatchEventHandlerLambdaFunction.Arn
        Protocol: lambda
    TopicName: MyFlexMatchEventTopic

FlexMatchEventTopicPolicy:
  Type: "AWS::SNS::TopicPolicy"
  DependsOn: FlexMatchEventTopic
  Properties:
    PolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: gamelift.amazonaws.com
          Action:
            - "sns:Publish"
          Resource: !Ref FlexMatchEventTopic
    Topics:
      - Ref: FlexMatchEventTopic

FlexMatchEventHandlerLambdaPermission:
  Type: "AWS::Lambda::Permission"
  Properties:
    Action: "lambda:InvokeFunction"
    FunctionName: !Ref FlexMatchEventHandlerLambdaFunction
    Principal: sns.amazonaws.com
    SourceArn: !Ref FlexMatchEventTopic
```

Preparing games for FlexMatch

Use GameLift FlexMatch to add player matchmaking functionality to your games. FlexMatch is available with the managed GameLift solutions for custom game servers and Realtime Servers.

FlexMatch pairs the matchmaking service with a customizable rules engine. This lets you design how to match players together based on player attributes and game modes that make sense for your game, and rely on FlexMatch to manage the nuts and bolts of forming player groups and placing them into games. See more details about custom matchmaking in FlexMatch rule set examples (p. 25).

FlexMatch builds on the Queues feature. Once a match is formed, FlexMatch hands the match details to a queue of your choice. The queue searches for available hosting resources on your Amazon GameLift fleets and starts a new game session for the match.

The topics in this section cover how to add matchmaking support to your game servers and game clients. To create a matchmaker for your game, see Buiding a GameLift FlexMatch matchmaker (p. 10). For more information on how FlexMatch works, see How Amazon GameLift FlexMatch works (p. 2).

Add FlexMatch to a game client

This topic describes how to add FlexMatch matchmaking support to your client-side game services. The process is essentially the same whether you're using FlexMatch with GameLift managed hosting or with another hosting solution. To learn more about FlexMatch and how to set up a custom matchmaker for your games, see these topics:

- FlexMatch integration with GameLift hosting (p. 8)
- How Amazon GameLift FlexMatch works (p. 2)
- Buiding a GameLift FlexMatch matchmaker (p. 10)
- FlexMatch rule set examples (p. 25)

To enable FlexMatch matchmaking in your game, add the following functionality:

- Prepare to request matchmaking for one or multiple players (required).
- Track the status of matchmaking requests (required).
- Request player acceptance for a proposed match (optional).
- After a game session is created for the new match, get player connection information and join the game.

Prepare to request matchmaking for players

We highly recommend that your game client make matchmaking requests through a client-side game service. By using a trusted source, you can more easily protect against hacking attempts and fake player data. If your game has a session directory service, this is a good option for handling matchmaking requests.

To prepare your client service, do the following tasks:

- **Add the GameLift API.** Your client service uses functionality in the GameLift API, which is part of the AWS SDK. See GameLift SDKs for client services to learn more about the AWS SDK and download the latest version. Add this SDK to your game client service project.
- **Set up a matchmaking ticket system.** All matchmaking requests must be assigned a unique ticket ID. You need a mechanism to generate unique IDs and assign them to new match requests. A ticket ID can use any string format, up to a maximum of 128 characters.
- **Get matchmaker information.** Get the name of the matchmaking configuration that you plan to use. You also need the matchmaker's list of required player attributes, which are defined in the matchmaker's rule set.
- **Get player data.** Set up a way to get relevant data for each player. This includes player ID, player attribute values, and updated latency data for each region where the player is likely be slotted into a game.
- **(optional) Enable match backfill.** Decide how you want to backfill your existing matched games. If your matchmakers have backfill mode set to "manual", you may want to add backfill support to your game. If backfill mode is set to "automatic", you may need a way to turn it off for individual game sessions. Learn more about managing match backfill in Backfill existing games with FlexMatch (p. 49).

Request matchmaking for players

Add code to your client service to create and manage matchmaking requests to a FlexMatch matchmaker. The process of requesting FlexMatch matchmaking is identical for games that use FlexMatch with GameLiftmanaged hosting and for games that use FlexMatch as a standalone solution.

Create a matchmaking request:

- Call the GameLift API StartMatchmaking. Each request must contain the following information.

  Matchmaker

  The name of the matchmaking configuration to use for the request. FlexMatch places each request into the pool for the specified matchmaker, and the request is processed based on how the matchmaker is configured. This includes enforcing a time limit, whether to request player acceptance of matches, which queue to use when placing a resulting game session, etc. Learn more about matchmakers and rules sets in Design a FlexMatch matchmaker (p. 10).

  Ticket ID

  A unique ticket ID assigned to the request. Everything related to the request, including events and notifications, will reference the ticket ID.

  Player data

  List of players that you want to create a match for. If any of the players in the request do not meet match requirements, based on the match rules and latency minimums, the matchmaking request will never result in a successful match. You can include up to ten players in a match request. When there are multiple players in a request, FlexMatch tries to create a single match and assign all players to the same team (randomly selected). If a request contains too many players to fit in one of the match teams, the request will fail to be matched. For example, if you've set up your matchmaker to create 2v2 matches (two teams of two players), you cannot send a matchmaking request containing more than two players.

  Note
  A player (identified by their player ID) can only be included in one active matchmaking request at a time. When you create a new request for a player, any active matchmaking tickets with the same player ID are automatically canceled.

For each listed player, include the following data:

- *Player ID* – Each player must have a unique player ID, which you generate. See Generate player IDs.
- *Player attributes* – If the matchmaker in use calls for player attributes, the request must provide those attributes for each player. The required player attributes are defined in the matchmaker's rule set, which also specifies the data type for the attribute. A player attribute is optional only when the rule set specifies a default value for the attribute. If the match request does not provide required player attributes for all players, the matchmaking request can never succeed. Learn more about matchmaker rule sets and player attributes in Build a FlexMatch rule set (p. 15) and FlexMatch rule set examples (p. 25).
- *Player latencies* – If the matchmaker in use has a player latency rule, the request must report latency for each player. Player latency data is a list of one or more values per player. It represents the latency that the player experiences for regions in the matchmaker's queue. If no latency values for a player are included in the request, the player cannot be matched, and the request fails.

Retrieve match request details:

- Once a match request is sent, you can view the request details by calling DescribeMatchmaking with the request's ticket ID. This call returns the request information, including current status. Once a request has been successfully completed, the ticket also contains the information that a game client needs to connect to the match.

Cancel a match request:

- You can cancel a matchmaking request at any time by calling StopMatchmaking with the request's ticket ID.

Track matchmaking events

Set up notifications to track events that GameLift emits for matchmaking processes. You can set up notifications either directly, by creating an SNS topic, or by using Amazon EventBridge. For more information on setting up notifications, see Set up FlexMatch event notifications (p. 40). Once you've set up notifications, add a listener on your client service to detect the events and respond as needed.

It's also a good idea to back up notifications by periodically polling for status updates when a significant period of time passes without notification. To minimize impact on matchmaking performance, be sure to poll only after waiting at least 30 seconds after the matchmaking ticket was submitted or after the last received notification.

Retrieve a matchmaking request ticket, including current status, by calling DescribeMatchmaking with the request's ticket ID. We recommend polling no more than once every 10 seconds. This approach is for use during low-volume development scenarios only.

Note
You should set up your game with event notifications before you have high-volume matchmaking usage, such as with pre-production load testing. All games in public release should use notifications regardless of volume. The continuous polling approach is only appropriate for games in development with low matchmaking usage.

Request player acceptance

If you're using a matchmaker that has player acceptance turned on, add code to your client service to manage the player acceptance process. The process of managing player acceptances is identical for

games that use FlexMatch with GameLift-managed hosting and for games that use FlexMatch as a standalone solution.

Request player acceptance for a proposed match:

1. **Detect when a proposed match needs player acceptance.** Monitor the matchmaking ticket to detect when the status changes to `REQUIRES_ACCEPTANCE`. A change to this status triggers the FlexMatch event `MatchmakingRequiresAcceptance`.
2. **Get acceptances from all players.** Create a mechanism to present the proposed match details to every player in the matchmaking ticket. Players must be able to indicate that they either accept or reject the proposed match. You can retrieve match details by calling DescribeMatchmaking. Players have a limited time to respond before the matchmaker withdraws the proposed match and moves on.
3. **Report player responses to FlexMatch.** Report player responses by calling AcceptMatch with either accept or reject. All players in a matchmaking request must accept the match for it to go forward.
4. **Handle tickets with failed acceptances.** A request fails when any player in the proposed match either rejects the match or fails to respond by the acceptance time limit. Tickets for players who did accept the match are automatically returned to the ticket pool. Tickets for players who did not accept the match move to FAILURE status and are no longer processed. For tickets with multiple players, if any players in the ticket did not accept the match, the entire ticket fails.

Connect to a match

Add code to your client service to handle a successfully formed match (status `COMPLETED` or event `MatchmakingSucceeded`). This includes notifying the match's players and handing off connection information to their game clients.

For games that use GameLift managed hosting, when a matchmaking request is successfully fulfilled, the game session connection information is added to the matchmaking ticket. Retrieve a completed matchmaking ticket by calling DescribeMatchmaking. Connection information includes the game session's IP address and port, as well as a player session ID for each player ID. Learn more in GameSessionConnectionInfo. Your game client can use this information to connect directly to the game session for the match. The connection request should include a player session ID and a player ID. This data associates the connected player to the game session's match data, which includes team assignments (see GameSession).

For games that use other hosting solutions, including GameLift FleetIQ, you must build in a mechanism to enable match players to connect to the appropriate game session.

Sample matchmaking requests

The following code snippets build matchmaking requests for several different matchmakers. As described, a request must provide the player attributes that are required by the matchmaker in use, as defined in the matchmaker's rule set. The attribute provided must use the same data type, number (N) or string (S) that is defined in the rule set.

```
# Uses matchmaker for two-team game mode based on player skill level
def start_matchmaking_for_cowboys_vs_aliens(config_name, ticket_id, player_id, skill,
 team):
    response = gamelift.start_matchmaking(
        ConfigurationName=config_name,
        Players=[{
            "PlayerAttributes": {
                "skill": {"N": skill}
            },
            "PlayerId": player_id,
```

```
            "Team": team
        }],
        TicketId=ticket_id)

# Uses matchmaker for monster hunter game mode based on player skill level
def start_matchmaking_for_players_vs_monster(config_name, ticket_id, player_id, skill,
 is_monster):
    response = gamelift.start_matchmaking(
        ConfigurationName=config_name,
        Players=[{
            "PlayerAttributes": {
                "skill": {"N": skill},
                "desiredSkillOfMonster": {"N": skill},
                "wantsToBeMonster": {"N": int(is_monster)}
            },
            "PlayerId": player_id
        }],
        TicketId=ticket_id)

# Uses matchmaker for brawler game mode with latency
def start_matchmaking_for_three_team_brawler(config_name, ticket_id, player_id, skill,
 role):
    response = gamelift.start_matchmaking(
        ConfigurationName=config_name,
        Players=[{
            "PlayerAttributes": {
                "skill": {"N": skill},
                "character": {"S": [role]},
            },
            "PlayerId": player_id,
            "LatencyInMs": { "us-west-2": 20}
        }],
        TicketId=ticket_id)

# Uses matchmaker for multiple game modes and maps based on player experience
def start_matchmaking_for_multi_map(config_name, ticket_id, player_id, skill, maps, modes):
    response = gamelift.start_matchmaking(
        ConfigurationName=config_name,
        Players=[{
            "PlayerAttributes": {
                "experience": {"N": skill},
                "gameMode": {"SL": modes},
                "mapPreference": {"SL": maps}
            },
            "PlayerId": player_id
        }],
        TicketId=ticket_id)
```

Add FlexMatch to a GameLift-hosted game server

This topic describes how to add FlexMatch matchmaking support to custom game servers that are using GameLift managed hosting. To learn more about adding FlexMatch to your games, see these topics:

- How Amazon GameLift FlexMatch works (p. 2)
- FlexMatch integration with GameLift hosting (p. 8)

The information in this topic assumes that you've successfully integrated the GameLift Server SDK into your game server project, as described in Add GameLift to your game server. With this work completed, you have most of the mechanisms you need. The sections in this topic cover the remaining work to handle games that are set up with FlexMatch.

Set up your game server for matchmaking

To set up your game server to handle matched games, complete the following tasks.

1. **Start game sessions created with matchmaking.** To request a new game session, GameLift sends an on`StartGameSession()` request to your game server with a game session object (see GameSession). Your game server uses the game session information, including customized game data, to start the requested game session. For more details, see Start a game session.

   For matched games, the game session object also contains a set of matchmaker data. Matchmaker data includes information that your game server needs to start a new game session for the match. This includes the match's team structure, team assignments, and certain player attributes that may be relevant to your game. For example, your game might unlock certain features or levels based on the average player skill level, or choose a map based on players' preferences. Learn more in Work with matchmaker data (p. 48).
2. **Handle player connections.** When connecting to a matched game, a game client references a player ID and a player session ID (see Validate a new player). Your game server uses the player ID to associate an incoming player with player information in the matchmaker data. Matchmaker data identifies a player's team assignment and may provide other information to correctly represent the player in the game.
3. **Report when players leave a game.** Make sure that your game server is calling the Server API `RemovePlayerSession()` to report dropped players (see Report a player session ending). This step is important if you're using FlexMatch backfill to fill empty slots in existing games. It is critical if your game initiates backfill requests through a client-side game service. Learn more on implementing FlexMatch backfill in Backfill existing games with FlexMatch (p. 49).
4. **Request new players for existing matched game sessions (optional).** Decide how you want to backfill your existing matched games. If your matchmakers have backfill mode set to "manual", you may want to add backfill support to your game. If backfill mode is set to "automatic", you may need a way to turn it off for individual game sessions. For example, you might want to stop backfilling a game session once a certain point in the game is reached. Learn more about managing match backfill in Backfill existing games with FlexMatch (p. 49).

Work with matchmaker data

Your game server must be able to recognize and use the game information in a GameSession object. The GameLift service passes these objects to your game server whenever a game session is started or updated. Core game session information includes game session ID and name, maximum player count, connection information, and custom game data (if provided).

For game sessions that are created using FlexMatch, the `GameSession` object also contains a set of matchmaker data. In addition to a unique match ID, it identifies the matchmaker that created the match and describes the teams, team assignments, and players. It includes the player attributes from the original matchmaking request (see the Player object). It doesn't include the player latency; if you need latency data on current players, such as for match backfill, we recommend getting fresh data.

> **Note**
> Matchmaker data specifies the full matchmaking configuration ARN, which identifies the configuration name, AWS account, and region. When requesting match backfill from a game client or service, need the configuration name only. You can extract the configuration name by parsing out the string that follows ":matchmakingconfiguration/". In the example shown, the matchmaking configuration name is "MyMatchmakerConfig".

The following JSON shows a typical set of matchmaker data. This example describes a two-player game, with players matched based on skill ratings and highest level attained. The matchmaker also matched based on character, and ensured that matched players have at least one map preference in common. In

this scenario, the game server should be able to determine which map is most preferred and use it in the game session.

```
{
 "matchId":"1111aaaa-22bb-33cc-44dd-5555eeee66ff",
 "matchmakingConfigurationArn":"arn:aws:gamelift:us-
west-2:111122223333:matchmakingconfiguration/MyMatchmakerConfig",
 "teams":[
    {"name":"attacker",
  "players":[
           {"playerId":"4444dddd-55ee-66ff-77aa-8888bbbb99cc",
   "attributes":{
    "skills":{
     "attributeType":"STRING_DOUBLE_MAP",
     "valueAttribute":{"Body":10.0,"Mind":12.0,"Heart":15.0,"Soul":33.0}}
   }
  }]
 },{
  "name":"defender",
  "players":[{
   "playerId":"3333cccc-44dd-55ee-66ff-7777aaaa88bb",
   "attributes":{
    "skills":{
     "attributeType":"STRING_DOUBLE_MAP",
     "valueAttribute":{"Body":11.0,"Mind":12.0,"Heart":11.0,"Soul":40.0}}
   }
  }]
 }]
}
```

Backfill existing games with FlexMatch

Match backfill uses your FlexMatch mechanisms to find new players for existing matched game sessions. Although you can always add players to any game (see Join a player to a game session), match backfill ensures that new players meet the same match criteria as current players. In addition, match backfill assigns the new players to teams, manages player acceptance, and sends updated match information to the game server. Learn more about match backfill in FlexMatch matchmaking process (p. 4).

> **Note**
> FlexMatch backfill is not currently available for games using Realtime Servers.

There are two types of backfill mechanisms:

- To fill game sessions that start with fewer than the maximum allowed players, enable automatic backfill.
- To replace players who drop out of a game session in progress, add functionality to your game server to send backfill requests.

Turn on automatic backfill

With automatic match backfill, GameLift automatically triggers a backfill request whenever a game session starts with one or more unfilled player slots. This feature allows games to start as soon as the minimum number of matched players is found and fill remaining slots later as additional players are matched. You can opt to stop automatic backfill at any time.

As an example, consider a game that can hold six to ten players. FlexMatch initially locates six players, forms the match, and starts a new game session. With automatic backfill, the new game session can immediately request an additional four players. Depending on the game style, we might want to allow

new players to join at any time during the game session. Alternatively, we might want to stop automatic backfill after initial setup phase and before gameplay starts.

To add automatic backfill to your game, make the following updates to your game.

1. **Enable automatic backfill.** Automatic backfill is managed in a matchmaking configuration. When enabled, it is used with all matched game sessions that are created with that matchmaker. GameLift begins generating backfill requests for a non-full game session as soon as the game session starts up on a game server.

   To turn on automatic backfill, open a match configuration and set the backfill mode to "AUTOMATIC". For more details, see Create a matchmaking configuration (p. 12)
2. **Turn on backfill prioritization**. Customize your matchmaking process to prioritize filling backfill requests before creating new matches. In your matchmaking rule set, add an algorithm component and set backfill priority to "high". For more details, see Customize the match algorithm (p. 16).
3. **Update game session with new matchmaker data.** Amazon GameLift updates your game server with match information using the Server SDK callback function `onUpdateGameSession` (see Initialize the server process). Add code to your game server to handle updated game session objects as a result of backfill activity. Learn more in Update match data on the game server (p. 53).
4. **Turn off automatic backfill for a game session.** You can opt to stop automatic backfill at any point during an individual game session. To stop automatic backfill, add code to your game client or game server to make the GameLift API call StopMatchmaking. This call requires a ticket ID. Use the backfill ticket ID from the latest backfill request. You can get this information from the game session matchmaking data, which is updated as described in the previous step.

Send backfill requests (from a game server)

You can initiate match backfill requests directly from the game server process that is hosting the game session. The server process has the most up-to-date information on current players connected to the game and the status of empty player slots.

This topic assumes that you've already built the necessary FlexMatch components and successfully added matchmaking processes to your game server and a client-side game service. For more details on setting up FlexMatch, see FlexMatch integration with GameLift hosting (p. 8).

To enable match backfill for your game, add the following functionality:

- Send matchmaking backfill requests to a matchmaker and track the status of requests.
- Update match information for the game session. See Update match data on the game server (p. 53).

As with other server functionality, a game server uses the Amazon GameLift Server SDK. This SDK is available in C++ and C#.

To make match backfill requests from your game server, complete the following tasks.

1. **Trigger a match backfill request.** Generally, you want to initiate a backfill request whenever a matched game has one or more empty player slots. You may want to tie backfill requests to specific circumstances, such as to fill critical character roles or balance out teams. You'll likely also want to limit backfilling activity based on a game session's age.
2. **Create a backfill request.** Add code to create and send match backfill requests to a FlexMatch matchmaker. Backfill requests are handled using these server APIs:

   - StartMatchBackfill()
   - StopMatchBackfill()

To create a backfill request, call `StartMatchBackfill` with the following information. To cancel a backfill request, call `StopMatchBackfill` with the backfill request's ticket ID.

- **Ticket ID** — Provide a matchmaking ticket ID (or opt to have them autogenerated). You can use the same mechanism to assign ticket IDs to both matchmaking and backfill requests. Tickets for matchmaking and backfilling are processed the same way.
- **Matchmaker** — Identify which matchmaker to use for the backfill request. Generally, you'll want to use the same matchmaker that was used to create the original match. This request takes a matchmaking configuration ARN. This information is stored in the game session object (GameSession), which was provided to the server process by Amazon GameLift when activating the game session. The matchmaking configuration ARN is included in the `MatchmakerData` property.
- **Game session ARN** — Identify the game session being backfilled. You can get the game session ARN by calling the server API GetGameSessionId(). During the matchmaking process, tickets for new requests do not have a game session ID, while tickets for backfill requests do. The presence of at game session ID is one way to tell the difference between tickets for new matches and tickets for backfills.
- **Player data** — Include player information (Player) for all current players in the game session you are backfilling. This information allows the matchmaker to locate the best possible player matches for the players currently in the game session. You must include the team membership for every player. Do not specify a team if you are not using backfill. If your game server has been accurately reporting player connection status, you should be able to acquire this data as follows:
  1. The server process hosting the game session should have the most up-to-date information which players are currently connected to the game session.
  2. To get player IDs, attributes, and team assignments, pull player data from the game session object (GameSession), `MatchmakerData` property (see Work with matchmaker data (p. 48)). The matchmaker data includes all players who were matched to the game session, so you'll need to pull the player data for only the currently connected players.
  3. For player latency, if the matchmaker calls for latency data, collect new latency values from all current players and include it in each `Player` object. If latency data is omitted and the matchmaker has a latency rule, the request will not be successfully matched. Backfill requests require latency data only for the region that the game session is currently in. You can get a game session's region from the `GameSessionId` property of the `GameSession` object; this value is an ARN, which includes the region.

3. **Track the status of a backfill request.** Amazon GameLift updates your game server about the status of backfill requests using the Server SDK callback function `onUpdateGameSession` (see Initialize the server process). Add code to handle the status messages—as well as updated game session objects as a result of successful backfill requests—at Update match data on the game server (p. 53).

   A matchmaker can process only one match backfill request from a game session at a time. If you need to cancel a request, call StopMatchBackfill(). If you need to change a request, call `StopMatchBackfill` and then submit an updated request.

Send backfill requests (from a client service)

As an alternative to sending backfill requests from a game server, you may want to send them from a client-side game service. To use this option, the client-side service must have access to current data on game session activity and player connections; if your game uses a session directory service, this might be a good choice.

This topic assumes that you've already built the necessary FlexMatch components and successfully added matchmaking processes to your game server and a client-side game service. For more details on setting up FlexMatch, see FlexMatch integration with GameLift hosting (p. 8).

To enable match backfill for your game, add the following functionality:

- Send matchmaking backfill requests to a matchmaker and track the status of requests.
- Update match information for the game session. See Update match data on the game server (p. 53)

As with other client functionality, a client-side game service uses the AWS SDK with Amazon GameLift API. This SDK is available in C++, C#, and several other languages. For a general description of client APIs, see the Amazon GameLift Service API Reference, which describes the low-level service API for Amazon GameLift-related actions and includes links to language-specific reference guides.

To set up a client-side game service to backfill matched games, complete the following tasks.

1. **Trigger a request for backfilling.** Generally, a game initiates a backfill request whenever a matched game has one or more empty player slots. You may want to tie backfill requests to specific circumstances, such as to fill critical character roles or balance out teams. You'll likely also want to limit backfilling based on a game session's age. Whatever you use for a trigger, at a minimum you'll need to the following information. You can get this information from the game session object (GameSession) by calling DescribeGameSessions with a game session ID.

   - *Number of currently empty player slots*. This value can be calculated from a game session's maximum player limit and the current player count. Current player count is updated whenever your game server contacts the Amazon GameLift service to validate a new player connection or to report a dropped player.

   - *Creation policy*. This setting indicates whether the game session is currently accepting new players.

   The game session object contains other potentially useful information, including game session start time, custom game properties, and matchmaker data.
2. **Create a backfill request.** Add code to create and send match backfill requests to a FlexMatch matchmaker. Backfill requests are handled using these client APIs:

   - StartMatchBackfill
   - StopMatchmaking

   To create a backfill request, call `StartMatchBackfill` with the following information. A backfill request is similar to a matchmaking request (see Request matchmaking for players (p. 44)), but also identifies the existing game session. To cancel a backfill request, call `StopMatchmaking` with the backfill request's ticket ID.

   - **Ticket ID** — Provide a matchmaking ticket ID (or opt to have them autogenerated). You can use the same mechanism to assign ticket IDs to both matchmaking and backfill requests. Tickets for matchmaking and backfilling are processed the same way.
   - **Matchmaker** — Identify the name of a matchmaking configuration to use. Generally, you'll want to use the same matchmaker for backfilling that was used to create the original match. This information is in a game session object (GameSession), `MatchmakerData` property, under the matchmaking configuration ARN. The name value is the string following ""matchmakingconfiguration/". (For example, in the ARN value "arn:aws:gamelift:us-west-2:111122223333:matchmakingconfiguration/MM-4v4", the matchmaking configuration name is "MM-4v4".)

- **Game session ARN** — Specify the game session being backfilled. Use the `GameSessionId` property from the game session object; this ID uses the ARN value that you need. Matchmaking tickets (MatchmakingTicket) for backfill requests have the game session ID while being processed; tickets for new matchmaking requests do not get a game session ID until the match is placed; the presence of at game session ID is one way to tell the difference between tickets for new matches and tickets for backfills.
- **Player data** — Include player information (Player) for all current players in the game session you are backfilling. This information allows to matchmaker to locate the best possible player matches for the players currently in the game session. You must include the team membership for every player. Do not specify a team if you are not using backfill. If your game server has been accurately reporting player connection status, you should be able to acquire this data as follows:

  1. Call DescribePlayerSessions() with the game session ID to discover all players who are currently connected to the game session. Each player session includes a player ID. You can add a status filter to retrieve active player sessions only.
  2. Pull player data from the game session object (GameSession), `MatchmakerData` property (see Work with matchmaker data (p. 48). Use the player IDs acquired in the previous step to get data for currently connected players only. Since matchmaker data is not updated when players drop out, you will need extract the data for current players only.
  3. For player latency, if the matchmaker calls for latency data, collect new latency values from all current players and include it in the `Player` object. If latency data is omitted and the matchmaker has a latency rule, the request will not be successfully matched. Backfill requests require latency data only for the region that the game session is currently in. You can get a game session's region from the `GameSessionId` property of the `GameSession` object; this value is an ARN, which includes the region.

3. **Track status of backfill request.** Add code to listen for matchmaking ticket status updates. You can use the mechanism set up to track tickets for new matchmaking requests (see Track matchmaking events (p. 45)) using event notification (preferred) or polling. Although you don't need to trigger player acceptance activity with backfill requests, and player information is updated on the game server, you still need to monitor ticket status to handle request failures and resubmissions.

   A matchmaker can process only one match backfill request from a game session at a time. If you need to cancel a request, call StopMatchmaking. If you need to change a request, call `StopMatchmaking` and then submit an updated request.

   Once a match backfill request is successful, your game server receives an updated `GameSession` object and handles the tasks needed to join new players to the game session. See more at Update match data on the game server (p. 53).

Update match data on the game server

No matter how you initiate match backfill requests in your game, your game server must be able to handle the game session updates that Amazon GameLift delivers as a result of match backfill requests.

When Amazon GameLift completes a match backfill request—successfully or not—it calls your game server using the callback function `onUpdateGameSession`. This call has three input parameters: a match backfill ticket ID, a status message , and a GameSession object containing the most up-to-date matchmaking data including player information. You need to add the following code to your game server as part of your game server integration:

1. Implement the `onUpdateGameSession` function. This function must be able to handle the following status messages (`updateReason`):
   - MATCHMAKING_DATA_UPDATED – New players were successfully matched to the game session. The `GameSession` object contains updated matchmaker data, including player data on existing players and newly matched players.

- BACKFILL_FAILED – The match backfill attempt failed due to an internal error. The `GameSession` object is unchanged.
- BACKFILL_TIMED_OUT – The matchmaker failed to find a backfill match within the time limit. The `GameSession` object is unchanged.
- BACKFILL_CANCELLED – The match backfill request was canceled by a call to StopMatchmaking (client) or StopMatchBackfill (server). The `GameSession` object is unchanged.

2. For successful backfill matches, use the updated matchmaker data to handle the new players when they connect to the game session. At a minimum, you'll need to use the team assignments for the new player(s), as well as other player attributes that are required to get the player started in the game.
3. In your game server's call to the Server SDK action ProcessReady(), add the on`UpdateGameSession` callback method name as a process parameter.

GameLift FlexMatch reference

This section contains reference documentation for matchmaking with GameLift FlexMatch.

Topics

GameLift FlexMatch API reference (AWS SDK)

This topic provides a task-based list of API operations for GameLift FlexMatch. The GameLift FlexMatch service API is packaged into the AWS SDK in the `aws.gamelift` namespace. Download the AWS SDK or view the Amazon GameLift API reference documentation.

GameLift FlexMatch provides matchmaking services for use with games that are hosted with GameLift hosting solutions (including managed hosting for custom game servers or Realtime Servers, and hosting on Amazon EC2 with GameLift FleetIQ), as well as with other hosting systems such as peer-to-peer, on-premises, or cloud compute primitives. See the GameLift Developer Guide for more information on other GameLift hosting options.

Set up matchmaking rules and processes

Call these operations to create a FlexMatch matchmaker, configure the matchmaking process for your game, and define a set of custom rules for creating matches and teams.

Matchmaking configuration

- CreateMatchmakingConfiguration – Create a matchmaking configuration with instructions for evaluating groups of players and building player teams. When using GameLift for hosting, also specify how to create a new game session for the match.
- DescribeMatchmakingConfigurations – Retrieve matchmaking configurations defined a GameLift region.
- UpdateMatchmakingConfiguration – Change settings for matchmaking configuration. queue.
- DeleteMatchmakingConfiguration – Remove a matchmaking configuration from the region.

Matchmaking rule set

- CreateMatchmakingRuleSet – Create a set of rules to use when searching for player matches.
- DescribeMatchmakingRuleSets – Retrieve matchmaking rule sets defined in a GameLift region.
- ValidateMatchmakingRuleSet – Verify syntax for a set of matchmaking rules.
- DeleteMatchmakingRuleSet – Remove a matchmaking rule set from the region.

Request a match for a player or players

Call these operations from your game client service to manage player matchmaking requests.

- StartMatchmaking – Request matchmaking for one player or a group who want to play in the same match.
- DescribeMatchmaking – Get details on a matchmaking request, including status.
- AcceptMatch – For a match that requires player acceptance, notify GameLift when a player accepts a proposed match.
- StopMatchmaking – Cancel a matchmaking request.
- StartMatchBackfill - Request additional player matches to fill empty slots in an existing game session.

Available programming languages

The AWS SDK with support for GameLift is available in the following languages. For information about support for development environments, see the documentation for each language.

- C++ (SDK docs) (GameLift)
- Java (SDK docs) (GameLift)
- .NET (SDK docs) (GameLift)
- Go (SDK docs) (GameLift)
- Python (SDK docs) (GameLift)
- Ruby (SDK docs) (GameLift)
- PHP (SDK docs) (GameLift)
- JavaScript/Node.js (SDK docs) (GameLift)

FlexMatch rules language

The reference topics in this section describe the syntax and semantics that are used to build matchmaking rules for use with GameLift FlexMatch. For detailed help with writing matchmaking rules and rule sets, see Build a FlexMatch rule set (p. 15).

Topics

FlexMatch rule set schema

FlexMatch rule sets use standard schema for small-match and large-match rules. For detailed descriptions of each section, see FlexMatch rule set property definitions (p. 59).

Rule set schema for small matches

The following schema documents all possible properties and allowed values for a rule set that is used to build matches of up to 40 players.

```
{
    "name": "string",
    "ruleLanguageVersion": "1.0",
    "playerAttributes":[{
         "name": "string,
```

```
     "type": <"string", "number", "string_list", "string_number_map">,
     "default": "string"
}],
"algorithm": {
    "strategy": "exhaustiveSearch",
    "batchingPreference": <"random", "sorted">,
    "sortByAttributes": [ "string" ],
    "expansionAgeSelection": <"newest", "oldest">,
    "backfillPriority": <"normal", "low", "high">
},
"teams": [{
    "name": "string",
    "maxPlayers": number,
    "minPlayers": number,
    "quantity": integer
}],
"rules": [{
    "type": "distance",
    "name": "string",
    "description": "string",
    "measurements": "string",
    "referenceValue": number,
    "maxDistance": number,
    "minDistance": number,
    "partyAggregation": <"avg", "min", "max">
    },{
    "type": "comparison",
    "name": "string",
    "description": "string",
    "measurements": "string",
    "referenceValue": number,
    "operation": <"<", "<=", "=", "!=", ">", ">=">,
    "partyAggregation": <"avg", "min", "max">
    },{
    "type": "collection",
    "name": "string",
    "description": "string",
    "measurements": "string",
    "referenceValue": number,
    "operation": <"intersection", "contains", "reference_intersection_count">,
    "maxCount": number,
    "minCount": number,
    "partyAggregation": <"union", "intersection">
    },{
    "type": "latency",
    "name": "string",
    "description": "string",
    "maxLatency": number,
    "maxDistance": number,
    "distanceReference": number,
    "partyAggregation": <"avg", "min", "max">
    },{
    "type": "distanceSort",
    "name": "string",
    "description": "string",
    "sortDirection": <"ascending", "descending">,
    "sortAttribute": "string",
    "mapKey": <"minValue", "maxValue">,
    "partyAggregation": <"avg", "min", "max">
    },{
    "type": "absoluteSort",
    "name": "string",
    "description": "string",
    "sortDirection": <"ascending", "descending">,
    "sortAttribute": "string",
    "mapKey": <"minValue", "maxValue">,
```

```
        "partyAggregation": <"avg", "min", "max">
        },{
        "type": "compound",
        "name": "string",
        "description": "string",
        "statement": "string"
        }
    }],
    "expansions": [{
        "target": "string",
        "steps": [{
            "waitTimeSeconds": number,
            "value": number
        }, {
            "waitTimeSeconds": number,
            "value": number
        }]
    }]
}
```

Rule set schema for large matches

The following schema documents all possible properties and allowed values for a rule set that is used to build matches of greater than 40 players. If the total of `maxPlayers` values for all teams in the rule set exceeds 40, then FlexMatch processes match requests that use this rule set under the large-match guidelines.

```
{
    "name": "string",
    "ruleLanguageVersion": "1.0",
    "playerAttributes":[{
         "name": "string,
         "type": <"string", "number", "string_list", "string_number_map">,
         "default": "string"
    }],
    "algorithm": {
        "strategy": "balanced",
        "batchingPreference": <"largestPopulation", "fastestRegion">,
        "balancedAttribute": "string",
        "expansionAgeSelection": <"newest", "oldest">,
        "backfillPriority": <"normal", "low", "high">
    },
    "teams": [{
        "name": "string",
        "maxPlayers": number,
        "minPlayers": number,
        "quantity": integer
    }],
    "rules": [{
        "name": "string",
        "type": "latency",
        "description": "string",
        "maxLatency": number,
        "partyAggregation": <"avg", "min", "max">
    }, {
        "name": "string",
        "type": "batchDistance",
        "batchAttribute": "string",
        "maxDistance": number
    }],
    "expansions": [{
        "target": "string",
        "steps": [{
```

```
            "waitTimeSeconds": number,
            "value": number
        }, {
            "waitTimeSeconds": number,
            "value": number
        }]
    }]
}
```

FlexMatch rule set property definitions

This section defines each property in the rule set schema. For additional help with creating a rule set, see Build a FlexMatch rule set (p. 15).

`name`

A descriptive label for the rule set. This value is not associated with the name assigned to the GameLift MatchmakingRuleSet resource. This value is included in the matchmaking data describing a completed match, but it not used by any GameLift processes.

Allowed values: String

Required? No

`ruleLanguageVersion`

The version of the FlexMatch property expression language being used.

Allowed values: "1.0"

Required? Yes

`playerAttributes`

A collection of player data that is included in matchmaking requests and is used in the matchmaking process. You can also declare attributes here to have the player data included in the matchmaking data that is passed to game servers, even if the data is not used in the matchmaking process.

Required? No

`name`

A unique name for player attribute to be used by matchmaker. This name must match the player attribute name that is referenced in matchmaking requests.

Allowed values: String

Required? Yes

`type`

The data type of the player attribute value.

Allowed values: "string", "number", "string_list", "string_number_map"

Required? Yes

`default`

A default value to use when a matchmaking request does not provide one for a player.

Allowed values: Any value allowed for the player attribute.

Required? No

`algorithm`

Optional configuration settings to customize the matchmaking process.

Required? No

`strategy`

The method to use when building matches. If this property is not set, the default behavior is "exhaustiveSearch".

Allowed values:

- "exhaustiveSearch" – Standard matching method. FlexMatch forms a match around the oldest ticket in a batch by evaluating other tickets in the pool based a set of custom match rules. This strategy is used for matches of 40 players or fewer. When using this strategy, `batchingPreference` should be set to either "random" or "sorted".
- "balanced" – Method that's optimized to form large matches quickly. This strategy is used only for matches of 41 to 200 players. It forms matches by pre-sorting the ticket pool, building potential matches and assigning players to teams, and then balancing each team in a match using a specified player attribute. For example, this strategy can be used to equalize the average skill levels of all teams in a match. When using this strategy, `balancedAttribute` must be set, and `batchingPreference` should be set to either "largestPopulation" or "fastestRegion". Most custom rule types are not recognized with this strategy.

Required? Yes

`batchingPreference`

The pre-sorting method to use before grouping tickets for match building. Pre-sorting the ticket pool causes tickets to be batched together based on a specific characteristic, which tends to increase uniformity across players in the final matches.

Allowed values:

- "random" – Valid only with `strategy` = "exhaustiveSearch". No pre-sorting is done; tickets in the pool are randomly batched. This is the default behavior for an exhaustive search strategy.
- "sorted" – Valid only with `strategy` = "exhaustiveSearch". The ticket pool is pre-sorted based on the player attributes listed in `sortbyAttributes`.
- "largestPopulation" – Valid only with `strategy` = "balanced". The ticket pool is pre-sorted by regions where players are reporting acceptable latency levels. This is the default behavior for a balanced strategy.
- "fastestRegion" – Valid only with `strategy` = "balanced". The ticket pool is pre-sorted by regions where players are reporting their lowest latency levels. Resulting matches take longer to complete, but latency for all players tends to be low.

Required? Yes

`balancedAttribute`

The name of a player attribute to use when building large matches with the balanced strategy.

Allowed values: Any attribute declared in `playerAttributes` with `type` = "number".

Required? Yes, if `strategy` = "balanced".

`sortByAttributes`

A list of player attributes to use when pre-sorting the ticket pool prior to batching. This property is only used when pre-sorting with the exhaustive search strategy. The order of the attribute list determines sort order. FlexMatch uses standard sorting convention for alpha and numeric values.

Allowed values: Any attribute declared in `playerAttributes`.

Required? Yes, if `batchingPreference` = "sorted".

`backfillPriority`

The prioritization method for matching backfill tickets. This property determines when FlexMatch processes the backfill tickets in a batch. It is only used when pre-sorting with the exhaustive search strategy. If this property is not set, the default behavior is "normal".

Allowed values:

- "normal" – A ticket's request type (backfill or new match) is not considered when forming matches.
- "high" – A ticket batch is sorted by request type (and then by age), and FlexMatch attempts to match backfill tickets first.
- "low" – A ticket batch is sorted by request type (and then by age), and FlexMatch attempts to match non-backfill tickets first.

Required? No

`expansionAgeSelection`

The method for calculating the wait time for a match rule expansion. Expansions are used to relax match requirements if a match hasn't been completed after a certain amount of time passes. Wait time is calculated based on the age of tickets that are already in the partially filled match. If this property is not set, the default behavior is "newest".

Allowed values:

- "newest" – Expansion wait time is calculated based on the ticket with the most recent creation timestamp in the partially completed match. Expansions tend to be triggered more slowly, because one newer ticket can restart the wait time clock.
- "oldest" – Expansion wait time is calculated based on the ticket with the oldest creation timestamp in the match. Expansions tend to be triggered more quickly.

Required? No

`teams`

The configuration of teams in a match. Provide a team name and size range for each team. A rule set must define at least one team.

`name`

A unique name for the team. Team names can be referred to in rules and expansions. On a successful match, players are assigned by team name in the matchmaking data.

Allowed values: String

Required? Yes

`maxPlayers`

The maximum number of players that can be assigned to the team.

Allowed values: Number

Required? Yes

`minPlayers`

The minimum number of players that must be assigned to the team before the match is viable.

Allowed values: Number

Required? Yes

`quantity`

The number of teams of this type to create in a match. Teams with quantities greater than 1 are designated with an appended number ("Red_1", "Red_2", etc.). If this property is not set, the default value is "1".

Allowed values: Number

Required? No

`rules`

A collection of rule statements that define how to evaluate players for a match.

Required? No

`name`

A unique name for the rule. All rules in a rule set must have unique names. Rule names are referenced in event logs and metrics that track activity related to the rule.

Allowed values: String

Required? Yes

`description`

A text description for the rule. This information can be used to identify the purpose of a rule. It is not used in the matchmaking process.

Allowed values: String

Required? No

`type`

The type of rule statement. Each rule type has additional properties that must be set. For more details on the structure and use of each rule type, see FlexMatch rule types (p. 63).

Allowed values:

- "absoluteSort" – Sorts using an explicit sorting method that orders tickets in a batch based on whether a specified player attribute compares to the oldest ticket in the batch.
- "collection" – Evaluates the values in a collection, such as a player attribute that's a collection, or a set of values for multiple players.
- "comparison" – Compares two values.
- "compound" – Defines a compound matchmaking rule using a logical combination of other rules in the rule set. Supported only for matches of 40 or fewer players.
- "distance" – Measures the distance between number values.
- "batchDistance" – Measures the difference between an attribute value and uses it to group match requests.
- "distanceSort" – Sorts using an explicit sorting method that orders tickets in a batch based on how a specified player attribute with a numerical value compares to the oldest ticket in the batch.
- "latency" – Evaluates the regional latency data that is reported for a matchmaking request.

Required? Yes

`expansions`

Rules for relaxing match requirements over time when a match cannot be completed. Set up expansions as a series of steps that apply gradually in order to make matches easier to find.

By default, FlexMatch calculates wait time based on the age of the newest ticket added to a match. You can change how expansion wait times are calculated using the algorithm property `expansionAgeSelection`.

Expansion wait times are absolute values, so each step should have a wait time longer than the previous step. For example, to schedule a gradual series of expansion, you might use wait times of 30 seconds, 40 seconds, and 50 seconds. Wait times cannot exceed the maximum time allowed for a match request, which is set in the matchmaking configuration.

Required? No

`target`

The rule set element to be relaxed. You can relax team size properties or any rule statement property. The syntax is "<component name>[<rule/team name>].<property name>". For example, to change team minimum sizes: `teams[Red, Yellow].minPlayers`. To change the minimum skill requirement in a comparison rule statement named "minSkill": `rules[minSkill].referenceValue`.

Required? Yes

`steps`

`waitTimeSeconds`

The length of time, in seconds, to wait before applying the new value for the target rule set element.

Required? Yes

`value`

The new value for the target rule set element.

FlexMatch rule types

Batch distance rule

```
batchDistance
```

Batch distance rules measure the difference between two attribute values. You can use the batch distance rule type with both large and small matches. There are two types of batch distance rules:

- *Compare numerical attribute values*. For example, a batch distance rule of this type might require that all players in a match be within two skill levels of each other. For this type, define a maximum distance between the `batchAttribute` of all tickets.
- *Compare string attribute values*. For example, a batch distance rule of this type might require that all players in a match request the same game mode. For this type, define a `batchAttribute` value that FlexMatch uses to form batches.

Batch distance rule properties

- **`batchAttribute`** – The player attribute value used to form batches.
- **`maxDistance`** – The maximum distance value for a successful match. Used to compare numerical attributes.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`), maximum (`max`), and average (`avg`) values for a ticket's players. The default is `avg`.

Example

Examples

```
{
  "name":"SimilarSkillRatings",
  "description":"All players must have similar skill ratings",
  "type":"batchDistance",
  "batchAttribute":"SkillRating",
  "maxDistance":"500"
}
```

```
{
  "name":"SameGameMode",
  "description":"All players must have the same game mode",
  "type":"batchDistance",
  "batchAttribute":"GameMode"
}
```

Comparison rule

```
comparison
```

Comparison rules compare a player attribute value to another value. There are two types of comparison rules:

- *Compare to reference value*. For example, a comparison rule of this type might require that matched players have a certain skill level or greater. For this type, specify a player attribute, reference value, and a comparison operation.
- *Compare across players*. For example, a comparison rule of this type might require that all players in the match use different characters. For this type, specify a player attribute and either the equal (=) or not-equal (!=) comparison operation. Don't specify a reference value.

> **Note**
> Batch distance rules are more efficient for comparing player attributes. To reduce matchmaking latency, use a batch distance rule when possible.

Comparison rule properties

- **`measurements`** – The player attribute value to compare.
- **`referenceValue`** – The value to compare the measurement to for a prospective match.
- **`operation`** – The value that determines how to compare the measurement to the reference value. Valid operations include: <, <=, =, !=, >, >=.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`), maximum (`max`), and average (`avg`) values for a ticket's players. The default is `avg`.

Distance rule

```
distance
```

Distance rules measure the difference between two number values, such as the distance between player skill levels. For example, a distance rule might require that all players have played the game for at least 30 hours.

Note
Batch distance rules are more efficient for comparing player attributes. To reduce matchmaking latency, use a batch distance rule when possible.

Distance rule properties

- **`measurements`** – The player attribute value to measure distance for. This must be an attribute with a numerical value.
- **`referenceValue`** – The numerical value to measure distance against for a prospective match.
- **`minDistance/maxDistance`** – The minimum or maximum distance value for a successful match.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`), maximum (`max`), and average (`avg`) values for a ticket's players. The default is `avg`.

Collection rule

```
collection
```

Collection rules compare a group of player attribute values to those of other players in the batch or to a reference value. A collection can contain attribute values for multiple players, a player attribute as a string list, or both. For example, a collection rule might look at the characters that the players in a team choose. The rule might then require the team to have at least one of a certain character.

Collection rule properties

- **`measurements`** – The collection of player attribute values to compare. The attribute values must be string lists.
- **`referenceValue`** – The value (or collection of values) to use to compare measurements for a prospective match.
- **`operation`** – The value that determines how to compare a collection of measurements. Valid operations include the following:
  - `intersection` – This operation measures the number of values that are the same in all players' collections. For an example of a rule that uses the intersection operation, see Example 4: Use explicit sorting to find best matches (p. 30).
  - `contains` – This operation measures the number of player attribute collections that contain the specified reference value. For an example of a rule that uses the contains operation, see Example 3: Set team-level requirements and latency limits (p. 28).
  - `reference_intersection_count` – This operation measures the number of items in a player attribute collection that match items in the reference value collection. You can use this operation to compare multiple different player attributes. For an example of a rule that compares multiple player attribute collections, see Example 5: Find intersections across multiple player attributes (p. 32).
- **`minCount/maxCount`** – The minimum or maximum count value for a successful match.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). For this value, you can use `union` to combine the player attributes of all players in the party. Or, you can use `intersection` to use player attributes that the party has in common. The default is `union`.

Compound rule

```
compound
```

Compound rules use logical statements to form matches of 40 or fewer players. You can use multiple compound rules in a single rule set. When using multiple compound rules, all compound rules must be true to form a match.

You can't expand a compound rule using expansion rules (p. 18), but you can expand underlying or supporting rules.

Compound rule properties

- **`statement`** – The logic used to combine individual rules to form the compound rule. The rules that you specify in this property must have been defined earlier in your rule set. You can't use `batchDistance` rules in a compound rule.

  This property supports the following logical operators:
  - `and` – The expression is true if the two provided arguments are true.
  - `or` – The expression is true if either of the two provided arguments are true.
  - `not` – Reverses the outcome of the argument in the expression.
  - `xor` – The expression is true if only one of the arguments is true.

Example Example

The following example matches players of varying skill levels based on the game mode that they select.

```
{
    "name": "CompoundRuleExample",
    "type": "compound",
    "statement": "or(and(SeriousPlayers, VeryCloseSkill), and(CasualPlayers,
 SomewhatCloseSkill))"
}
```

Latency rule

```
latency
```

Latency rules measure player latency per location. A latency rule ignores any location with a latency higher than the maximum. A player must have a latency value below the maximum in at least one location for the latency rule to accept them. You can use this rule type with large matches by specifying the `maxLatency` property.

Latency rule properties

- **`maxLatency`** – The maximum acceptable latency value for a location. If a ticket has no locations with latency under the maximum, then the ticket doesn't match the latency rule.
- **`maxDistance`** – The maximum value between the latency of each ticket and the distance reference value.
- **`distanceReference`** – The latency value to compare ticket latency with. Tickets within the maximum distance of the distance reference value result in a successful match. Valid options include the minimum (`min`) and average (`avg`) player latency values.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`), maximum (`max`), and average (`avg`) values for a ticket's players. The default is `avg`.

> **Note**
> A queue can place a game session in a Region that doesn't match a latency rule. For more information about latency policies for queues, see Create a player latency policy.

Absolute sort rule

```
absoluteSort
```

Absolute sort rules sort a batch of matchmaking tickets based on a specified player attribute compared to the first ticket added to the batch.

Absolute sort rule properties

- **`sortDirection`** – The order to sort the matchmaking tickets in. Valid options include `ascending` and `descending`.
- **`sortAttribute`** – The player attribute to sort tickets by.
- **`mapKey`** – The options to sort the player attribute if it's a map. Valid options include:
  - `minValue` – The key with the lowest value is first.
  - `maxValue` – The key with the highest value is first.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`) player attribute, the maximum (`max`) player attribute, and the average (`avg`) of all player attributes for players in the party. The default is `avg`.

Example

Example

The following example rule sorts players by skill level and averages the skill level of parties.

```
{
    "name":"AbsoluteSortExample",
    "type":"absoluteSort",
    "sortDirection":"ascending",
    "sortAttribute":"skill",
    "partyAggregation":"avg"
}
```

Distance sort rule

```
distanceSort
```

Distance sort rules sort a batch of matchmaking tickets based on the distance of a specified player attribute from the first ticket added to the batch.

Distance sort rule properties

- **`sortDirection`** – The direction to sort matchmaking tickets. Valid options include `ascending` and `descending`.
- **`sortAttribute`** – The player attribute to sort tickets by.
- **`mapKey`** – The options to sort the player attribute if it's a map. Valid options include:
  - `minValue` – For the first ticket added to the batch, find the key with the lowest value.
  - `maxValue` – For the first ticket added to the batch, find the key with the highest value.
- **`partyAggregation`** – The value that determines how FlexMatch handles tickets with multiple players (parties). Valid options include the minimum (`min`), maximum (`max`), and average (`avg`) values for a ticket's players. The default is `avg`.

FlexMatch property expressions

Property expressions can be used to define certain matchmaking-related properties. They allow you to use calculations and logic when defining a property value. Property expressions generally result in one of two forms:

- Individual player data.
- Calculated collections of individual player data.

Common matchmaking property expressions

A property expression identifies a specific value for a player, team, or match. The following partial expressions illustrate how to identify teams and players:

Goal	Input	Meaning	Output
To identify a specific team in a match:	`teams[red]`	The Red team	Team
To identify a set of specific teams in a match:	`teams[red,blue]`	The Red team and the Blue team	List<Team>
To identify all teams in a match:	`teams[*]`	All teams	List<Team>
To identify players in a specific team:	`team[red].players`	Players in the Red team	List<Player>
To identify players in a set of specific teams in a match:	`team[red,blue].players`	Players in the match, grouped by team	List<List<Player>>
To identify players in a match:	`team[*].players`	Players in the match, grouped by team	List<List<Player>>

Property expression examples

The following table illustrates some property expressions that build on the previous examples:

Expression	Meaning	Resulting Type
`teams[red].players[playerid]`	The player IDs of all players on the red team	List<string>
`teams[red].players.attributes["skill"]`	The "skill" attributes of all players on the red team	List<number>
`teams[red,blue].players.attributes["skill"]`	The "skill" attributes of all players on the Red team and the Blue team, grouped by team	List<List<number>>
`teams[*].players.attributes["skill"]`	The "skill" attributes of all players in the match, grouped by team	List<List<number>>

Property expression aggregations

Property expressions can be used to aggregate team data by using the following functions or combinations of functions:

Aggregation	Input	Meaning	Output
`min`	List<number>	Get the minimum of all numbers in the list.	number
`max`	List<number>	Get the maximum of all numbers in the list.	number
`avg`	List<number>	Get the average of all numbers in the list.	number
`median`	List<number>	Get the median of all numbers in the list.	number
`sum`	List<number>	Get the sum of all numbers in the list.	number
`count`	List<?>	Get the number of elements in the list.	number
`stddev`	List<number>	Get the standard deviation of all numbers in the list.	number
`flatten`	List<List<?>>	Turn a collection of nested lists into a single list containing all elements.	List<?>
`set_intersection`	List<List<string>>	Get a list of strings that are found in all string lists in a collection.	List<string>
All above	List<List<?>>	All operations on a nested list operate on each sublist individually to produce a list of results.	List<?>

The following table illustrates some valid property expressions that use aggregation functions:

Expression	Meaning	Resulting Type
flatten(teams[*].players.attributes[skill])	The "skill" attributes of all players in the match (not grouped)	List<number>
avg(teams[red].players.attributes[skill])	The average skill of the red team players	number
avg(teams[*].players.attributes[skill])	The average skill of each team in the match	List<number>

Expression	Meaning	Resulting Type
avg(flatten(teams[*].players.attributes[skill]))	The average skill level of all players in the match. This expression gets a flattened list of player skills and then averages them.	number
count(teams[red].players)	The number of players on the red team	number
count (teams[*].players)	The number of players on each team in the match	List<number>
max(avg(teams[*].players.attributes[skill]))	The highest team skill level in the match	number

FlexMatch matchmaking events

GameLift FlexMatch emits events for each matchmaking ticket as it is processed. You can publish these events to an Amazon SNS topic, as described in Set up FlexMatch event notifications (p. 40). These events are also emitted to Amazon CloudWatch Events in near real time and on a best-effort basis.

This topic describes the structure of FlexMatch events and provides an example for each event type. For more information on matchmaking ticket statuses, see MatchmakingTicket in the *Amazon GameLift API Reference*.

MatchmakingSearching

Ticket has been entered into matchmaking. This includes new requests and requests that were part of a proposed match that failed.

Resource: ConfigurationArn

Detail: type, tickets, estimatedWaitMillis, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "cc3d3ebe-1d90-48f8-b268-c96655b8f013",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-08T21:15:36.421Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-08T21:15:35.676Z",
        "players": [
          {
```

```
            "playerId": "player-1"
          }
        ]
      }
    ],
    "estimatedWaitMillis": "NOT_AVAILABLE",
    "type": "MatchmakingSearching",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1"
        }
      ]
    }
  }
}
```

PotentialMatchCreated

A potential match has been created. This is emitted for all new potential matches, regardless of whether acceptance is required.

Resource: ConfigurationArn

Detail: type, tickets, acceptanceTimeout, acceptanceRequired, ruleEvaluationMetrics, gameSessionInfo, matchId

Example

```
{
  "version": "0",
  "id": "fce8633f-aea3-45bc-aeba-99d639cad2d4",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-08T21:17:41.178Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-08T21:15:35.676Z",
        "players": [
          {
            "playerId": "player-1",
            "team": "red"
          }
        ]
      },
      {
        "ticketId": "ticket-2",
        "startTime": "2017-08-08T21:17:40.657Z",
        "players": [
          {
            "playerId": "player-2",
            "team": "blue"
          }
        ]
      }
```

```
    ],
    "acceptanceTimeout": 600,
    "ruleEvaluationMetrics": [
      {
        "ruleName": "EvenSkill",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "EvenTeams",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "FastConnection",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "NoobSegregation",
        "passedCount": 3,
        "failedCount": 0
      }
    ],
    "acceptanceRequired": true,
    "type": "PotentialMatchCreated",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1",
          "team": "red"
        },
        {
          "playerId": "player-2",
          "team": "blue"
        }
      ]
    },
    "matchId": "3faf26ac-f06e-43e5-8d86-08feff26f692"
  }
}
```

AcceptMatch

Players have accepted a potential match. This event contains the current acceptance status of each player in the match. Missing data means that AcceptMatch hasn't been called for that player.

Resource: ConfigurationArn

Detail: type, tickets, matchId, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "b3f76d66-c8e5-416a-aa4c-aa1278153edc",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-09T20:04:42.660Z",
  "region": "us-west-2",
  "resources": [
```

```
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-09T20:01:35.305Z",
        "players": [
          {
            "playerId": "player-1",
            "team": "red"
          }
        ]
      },
      {
        "ticketId": "ticket-2",
        "startTime": "2017-08-09T20:04:16.637Z",
        "players": [
          {
            "playerId": "player-2",
            "team": "blue",
            "accepted": false
          }
        ]
      }
    ],
    "type": "AcceptMatch",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1",
          "team": "red"
        },
        {
          "playerId": "player-2",
          "team": "blue",
          "accepted": false
        }
      ]
    },
    "matchId": "848b5f1f-0460-488e-8631-2960934d13e5"
  }
}
```

AcceptMatchCompleted

Match acceptance is complete due to player acceptance, player rejection, or acceptance timeout.

Resource: ConfigurationArn

Detail: type, tickets, acceptance, matchId, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "b1990d3d-f737-4d6c-b150-af5ace8c35d3",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-08T20:43:14.621Z",
  "region": "us-west-2",
  "resources": [
```

```
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-08T20:30:40.972Z",
        "players": [
          {
            "playerId": "player-1",
            "team": "red"
          }
        ]
      },
      {
        "ticketId": "ticket-2",
        "startTime": "2017-08-08T20:33:14.111Z",
        "players": [
          {
            "playerId": "player-2",
            "team": "blue"
          }
        ]
      }
    ],
    "acceptance": "TimedOut",
    "type": "AcceptMatchCompleted",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1",
          "team": "red"
        },
        {
          "playerId": "player-2",
          "team": "blue"
        }
      ]
    },
    "matchId": "a0d9bd24-4695-4f12-876f-ea6386dd6dce"
  }
}
```

MatchmakingSucceeded

Matchmaking has successfully completed and a game session has been created.

Resource: ConfigurationArn

Detail: type, tickets, matchId, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "5ccb6523-0566-412d-b63c-1569e00d023d",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-09T19:59:09.159Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
```

```
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-09T19:58:59.277Z",
        "players": [
          {
            "playerId": "player-1",
            "playerSessionId": "psess-6e7c13cf-10d6-4756-a53f-db7de782ed67",
            "team": "red"
          }
        ]
      },
      {
        "ticketId": "ticket-2",
        "startTime": "2017-08-09T19:59:08.663Z",
        "players": [
          {
            "playerId": "player-2",
            "playerSessionId": "psess-786b342f-9c94-44eb-bb9e-c1de46c472ce",
            "team": "blue"
          }
        ]
      }
    ],
    "type": "MatchmakingSucceeded",
    "gameSessionInfo": {
      "gameSessionArn": "arn:aws:gamelift:us-west-2:123456789012:gamesession/836cf48d-
bcb0-4a2c-bec1-9c456541352a",
      "ipAddress": "192.168.1.1",
      "port": 10777,
      "players": [
        {
          "playerId": "player-1",
          "playerSessionId": "psess-6e7c13cf-10d6-4756-a53f-db7de782ed67",
          "team": "red"
        },
        {
          "playerId": "player-2",
          "playerSessionId": "psess-786b342f-9c94-44eb-bb9e-c1de46c472ce",
          "team": "blue"
        }
      ]
    },
    "matchId": "c0ec1a54-7fec-4b55-8583-76d67adb7754"
  }
}
```

MatchmakingTimedOut

Matchmaking ticket has failed by timing out.

Resource: ConfigurationArn

Detail: type, tickets, ruleEvaluationMetrics, message, matchId, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "fe528a7d-46ad-4bdc-96cb-b094b5f6bf56",
  "detail-type": "GameLift Matchmaking Event",
```

```
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-09T20:11:35.598Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "reason": "TimedOut",
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-09T20:01:35.305Z",
        "players": [
          {
            "playerId": "player-1",
            "team": "red"
          }
        ]
      }
    ],
    "ruleEvaluationMetrics": [
      {
        "ruleName": "EvenSkill",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "EvenTeams",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "FastConnection",
        "passedCount": 3,
        "failedCount": 0
      },
      {
        "ruleName": "NoobSegregation",
        "passedCount": 3,
        "failedCount": 0
      }
    ],
    "type": "MatchmakingTimedOut",
    "message": "Removed from matchmaking due to timing out.",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1",
          "team": "red"
        }
      ]
    }
  }
}
```


MatchmakingCancelled

Matchmaking ticket has been canceled.

Resource: ConfigurationArn

Detail: type, tickets, ruleEvaluationMetrics, message, matchId, gameSessionInfo

Example


```
{
  "version": "0",
  "id": "8d6f84da-5e15-4741-8d5c-5ac99091c27f",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-09T20:00:07.843Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "reason": "Cancelled",
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-09T19:59:26.118Z",
        "players": [
          {
            "playerId": "player-1"
          }
        ]
      }
    ],
    "ruleEvaluationMetrics": [
      {
        "ruleName": "EvenSkill",
        "passedCount": 0,
        "failedCount": 0
      },
      {
        "ruleName": "EvenTeams",
        "passedCount": 0,
        "failedCount": 0
      },
      {
        "ruleName": "FastConnection",
        "passedCount": 0,
        "failedCount": 0
      },
      {
        "ruleName": "NoobSegregation",
        "passedCount": 0,
        "failedCount": 0
      }
    ],
    "type": "MatchmakingCancelled",
    "message": "Cancelled by request.",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1"
        }
      ]
    }
  }
}
```

MatchmakingFailed

Matchmaking ticket has encountered an error. This may be due to the game session queue not accessible or to an internal error.

Resource: ConfigurationArn

Detail: type, tickets, ruleEvaluationMetrics, message, matchId, gameSessionInfo

Example

```
{
  "version": "0",
  "id": "025b55a4-41ac-4cf4-89d1-f2b3c6fd8f9d",
  "detail-type": "GameLift Matchmaking Event",
  "source": "aws.gamelift",
  "account": "123456789012",
  "time": "2017-08-16T18:41:09.970Z",
  "region": "us-west-2",
  "resources": [
    "arn:aws:gamelift:us-west-2:123456789012:matchmakingconfiguration/SampleConfiguration"
  ],
  "detail": {
    "tickets": [
      {
        "ticketId": "ticket-1",
        "startTime": "2017-08-16T18:41:02.631Z",
        "players": [
          {
            "playerId": "player-1",
            "team": "red"
          }
        ]
      }
    ],
    "customEventData": "foo",
    "type": "MatchmakingFailed",
    "reason": "UNEXPECTED_ERROR",
    "message": "An unexpected error was encountered during match placing.",
    "gameSessionInfo": {
      "players": [
        {
          "playerId": "player-1",
          "team": "red"
        }
      ]
    },
    "matchId": "3ea83c13-218b-43a3-936e-135cc570cba7"
  }
}
```

Security with FlexMatch

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. For information on how to apply the shared responsibility model when using FlexMatch, see Security in Amazon GameLift.

GameLift FlexMatch release notes and SDK versions

The GameLift release notes provide details about new FlexMatch features, updates, and fixes related to the service. This page also includes GameLift SDK version history.

GameLift developer resources

To view all GameLift documentation and developer resources, see the Amazon GameLift Documentation home page.

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.